Parallelism and Compilers

Christoph W. Keßler

Habilitationsschrift

Fachbereich IV
Universität Trier

December 13, 2000

Preface

This thesis describes some of my contributions to several aspects of the large and rapidly growing area of compiling and programming for parallel computer architectures.

It is submitted as habilitation thesis to the Fachbereich IV of the University of Trier, Germany, in partial fulfillment of the pre-requirements for the habilitation examination.

I assure that I have written this thesis myself, and that I have given proper credit to all ideas contributed by other people or taken from the literature, according to my best knowledge. Most of the results described here have been previously published in international refereed journals, conferences, workshops, and in technical reports, and part of the material presented in Chapters 4 and 5 follows my contributions to a recent textbook entitled *Practical PRAM Programming* [B1]. For this reason, this thesis will not be published in its present form. Some ideas and results are, however, new and by now unpublished. New work can be found in each of the main chapters; most of this was developed in parallel to writing this thesis. References to my own publications are marked and listed separately; the complete list of my publications can be found in Appendix D after the general bibliography. A part of the work has been done jointly with other researchers, who are explicitly mentioned in the introduction and in the acknowledgement sections at the end of each main chapter.

The introductory chapter (Chapter 1) gives a personalized view of the main topics discussed in this thesis. In particular, it gives a comprehensive survey of my scientific work and my publications of the last decade.

The main body of this thesis is structured into four main chapters: Chapters 2, 3, 4, and 5. Addressing different aspects of the common area of programming and compiling for parallel architectures, Chapters 2, 3, and 4 are self-contained and may be read independently. Chapter 5 discusses compilation aspects of the parallel programming languages introduced in Chapter 4 and thus depends partially on that chapter.

The appendices contain background material, longer proofs and longer example programs that have been shifted from the main chapters in order to not disrupt the flow of reading.

Acknowledgments

My scientific work in the last two years, and the work on this thesis in particular, has been supported by a two-year habilitation fellowship of the *Deutsche Forschungsgemeinschaft* (DFG) in 1999 and 2000. From 1997 to 2000, the DFG also supported the SPARAMAT project, which is described in Chapter 3.

From 1995 to 2000, I worked in the group for programming languages and compiler con-

struction, led by Prof. Dr. Helmut Seidl, at the University of Trier. I thank Helmut Seidl for many constructive discussions and fruitful cooperation.

From 1995 to 1998 I was, as an assistant professor, fully involved in the teaching activities (lectures, seminars, exercises, labs) at the computer science department of the University of Trier. I also thank my colleagues and former colleagues in Trier for their friendship and a cooperative working environment. Particular thanks go to the group's secretary, Mrs. Brigitta Weiland.

A major component of my work originated from the SB-PRAM project at Prof. Dr. Wolfgang Paul's group of computer architecture and parallel computers at the computer science department of the University of Saarbrücken, where I did my diploma project in 1989–90 and my PhD project in 1991–94. I thank Wolfgang Paul and my former colleagues in his group for the constructive cooperation. In 1991–94 my PhD project was supported by a three-year PhD fellowship of the DFG in the Graduiertenkolleg Informatik at the University of Saarbrücken. In 1994 I was involved in the SB-PRAM project work and started work on the design and implementation of the PRAM programming language Fork, which I continued when I started to work in Trier in 1995.

Finally, I thank my wife Susanne, for her continuous love, support of my work, and for tolerating my commuting to Trier during the last six years. I also thank all my friends and my family, in particular my parents.

In the end of 2000, I will leave the University of Trier and join the University of Linköping, Sweden, as an associate professor in the computer science department. In this context, I see this monograph, beyond its role as habilitation thesis, as a final documentation of my scientific work of the last years, which was mostly done while I was affiliated with the University of Trier.

Trier, December 13, 2000

CHRISTOPH KESSLER

Contents

Chapter 1

Introduction and Overview

Today's computer architectures increasingly rely on parallel execution to keep pace with the demand for computational power. For instance, for further improvements in single-processor speed it becomes more and more critical to exploit instruction-level parallelism because the "automatic" improvements due to higher integration in chip technology will soon approach their physical limits. At the upper end of the performance scale, the supercomputer industry has recognized that self-designing sophisticated special purpose hardware is too time-consuming, while using off-the-shelf components reduces development costs and allows to participate in the advances of microprocessor performance. Hence, most modern supercomputers consist of hundreds or thousands of standard processors that are connected by an interconnection network to allow for synchronizing and exchanging data. In the middle of the spectrum, today's advanced workstations and servers usually have multiple processors as well, which are connected by a shared memory. Hence, the opportunities to speed up computations by exploiting parallelism become ubiquitous.

On the other hand, the problem of writing efficient code for parallel computers or compiling for such architectures is notoriously difficult, such that there emerged the so-called "parallel software crisis". Modern microprocessor technology has even decided to solve part of this problem in hardware by analyzing data and control dependencies at run time and dispatching instructions to pipelined and parallel functional units on-line, although this automatic reordering at run time is only applicable within the limited scope of a small window containing a few subsequent instructions in the code. Nevertheless, the compiler has a much larger insight into the program and can, by an off-line algorithm for suitably ordering the instructions in the generated machine program, relax the limited scope problem of the runtime scheduler, such that it can issue more instructions in the same cycle and use registers economically. Hence, the main task of exploiting parallelism remains with the programmer and the compiler.

Parallelism can be exploited by compilers at several different levels. In some cases, the task of determining opportunities for concurrent execution can be partially automatized.

For instance, when compiling for modern superscalar processors, the code generation phase of the compiler should be aware of the existence of multiple functional units when determining the order of instructions in the target program, such that independent instructions may be executed concurrently. This allows to exploit fine-grain parallelism at the instruction level.

Loops in the source program often exhibit a large potential for parallel execution. Furthermore their simple control structure allows for a condensed analysis of parallelizability. There exists a well-known theory of analyzing data dependences among the iterations of a loop nest, for transforming loop nests, for detecting parallelizable loops, and for scheduling their iterations for execution on different processors of a parallel computer. Nevertheless, the automatic parallelization technology for loops is bound to the sequential control structure of the source program. On the other hand, sophisticated parallel algorithms have been devised for various parallel architectures and for many problems. In general, these can not be derived from corresponding sequential algorithms and programs just by applying simple program transformations. For the user of a parallel computer, there remain thus two possibilities to exploit this rich potential of parallel algorithms.

The first approach is to make automatic parallelization more intelligent. Although the general problem of automatically understanding the meaning of a program is undecidable, it is, given sufficient application-domain-specific information, possible to identify computations on a local level and replace them by a parallel algorithm with the same semantics. This allows to go beyond the sequential program's control structure, but assumes a certain homogeneity of the application domain, which is necessary to avoid a combinatorial explosion of the different program concepts to be recognized and replaced. This condition is fulfilled e.g. in the domain of numerical matrix computations, while it is hardly met in the area of nonnumerical or highly irregular computations.

The second approach imposes the task of specifying the parallelism to be exploited on the programmer—albeit at a higher level than that offered by the default programming interface to the parallel hardware. For this purpose, explicitly parallel programming languages are required. Such a language should offer more means for expressing parallelism than just a parallel loop construct, because there are, especially in nonnumerical and irregular problems, many flavours of parallelism that cannot be expressed easily using parallel loops. Nevertheless, the compiler is still responsible for efficiently mapping the parallel activities specified by the programmer to the parallel target machine by exploiting the features of the parallel architecture.

In the last years I have done research in all of these facets of compiling for parallel architectures. The main part of my scientific work can thus be subdivided into three main subjects:

1. *Instruction scheduling* for RISC, superscalar, and vector processors, with the goal of minimizing the execution time and / or the number of registers used.

2. *Automatic Program Comprehension*: Identifying computation and data structure concepts in numerical codes (idiom recognition, tree pattern matching) with the goal of replacing them by suitable parallel equivalents.

3. *Design, Implementation, and Application of Parallel Programming Languages*, providing practical programming environments for the—up to now—mainly theoretically-based machine models PRAM, Asynchronous PRAM, and BSP.

The remainder of this chapter gives an informal introduction to these issues and provides a summary of my own contributions to the topics discussed in this thesis. A more formal and detailed presentation of selected work follows in the subsequent chapters of this book.

In the following, citations like [J6] (refereed journal paper), [C18] (refereed conference paper), [I7] (invited conference / workshop paper), [D1] (thesis) and [M11] (other publication like technical reports) refer to the list of my own publications at the end of this book, starting with page 315. All other citations refer to the bibliography.

1.1 Instruction Scheduling

Since 1989, when I started with research for my diploma thesis [D1], I worked on problems in code generation for RISC, superscalar, and vector processors.

Instruction scheduling is one of the core problems to be solved in the code generation phase of a compiler The goal is to minimize the number of registers used and / or the number of machine cycles taken by the execution of the schedule on the target machine. Usually, the instructions are not generic any more, i.e. they are already specific to the target processor, but they still operate on symbolic registers.

We consider the problem of local instruction scheduling, that is, the scope of optimization is a basic block[1] The data dependences of a basic block form a directed acyclic graph (DAG), where the nodes represent the instructions and the values computed by them, and the edges imply precedence relations among the instructions. Most edges are due to value flow dependences, that is, the value generated by the source instruction is used as an operand by the target instruction. A *(list) schedule* of the basic block resp. DAG is a total order of the instructions that does not conflict with the partial order given by the DAG edges. By adding the time and space constraints of the target processor (e.g., delayed instructions or a limited number of available registers), the list schedule usually determines a fixed, unique execution scenario for the instructions of the DAG. We will discuss some exceptions later.

A *register allocation* for a given (list) schedule is a mapping of the DAG nodes to the available processor registers, such that each value computed by an instruction into a register remains in that register and is not overwritten till its last use by a parent instruction. The total number of registers used is the *register need* of the allocation. A register allocation is called *optimal* for the given schedule if there is no other register allocation for that schedule that needs fewer registers. An optimal register allocation can be computed in a greedy manner in linear time.

A *time slot allocation* (or *time schedule*) for a given (list) schedule is a mapping of the instructions to time slots 1,2,... that indicate the point in time where the execution of the corresponding instruction will start on the target processor. An instruction must not start unless all its direct predecessor instructions in the DAG have completed their computation and have written their result values into registers. The *execution time* of a schedule is the time slot where all instructions in that schedule have completed their execution. A time slot allocation is *optimal* for the given list schedule if there is no other time slot allocation that has a smaller execution time. An optimal time slot allocation for the given list schedule can be computed in a greedy manner in linear time, even for superscalar processors with out-of-order execution if the runtime scheduling strategy of the target processor's dispatcher is known at compile time.

[1]A basic block is a sequence of instructions that contains no branch and no join of control flow.

A list schedule of a DAG is called *space-optimal* if there is no other list schedule for the same DAG that yields an optimal register allocation needing less registers than this schedule.

A list schedule of a DAG is called *time-optimal* if there is no other list schedule for the same DAG that yields an optimal time slot allocation needing less machine cycles than this schedule.

If the DAG has a special structure and the target processor meets certain conditions, several scheduling problems can be solved in polynomial time. For instance, if the DAG is a *tree*, a space-optimal list schedule for a single-issue RISC processor can be computed in linear time by the labeling algorithm of Sethi and Ullman [SU70], which is summarized in Section 2.3.1. For a general DAG the problem of computing a space-optimal schedule is NP-complete [Set75]. In the same way, computing a time-optimal schedule for a DAG and a processor with delayed instructions or parallel functional units is NP-complete. A thorough overview of the relevant literature on instruction scheduling will be given in Chapter 2, especially in Section 2.7.

When I started working on computing space-optimal schedules, I first developed a randomized linear-time heuristic for computing a schedule with low register need [C1], which is described in Section 2.3.2. By experiments with many randomly generated DAGs I showed that, compared to a single randomly generated schedule, the register need can be reduced by 30% on the average by this heuristic. However, no guarantee is given how far away is the register need of the reported solution from the actual optimum. Moreover, no method was known at this time to compute the optimum at all.

The heuristic [C1] was implemented in a Vector–Pascal compiler [FOP$^+$92] for the vector processor SPARK2 [FOP$^+$92] built at the institute for parallel computer architecture of Prof. Dr. W.J. Paul at the University of Saarbrücken. The SPARK2 has a vector register file of 4096 words that can be partitioned by the programmer into vector registers of arbitrary size, which are addressed by pointer arithmetics. For vector basic blocks derived from Vector-Pascal programs, I developed a method [C2] that determines, for the computed schedule, the optimal vector register length (which implies the number of available vector registers) and, if necessary, decides about spilling some register contents to the main memory, such that the total execution time on the SPARK2 is minimized.

As discussed above, computing space-optimal schedules for DAGs is hard. Trying to avoid the complexity of the general problem, I restricted the scope of optimization to the so-called *contiguous* schedules. A contiguous schedule is a schedule that can be obtained by a postorder traversal of the DAG. Hence, the number of contiguous schedules for a DAG with n nodes is bound by $O(2^n)$, while the number of (all) schedules is bound by $O(n!)$. Together with Prof. Dr. Thomas Rauber I developed an algorithm [C3,C7,J1] that computes space-optimal contiguous schedules for DAGs. The algorithm (see Section 2.3.2) enumerates the search space of different postorder traversals of the DAG and applies sophisticated pruning techniques that avoid the generation of duplicate or symmetric schedules and exploit tree structures of the remaining DAG nodes not yet traversed, which are handled by a modification of the Sethi–Ullman labeling algorithm. On the average, this algorithm reduces the complexity of the problem to about $O(2^{n/2})$. In practice, this is good enough to handle basic blocks with up to 80 instructions within acceptable time. Note that nearly all basic blocks that occur in real programs are much smaller.

The restriction to contiguous schedules means nevertheless that the described algorithm could find only suboptimal solutions, namely if no contiguous schedule is an optimal schedule. In order to be able to determine an optimal solution, I developed an algorithm [C12,J4] that solves the problem of determining a space-optimal schedule. As described in Section 2.4, the algorithm traverses the solution space by generating (conceptually) all topological orderings of the nodes of the DAG bottom-up by a dynamic-programming method. The algorithm exploits the fact that prefixes of schedules that contain the same subset of DAG nodes can be handled uniformly with respect to time and register constraints, and hence can be summarized as a single subsolution for which only one optimal subschedule needs to be stored. By classifying the subsolutions according to schedule length, register need, and execution time, the space of subsolutions can be traversed in a way that is most suitable for the intended primary optimization goal. The user can thus select either space or time optimization as the primary and the other one as the secondary optimization direction. It is also possible to specify a combined optimization criterion. In this way, the subspace implied by the most promising subsolutions can be explored first, and hence the algorithm defers the combinatorial explosion of the enumeration process to a later point of time that is, ideally, never reached because an optimum solution has been found meanwhile. Furthermore, the algorithm allows to exploit massive parallelism. The implementation showed that the algorithm is practical for basic blocks with up to 40 instructions, a class that contains nearly all basic blocks occurring in application programs. Hence, this method could be used in highly optimizing compilers. It should be noted that, for correctness only, any schedule is acceptable for code generation, and hence these expensive optimizations need only be applied in the final, optimizing compilation of the application.

Beyond describing these previously published results, Chapter 2 contains some by now unpublished material. For a faster optimization of large basic blocks I present and evaluate two heuristics. Then, I discuss how the performance of the dynamic programming method for time and time-space optimizations can be further improved by the novel concept of time-space profiles. Moreover, I discuss the extensibility of my enumeration algorithm for situations where spilling of some register contents to main memory or recomputations of some DAG nodes are permitted. I conclude with an extensive survey of related work.

This work will be continued in the near future in a new research project that is funded by the Ceniit programme of the University of Linköping.

1.2 Automatic Program Comprehension and Automatic Parallelization

Program comprehension is the process of discovering abstract concepts in the source code, e.g. the identification of data structures and (sub-)algorithms used. In its generality, this problem is undecidable, but, given some knowledge of the program's application domain, automatic understanding of programs is possible at least on a local level. In this case, understanding becomes a recognition process: it can be sketched as matching a given source program against a set of known programming concepts.

A number of problems have to be dealt with when facing automated recognition of algorithmic concepts [Wil90]: The most important ones are *syntactic variation*, *algorithmic variation* (a concept can be implemented in many different ways), *delocalization* (the implementation of a concept may be spread throughout the code), and *overlapping implementations* (portions of a program may be part of more than one concept instance).

Automatic program comprehension systems, working without user interaction, are mainly used for two purposes: to support software maintenance [RW90, HN90, KNS92, KNE93], e.g. for the automatic documentation of code, and to support automatically parallelizing compilers. Several methods for both areas have been proposed within the last years, and some (mostly experimental) systems have been built. Here we focus on the second issue.

Automated program recognition can play an important role in enhancing the capabilities of existing parallelizing compilers [BS87, Bos88, Cal91, PP91, RF93, SW93, PP94, PE95], in particular for target machines with a distributed memory [J2]. For instance, replacement of recognized sequential code by suitable parallel algorithms represents an aggressive code transformation which is not bound to the sequential program's control structure. Moreover, the acquired knowledge enables automatic selection of sequences of optimizing transformations, supports automatic array alignment and distribution, and improves accuracy and speed of performance prediction.

The application domain considered mainly consists of numerical computations, in particular linear algebra and PDE codes. Domain analysis done in my PhD thesis [D2] for the area of regular numerical computations on vectors and dense matrices has shown that the size of the set of concepts typically occurring in such codes remains reasonably small. But also in non-numerical or irregular numerical fields, recognition of algorithmic concepts in the code can drive the selection of suitable parallelization strategies ([DI94], [C15]).

When starting to do research for my PhD project, I discovered that concept recognition techniques may be well suited for numerical codes operating on dense matrices, because these exhibit a certain homegeneity of data structures (vectors, matrices) and algorithms and programming idioms used in Fortran source codes. In fact, I found by analyzing a representative set of Fortran77 programs that a rather small set of 150 concepts with only a few recognition rules per concept is sufficient to cover large parts of these programs, in particular for the time-consuming inner loops that are important to be parallelized.

Next, I developed and implemented an algorithm similar to bottom-up tree pattern matching for the automatic identification of these concepts [D2,I1,I2,C4,C5,J2]. Recognized code parts, represented as a subtree in the treelike intermediate representation of the program, are replaced by a summary node (concept instance) that contains all information about the identified concept occurrence, i.e. the function computed by this code part, together with a list of program objects that occur as parameters of this function. Nevertheless the summary node abstracts from all the details *how* this function is computed in the source program. Finally, it is more or less straightforward to generate calls to parallel library functions from the summary nodes. The integration of the concept recognizer in a large automatic parallelization system for distributed memory target machines, called PARAMAT, is described in [D2,C6,J2]. A front end and the concept recognizer of PARAMAT have been implemented; an implementation of the overall PARAMAT system was not possible because I was assigned other work after finishing my PhD thesis, and a corresponding project proposal was not accepted by DFG. Nevertheless, a simple back end [I2] exists for the shared-memory parallel computer SB-PRAM built at the institute of parallel computer architecture of the University of Saarbrücken.

There is only few related work on automatic program comprehension for automatic parallelization that has actually been implemented. One promising approach, although more targeted to interactive parallelization instead of code replacement, has been presented by Prof. Dr. Beniamino di Martino [DI94]. In a joint work [C9,M9,J6] we analyzed the fundamental differences and similarities of our approaches and proposed a method to combine them.

For a project proposal in 1996 I started to investigate the applicability of the PARAMAT approach to computations on sparse matrices. Sparse matrix codes lead to irregular numerical programs that are hard to parallelize automatically by today's parallelizing compiler technology, because part of the data dependency information required for parallelization is not known until run time. In a feasibility study [I5,C15] I discussed the potential and the challenges of a *speculative concept recognition* where a part of the identification of a concept may be deferred to run time tests. The placement of the run time tests can be optimized by a data flow framework. Out of these ideas grew the SPARAMAT project at the University of Trier, which was funded by the DFG from 1997 to 2000. SPARAMAT defines a special concept specification language CSL [C17,M11]. A generator parses a set of CSL specifications and constructs from these a hierarchically working concept recognizer (a C++ program), which is then able to process sequential source programs (Fortran77 in our implementation).

The foundations and the implementation of the SPARAMAT system are described in Section 3.

1.3 Design and Implementation of Parallel Programming Languages

The PRAM[2] is a popular programming model in the theory of parallel algorithms. It denotes a massively parallel synchronous MIMD[3] computer with a sequentially consistent shared memory and uniform memory access time. Although widely regarded as being highly unrealistic, a massively parallel realization of a Combine CRCW PRAM, the strongest PRAM variant[4] known in theory, has been designed and built by the parallel computer architecture group of Prof. Dr. Wolfgang Paul at the University of Saarbrücken, Germany. The SB-PRAM, theoretically based on Ranade's "Fluent machine" emulation approach [Ran87, RBJ88], combines several important concepts from parallel computer architecture, like multithreading and pipelining for latency hiding and cost-efficient use of silicon, a scalable combining intercon-

[2]PRAM = parallel random access machine, a straightforward extension of the sequential random access machine (RAM, also known as the von-Neumann architecture) by connecting multiple processors to a shared memory with machine-wide synchronous execution at the instruction level. The PRAM assumes unit memory access time for all processors and hence abstracts completely from the cost of shared memory access and data locality issues.

[3]MIMD = multiple instruction streams, multiple data streams. Program control is individual for each processor.

[4]Indeed, the strongest parallel machine model currently used is the BSR, which is even stronger than the Combine CRCW PRAM. Usually the BSR is not considered a PRAM variant but instead constitutes a different machine model.

nection network for resolving concurrent memory access conflicts and computing prefix and reduction operations on-the-fly in the network, simple RISC processors with a constant cycle time, and a common clock signal, which allows the whole machine to run synchronously at the machine instruction level and makes the computations (in principle) deterministic. The architecture is cost-effective and scalable. Unfortunately, no off-the-shelf components could be used; the prototypes are built with technology from 1991 when the design was fixed. Prototypes with (from the programmer's view) 31, 124, 498 and 1984 processors have been built; the largest prototype is currently (March 2000) still in the testing phase. The SB-PRAM group also wrote some system software and a software simulator that can be used for development and testing purposes. In short, the SB-PRAM is the only massively parallel shared memory MIMD machine in the world that is completely synchronous at the instruction level.

Together with the research on the architectural design of the SB-PRAM at the University of Saarbrücken, a programming language called FORK [HSS92] was proposed in 1989. The main goal of FORK was to exploit the synchronous execution feature of the SB-PRAM and make it transparent to the programmer at the operator level of the source language. A hierarchical processor group concept allows for the static and dynamic nesting of parallelism by splitting groups into subgroups, where the groups define the scope of sharing and of synchronous program execution. Group splitting can be arranged either explicitly by the programmer, or implicitly by the compiler where branches depending on nonshared values may compromise synchronous execution. In the latter case, the current group is split into one subgroup for each branch target, narrowing the scope of synchronous execution to the subgroups. With the group splitting feature, FORK enables the straightforward implementation of parallel divide-and-conquer algorithms.

But FORK was only a theoretical design and lacked nearly all features needed for practical work: pointers, data structures, dynamic arrays, function variables, floatingpoint numbers, input and output. There was no means (beyond group splitting) to escape from synchronous execution, and the language syntax was not compatible with any sequential programming language used in the real world. These restrictions in FORK seemed desirable to enable sophisticated program analyses [Sei93] and formal correctness proofs [Sch91, Sch92, RS92]. For the practical use with the SB-PRAM however, FORK was completely unusable. A compiler for FORK was started [Lil93] in the compiler construction group at the Computer Science department of the University of Saarbrücken but was finally abandoned in early 1994. In the meanwhile the SB-PRAM group at Saarbrücken implemented on their own an asynchronous C dialect as a straightforward extension of the gcc compiler and ported the P4 library to the SB-PRAM. pgcc has been used by the SB-PRAM group to implement the operating system PRAMOS and several large-scale applications from the SPLASH benchmark suite.

Immediately after finishing my PhD thesis in spring 1994 I was assigned the task to write a (new) compiler for FORK. Together with Prof. Dr. Helmut Seidl (now at FB IV Informatik, University of Trier, Germany) I completely redesigned [M6] the FORK language; the synchronous execution controlled by the processor groups was complemented by a second, asynchronous mode of program execution. We decided in favor of ANSI-C as the new sequential basis language [5]. In this way, the new language dialect, called Fork95 [I3,C8,J3], became a

[5]This was because C was the de-facto standard in 1994. A generalization of Fork to C++ as basis language is more or less straightforward, as its parallelism features are orthogonal to the object features of C++. On the

proper superset of the asynchronous C variant `pgcc` [I6].

The implementation of the Fork95 compiler for the SB-PRAM [I3], which I did on my own, was partly based on an existing ANSI-C compiler [FH91a, FH91b, FH95]. However, the entire code generation phase, driver and the standard libraries had to be (re)written from scratch. A first runnable version of the compiler could be presented in 1995.

In the subsequent years, Fork95 was extended several times. An important new language construct is the `join` statement [I4,C14,B1] which provides a flexible way to collect asynchronously operating processors and make them synchronously execute a (PRAM) algorithm. A straightforward application of `join` is the parallel execution of—up to now, always sequentialized—critical sections, because the synchronous program execution guarantees the deterministic parallel access to shared program objects.

Currently, the Fork language has reached a stable state and is used not only by myself but also by several instructors at various European universities for teaching classes on PRAM algorithms and parallel programming. Fork is also used in research projects on parallel programming, e.g. for the design of skeleton programming [Col89] languages [Mar97] or for the implementation of the PAD library of PRAM algorithms and data structures by Dr. Jesper Träff [C10,C13,J5,B1]. I have written myself many Fork programs, e.g. a solver for linear inequality systems [C11] or the highly irregular force calculation phase in a N-body simulation [B1].

Unfortunately, the SB-PRAM project was technologically overtaken by the continuous performance improvement of modern microprocessors, such that performance figures—except for highly irregular or I/O-bounded applications—do no longer look impressive if compared with small-scale shared memory multiprocessor servers or even sequential PCs.

On the other hand, the efficient compilation of Fork [B1] relies on some unique hardware features of the SB-PRAM, like the instruction-level synchronous program execution or the nonsequentializing execution of atomic fetch&add instructions that are used in basic synchronization mechanisms in the Fork compiler. When compiling Fork for asynchronous parallel machines [B1], these features must be emulated in software by the compiler or the runtime system, which would lead to a dramatical loss of performance. Hence, we had, in the long range, to retarget our research on parallel programming language design and implementation to commercially available parallel machines, while using Fork only for teaching and experimenting with parallelism.

In 1996/97 I designed together with Prof. Dr. Helmut Seidl the control-synchronous parallel programming language ForkLight [M10,C16], which is based on the Asynchronous PRAM model [CZ89, Gib89]. ForkLight has many similarities to Fork, like a sequentially consistent shared memory, SPMD execution with support for nested parallelism by a hierarchical group concept, and the duality of a synchronous and an asynchronous execution mode. However, the synchronicity of program execution is relaxed to the boundaries of the basic blocks in the program. This so-called *control-synchronicity* allows for more efficient compilation on asynchronous shared memory machines like e.g. the Tera MTA. In a similar way as in Fork, ForkLight relates the degree of control synchronous program execution to the block

---

other hand, Java could not be chosen because the parallelism concept used in Fork is not compliant with the thread-based parallelism of Java. In [C18,J7] I have studied the combination of Fork-like SPMD parallelism with a thread-free subset of Java.

structure of the program and makes it thus transparent for the programmer. I have written an experimental compiler for ForkLight. ForkLight may also be regarded as a more flexible alternative to the recent shared memory programming standard Open-MP [Ope97].

While ForkLight, in the same way as Fork, offers sequential consistency of the shared memory, this is not efficiently supported on many parallel platforms (e.g. some virtual shared memory systems) and must be emulated by the run-time system or the compiler. In many cases, though, sequential memory consistency is not really necessary at every point of the program. Many virtual shared memory systems offer low-level constructs for controlling the consistency of shared program objects individually. Unfortunately, this results in a low-level programming style that is as tedious and error-prone as message passing.

Valiant proposed the BSP[6] model [Val90] as a more realistic alternative to the PRAM model, by assuming a distributed-memory architecture and taking also the communication and synchronization costs into account when analyzing the complexity of a parallel program. The BSP model requires the programmer to structure a SPMD program as a sequence of so-called supersteps that are separated by global barriers. A superstep contains a phase of local computation followed by a global interprocessor communication phase specified by message passing primitives. This flat program organization is not well-suited for irregular applications. Some BSP library implementations [HMS$^+$98, BJvR98] support one-sided communication, which offers more comfort than two-sided message passing by automatizing receiving of messages and storing their data at the target site of a communication. Nevertheless, explicit message passing means low-level, error-prone programming.

In order to introduce a shared memory programming interface and to allow for nested parallelism in the BSP model, I developed the BSP programming language NestStep [C18,I8,J7]. Again exhibiting some similarities to Fork and ForkLight, NestStep uses the group concept to specify the scope of supersteps and of shared memory consistency. I have written an experimental implementation of the NestStep run time system based on Java; experience has shown that Java is unsuitable for this purpose because it exhibits poor performance. Hence, an implementation in C with (e.g.) MPI calls for the communication of updates to shared variables appeared to be preferable. Such a reimplementation was planned as a master student project since end of 1998. Due to a lack of master students in our group at the University of Trier, I finally decided early in 2000 to write the C/MPI variant myself. Early results show an improvement in the execution time of a factor of 5 (for sequential arithmetics-dominated programs) up to a factor of more than 20 for communications-dominated programs with up to 5 processors (our local Linux PC cluster) with respect to the corresponding Java versions. In the course of this implementation work for the C-based NestStep version NestStep-C, I reworked and simplified the concept of distributed arrays and abolished the support of volatile shared variables in order to obtain a unified, BSP-compliant memory consistency schema. Section 4.5 presents the new NestStep standard and thus partially contains previously unpublished material.

Together with Prof. Dr. Jörg Keller and Dr. Jesper Träff I have written a textbook [B1] with the (somewhat provocative) title *Practical PRAM Programming*, which appears in late 2000 at Wiley (New York). This book deals with the theory, emulation, implementation, programming, and application of synchronous MIMD computers with shared memory. It introduces

[6]BSP = bulk-synchronous parallel programming.

the PRAM model and basic PRAM theory, discusses PRAM emulation approaches, gives a complete reference to the SB-PRAM system architecture, and describes the Fork language design and implementation. It explains how to write structured parallel programs with Fork for many different parallel algorithmic paradigms and programming techniques, surveys the implementation of fundamental PRAM algorithms in the PAD library, and concludes with a parallel application written in Fork from the field of visualization and computational geometry. Part of the material that I contributed to *Practical PRAM Programming*, in particular the description of the language Fork, the concept of skeleton-style programming in Fork, and the implementation aspects of Fork, has been used and adapted in the Chapters 4 and 5 of this thesis.

Chapter 2

Instruction Scheduling for RISC, Superscalar, and Vector Processors

2.1 Introduction

The front end of a compiler usually generates an intermediate representation (IR) of the source program that is more or less independent of the source language as well as of the target machine. This intermediate representation could be regarded as a program for a virtual machine. It is the job of the compiler's back end to generate efficient target code from the intermediate representation.

The most important optimization problems in the code generation phase of a compiler are instruction selection, register allocation, and instruction scheduling. Although these three problems have complex dependence relations among each other, and thus should be solved simultaneously, it is the current state of the art in code generators that instruction selection is performed first, while there is no general agreement on the best order and degree of integration between register allocation and instruction scheduling.

In this work we focus on approaches to the integration of register allocation and instruction scheduling for basic blocks.

2.1.1 Processor Model

Our work is independent of any particular target processor. Rather, we characterize the run-time behaviour of the class of processors where our work may be immediately applied by a tuple

$$\mathcal{P} = (k, N_U, N_I, U, \Delta)$$

where k denotes the maximum number of instructions that may start execution in the same clock cycle, N_U specifies the number of (parallel) functional units, N_I is the number of different types of instructions, $U : \{1, ..., N_I\} \to \{1, ..., N_U\}$ specifies for each type of instruction on which unit it is to be executed, and $\Delta : \{1, ..., N_I\} \to \mathbb{N}_0$ gives the number of delay cycles for each type of instruction. The type of an instruction v in the intermediate representation of a program is given by *type*$(v) \in \{1, ..., N_I\}$.

A processor with $k = 1$ (e.g. the DLX architecture in [HP96]) is called a ***single-issue processor***, a processor with $k > 1$ is called a ***multi-issue processor***. A single-issue processor with $\Delta(i) = 0$ for all $i \in \{1, ..., N_I\}$ is called a ***simple (RISC) processor***; a single-issue processor with $\Delta(i) \geq 0$ for all $i \in \{1, ..., N_I\}$ and $\Delta(i) > 0$ for at least one instruction type i is called a ***pipelined (RISC) processor***. Multi-issue processors can be VLIW (very long instruction word), EPIC (explicitly parallel instruction code), or superscalar processors. In a ***VLIW processor*** the compiler determines concretely which IR instructions will execute at runtime in the same clock cycle on different units, and composes these to a single, wide instruction word (***code compaction***). The assembler-language interface to a VLIW processor thus consists of specialized, wide instructions as atomic units, where different parts of a wide instruction word address different functional units (*horizontal instruction set architecture, horizontal code*). In contrast, for a ***superscalar*** processor the instructions remain uncompacted, and the decoding and assignment of the instructions to the functional units is done online by a runtime scheduler (the ***instruction dispatcher***), which usually has a lookahead of a few instructions. Generally, a k-***issue superscalar processor*** is able to dispatch up to k instructions in the same clock cycle. Obviously, this latter variant imposes less work on the compiler and the resulting machine code is more portable than with VLIW architectures. On the other hand, the dispatcher adds time overhead and complexity to the target processor. The more recent EPIC approach [SR00] is a compromise that should combine the advantages of both approaches, namely the exploitation and transfer of compile-time information within the code as in VLIW architectures and the flexibility of superscalar processors.

Note that our characterization of the runtime behaviour of processors contains the following simplifying assumptions:

- For the first, we assume for simplicity that $k = N_U$. This property holds usually for VLIW architectures. For a superscalar processor it means that the dispatcher is not the performance bottleneck. In most single-issue and superscalar processors, however, the maximum degree of instruction-level parallelism is limited by the dispatcher's capacity k. For instance, the Motorola 88110 has 10 parallel functional units but can only issue up to 2 instructions per clock cycle.

- There is only one instance of each type of functional unit. Some superscalar processors provide multiple instances of certain types of units, e.g. multiple units for integer calculcations. For instance, the Motorola 88110 has two integer units and two graphics units.

- Each instruction can be executed on exactly one of the functional units. In some superscalar processors there may be certain types of instructions that may execute on different types of units (possibly with different time behaviour); for instance, some floatingpoint unit may be able to perform also certain integer arithmetics.

- Only one functional unit is involved in the execution of an instruction. This is not necessarily the case on all architectures: For instance, on the IBM Power architecture, storing a floatingpoint number simultaneously needs the integer unit (for the address calculation) and the floatingpoint unit [Wan93].

- Execution of an instruction occupies the corresponding functional unit only for one clock cycle. Hence, even for delayed instructions, another instruction may be fed into the same unit in the following clock cycle.
- The functional units are independent of each other. In some architectures (e.g. Motorola 88100) it is possible that some stages of different functional units share components, which may lead to further delays and additional scheduling constraints [EK91].

Even for target processors where one of these constraints is not fulfilled, we are often able to work around easily by applying only a small modification to our framework. For instance, for a k-issue superscalar processor with $k < N_U$, we only need to add that constraint to the function that computes the run time of a given schedule. In the same way we can easily handle multiple instances of the same type of functional unit. However, we renounce the full generality in order to keep the presentation simple.

For a further introduction to the design of RISC, VLIW, and superscalar processors, we refer to [Ung95] and [HP96].

2.1.2 Instruction Selection

The instruction selection problem consists in covering all abstract operations of the intermediate representation of the program by sequences of concrete instructions of the target processor, such that the semantics is the same as that of the intermediate representation. If the target machine provides multiple addressing modes for an operation, or there are different sequences of operations resulting in the same behaviour, instruction selection should favour a covering with minimum total cost, in terms of execution time and register requirements. A good introduction to the instruction selection problem can be found e.g. in [FH95].

There are several integrative approaches to instruction selection for basic blocks where the precedence constraints among the IR operations form a tree[1]. The dynamic programming algorithm by Aho and Johnson [AJ76] for trees integrates the problems of instruction selection and register allocation for a zero-delay RISC processor with multiple addressing modes for arithmetic operations, such that these can either directly access operands stored in the memory or fetch their operands from registers.

As a generalization, approaches based on *tree pattern matching* [Kro75, HO82, Cha87] allow to handle more complex patterns that consist of more than one IR tree node. Such systems are based on rewrite rules defined by tree grammars and work with one or several passes over the IR tree. Examples are [GG78], Twig [AG85, AGT89], [PG88], [WW88, Mat90], burg [FHP92b], iburg [FHP92a, FH95], [FSW92]. Beyond instruction selection, systems based on tree pattern matching are also able to perform certain simple tree transformations like operator strength reduction or application of distributivity rules automatically, such as OPTRAN [LMW88], Trafola [HS93], puma [Gro92], TXL [CC93].

The least-cost instruction selection problem for DAGs is NP-complete [Pro98]. Ertl [Ert99] proposes a heuristic that splits the DAG into disjoint subtrees and applies tree pattern matching by burg to each subtree separately. For a given tree grammar (i.e., processor type) it can

[1]Unless otherwise stated, by *tree* we mean an *in-tree*, i.e. a directed acyclic graph where each node has at most one successor node.

be checked easily whether a situation can occur where the heuristic may produce suboptimal results. Similar approaches to instruction selection for DAGs have been discussed by Boyland and Emmelmann [BE91] and Proebsting and Whaley [PW96]. The earlier approach by Davidson and Fraser [DF84] describes a heuristic for trees and DAGs which is not based on tree pattern matching but allows more flexible patterns (not just tree patterns).

2.1.3 Instruction Scheduling

By the instruction selection phase, we obtain a machine program that uses symbolic names (so-called *virtual registers*) for program objects which may be either stored in a register or reside in memory (or both). Although usually stored in a flattened representation, this program is merely a graph: the definitions and uses of the virtual registers imply only a partial order among the program's instructions, which must be preserved by the scheduler. The scheduler is responsible for deriving a total order of the instructions with respect to their time and space behaviour on the target processor (and, if necessary, a mapping of instructions to suitable functional units) such that all precedence constraints are preserved and furthermore a certain quality criterion is met, typically based on register need or execution time.

Depending on the target processor features and the scope of code optimization, we distinguish between several different scheduling problem groups and problem variations that are to be solved.

The group of *instruction scheduling problems for basic blocks* contains the following problem variations:

- **MRIS** = ***minimum register instruction scheduling problem***:

  Given a basic block whose precedence constraints are given by a directed acyclic graph, compute a schedule for a RISC processor with no delayed instructions such that the number of registers used is minimized.

  For the simplest case where instructions are not delayed (i.e., execution of each instruction takes unit time to complete and does not overlap) and partial recomputations are not admitted, MRIS is a classical NP-complete problem [Set75] (see also Table 2.1). Only if the DAG has some very simple structure, the problem can be solved in polynomial time. If the DAG is a *tree*, the labeling algorithm of Sethi and Ullman [SU70] yields an optimal solution; its run time is linear in the number of instructions; spilling of registers is not considered. If the DAG is *series-parallel*, a modification of the labeling algorithm by Güttler [Güt81, Güt82] computes an optimal solution in polynomial time. The general problem remains NP-complete even if considering only DAGs where nodes with multiple parents appear only as parents of leaves, and if the target processor has arbitrarily many registers available [AJU77], and if the target processor has only a single register [BS76].

  There are also some asymptotic results known for the minimum and maximum register need: Paul, Tarjan and Celoni proved that any DAG (with fixed indegree, here 2) of size n can be scheduled such that its register need is $O(n/\log n)$ [PTC77]. On the other hand, it is possible to construct DAGs of size n with a minimum register need of $\Omega(\sqrt{n})$ registers [Coo73], sharpened in [PTC77] to $\Omega(n/\log n)$, which reaches the upper bound. Paul and Tarjan [PT78] give, for a class of DAGs, an asymptotic lower and upper bound of $\Theta(\sqrt{n})$ registers and show that for the constructed class of DAGs, a growth in schedule length from linear to exponential time must be taken into account to achieve only a constant-factor improvement in the register need. As all these asymptotic results are based on contrived DAGs that are highly unlikely to occur in real-world programs, they are hardly helpful in practice, albeit providing fundamental insights for complexity theory.

- **MTIS** = ***minimum time instruction scheduling problem***:

  Given a basic block whose precedence constraints are given by a directed acyclic graph, compute a schedule for a given type of target processor with delayed instructions and / or with multiple functional units, such that the execution time of the schedule by the target processor is minimized, assuming an unlimited number of available registers.

  This problem is especially important for modern microprocessors which have sophisticated hardware features like pipelining and multiple functional units.

- **RCMTIS** = ***register-constrained MTIS problem***:

  Solve the MTIS problem under the additional constraint that only a fixed number of registers is available. If some values cannot be held in registers, spill code for loading and / or storing them must be generated and scheduled as well.

- The converse, a time-constrained MRIS problem, makes no sense in practice.

- **SMRTIS** = ***simultaneous minimization of register space and time in instruction scheduling***:

  Given a basic block whose precedence constraints are given by a directed acyclic graph, compute for a target processor with delayed instructions and / or multiple functional units a schedule that is optimal with respect to a user-specified combined criterion of register need and execution time, assuming an unlimited number of available registers.

  The MRIS, MTIS, and RCMTIS problems are thus special cases of the SMRTIS problem.

Instruction scheduling problems with a scope beyond basic blocks comprise techniques to enlarge basic blocks by loop unrolling [DJ79], trace scheduling region scheduling [GS90], by speculative or predicated execution of conditional branches [HHG$^+$95], by weighting cost estimations with execution frequencies [BSB96], by simple transformations to fill branch delay slots [GR90], or software pipelining of loops with and without loop unrolling.

Trace scheduling, described by Fisher [Fis81] and Ellis [Ell85], is a technique that virtually merges basic blocks on frequently executed control flow pathes to larger basic blocks (called traces), which allows to apply scheduling and optimizations at a larger scope for the time-critical parts of the program, while compensation code must be introduced in the less frequently executed parts of the program's control flow graph to reinstall correctness. In the worst case, the compensation code may cause the program to grow exponentially in length.

Percolation scheduling, introduced by Nicolau [Nic84] and later improved by Ebcioglu and Nicolau [EN89], considers individual instructions and repeatedly applies a set of local code transformations with the goal to move instructions “upwards” the program flow graph, that is, into the direction of program start. If necessary, compensation code must be inserted. The advantage of percolation scheduling over trace scheduling is that it usually produces less compensation code. Additional transformations may be integrated into the framework, for instance, rules that exploit associativiy or distributivity of arithmetic operators [NP90] in order to reduce the height of expression trees.

Region scheduling, proposed by Gupta and Soffa [GS90], is a heuristic technique that tries to avoid idle cycles caused by program regions with insufficient parallelism. Program regions consist of one or several blocks of statements that require the same control conditions, as defined by the program dependence graph (PDG) [FOW87]. Region scheduling repeatedly applies a set of local code transformations such as creating additional instructions by loop unrolling, moving instructions across basic block boundaries from regions with excess parallelism to regions with insufficient parallelism, and merging of regions, until all regions have sufficient parallelism. As a measure for the average degree of parallelism in a region, the approach uses the number of instructions in a region, divided by the length of the critical path of that region.

Software pipelining, introduced by Aiken and Nicolau [AN88] and Lam [Lam88], see also e.g. [ELM95, LVA95, CBS95], may reduce the execution time of a loop on a multi-issue processor by suitably overlapping the execution of subsequent iterations. The loop is transformed such that independent instructions from several consecutive iterations may be executed concurrently. Nevertheless, this technique tends to increase the length of operand live ranges and thus the register need. Various improvements of software pipelining have been proposed that are more sensitive with regard to the register pressure [NG93, WKE95, ANN95, BSC96].

But even for basic blocks, there was (prior to our work) no method that integrates register allocation and scheduling that is guaranteed to find an *optimal* solution for register allocation or both register need and completion time if the dependence graph is a DAG. In fact, nothing better than $O(n!)$ [C7,J1] was known, making the problem intractable for basic blocks of more than 15–20 instructions.

The two (not entirely orthogonal) optimization goals of minimizing register space (MRIS) and minimizing execution time (MTIS) often conflict with each other. During the 1990ies, MRIS was more or less subordinated to MTIS because, in order to exploit the deep pipelines of modern superpipelined and superscalar processors, time-conscious instruction scheduling was essential. Recently, MRIS regains importance because of two reasons [GZG99]: (a) Power consumption in embedded systems depends strongly on the number of memory accesses, and caches are usually quite small. (b) Superscalar processors with a runtime instruction dispatcher that is able to issue instructions out-of-order may, by hardware renaming, access internally more registers than are visible in the assembler language interface. Once a register contents has been spilled to memory by the compiler, the corresponding spill code must be executed, and the dispatcher cannot undo this decision at run time.

In the presence of delayed instructions or multiple functional units, the register requirements are either completely ignored for scheduling (e.g., [BG89]), or the DAG is assumed to

problem	MRIS	MTIS	MTIS	RCMTIS with spill code scheduling
dependencies	DAG	tree	DAG	DAG
machine model, delays	RISC no delays	Superscalar/ VLIW, 2 units no delays	pipelined RISC with different delays, $d > 1$	pipelined RISC delays ≤ 1
reference	NP-complete [Set75, BS76]	NP-complete [Li77, BRG89] [AJU77]	NP-complete [GJ79] [HG83] [LLM$^+$87] [PS90]	NP-complete [MPSR95]
remarks	linear for trees [SU70] s-p DAGs	$O(n \log n)$ for 1 arithmetic and 1 load/store unit	polynomial if $d \leq 1$	NP-hard also if all delays $= 0$ [Set75]

TABLE 2.1: NP-completeness results for some of the problems mentioned in this chapter. It should be noted that the global register allocation problem for a fixed program (usually solved by graph coloring) is NP-complete [GJ79], and that the least-cost instruction selection problem in DAGs (node cover) is also NP-complete [Pro98].

be a tree [BJR89, PF91, KPF95, VS95], or heuristic techniques are applied [HG83, GM86, Tie89, BJR89, BEH91, KPF95, MPSR95]. A summary of instruction scheduling methods for basic blocks is given in Section 2.7.

2.1.4 Register Allocation

If run before instruction scheduling, the ***register allocation problem*** consists of two subproblems: One subproblem is deciding, for each value in the program whether it is worth being stored in a register or whether it must reside in memory, e.g. by taking access frequencies into account. Moreover, the register allocator determines a mapping (the *register assignment*) of the values considered worth being held in a register to the machine registers, such that values that will be alive simultaneously in any schedule will be mapped to different registers. If two values are mapped to the same register, this implies sequentialization constraints for the scheduler.

If run after instruction scheduling, the lifetimes of the program values are fixed, and hence the register allocation can only be as good as the given schedule permits. It remains to compute a mapping of IR variables to machine registers such that simultaneously alive variables are mapped to different machine registers. If there are not enough machine registers available, the register allocator must decide about which values to spill to memory, and generate spill code and insert it correctly in the schedule.

For this second (and more common) variant, global register allocation by coloring a register interference graph [Ers71, Sch73, CAC$^+$81, Cha82] is still the state of the art and has been refined by several heuristic techniques [CH84, BGM$^+$89, BCKT89, CCK91, PF92, KH93, Pin93, AEBK94, NP94, BGS95, GA96, THS98, NP98]. There are precoloring strategies for basic blocks and for special control flow graph structures [KP95, JR96]. Spill code is automatically generated until the register need of the code does not exceed the number of physical registers. However, since the scheduling phase was not aware of the spill code introduced in the register allocation phase, the final schedule (with the spill code inserted a-posteriori) may be sub-optimal with respect to the RCMTIS problem, even if the scheduling phase had returned an optimal solution to the MTIS problem.

Hsu, Fischer, and Goodman [HFG89] propose a method to determine an optimal register allocation for a given schedule of a basic block such that the total spill cost is minimized. The given code needs not be in single-assignment form. The algorithm constructs a decision DAG of register status configurations, labels the edges with costs corresponding to load and store instructions implied by going from one configuration to the next one, and computes a minimum cost path through this DAG. Some pruning rules can be used to reduce the combinatorial explosion. Where the decision DAG would grow too large, a heuristic can be applied.

These pre-pass approaches to register allocation work on a given fixed schedule of the instructions. This typically results in a suboptimal usage of registers and increased register need. In [D1] we have shown that for large randomly generated basic blocks the register need could be reduced by approximately 30 percent on the average by suitably reordering the instructions within a range permitted by the data dependences among them.

Using as few registers as possible and minimizing the spill code is crucial for performance where registers are a scarce resource, because execution of load and store instructions takes, on most processors, more time than arithmetic operations on registers. This is especially important for vector processors which usually have a small number of vector registers (e.g., the CRAY vector computers have 8 vector register of 64×64 bit) or a register file that can be partitioned into a number of vector registers of a certain length (e.g., the vector acceleration units of the CM5 have register files of length 128×32 bit that can be partitioned into 1, 2, 4 or 8 vector registers, see [Thi92]). A vector operation is evaluated by splitting it into appropriate blocks and computing the blocks one after another. If the register file is partitioned into a small number of vector registers, each of them can hold more elements and the vector operations can be split into fewer blocks. This saves startup times and results in a faster computation. On the other hand, if such a small number of vector registers has to be enforced at the expense of holding some intermediate results not in vector registers but in the main memory, the overall execution time increases again due to the additional memory references. By balancing this trade-off, an optimal number of vector registers can be found [D1,C2].

Using fewer registers also pays off if less registers are to be saved and restored at procedure calls.

Basic blocks are typically rather small, up to 20 instructions. Nevertheless, scientific programs often contain also larger basic blocks, due to e.g. complex arithmetic expressions and array indexing. Larger basic blocks can also be produced by compiler techniques such as loop unrolling [DJ79] and trace scheduling [Fis81]. Therefore, it is important to derive register allocation techniques that cope with rather large basic blocks [GH88].

2.1.5 The Phase-Ordering Problem

As we can see from the previous description, there is a cyclic dependence between instruction scheduling and register allocation, known as the ***phase-ordering problem***.

If instruction scheduling is performed first (*pre-pass scheduling*), it is generally done with the primary goal of minimizing the execution time on the target processor. In any case, scheduling installs a total order among the instructions, which may be a disadvantageous one from the point of view of the subsequent register allocation phase, because, in particular in the presence of delayed instructions, time-conscious instruction scheduling tends to lengthen the lifetimes of values to be stored in registers and hence increases the register pressure, as more values are alive at the same time. If not enough registers are available, some register contents must be spilled and kept in main memory instead, resulting in additional load and store instructions that must again be scheduled, which may compromise the optimality of the schedule determined by the first phase.

On the other hand, if register allocation is performed first (*post-pass scheduling*), by assigning physical register names to the virtual registers, this allows to select good candidates for spilling (if necessary) and to generate the spill code before the subsequent scheduling phase, but it also introduces additional serialization constraints for the scheduling phase, because different virtual registers that are mapped to the same register must not be alive simultaneously. Hence, the instruction scheduler may have less opportunities for minimizing the execution time.

[BEH91] shows that separating register allocation and instruction scheduling produces inefficient code. Most instruction scheduling approaches (see the related work summary in Section 2.7) address just the MTIS problem (while trying to keep register pressure low if possible) and apply a-posteriori spilling of some values where the number of available registers is exceeded. Note that the problems of a-posteriori spilling just those virtual registers that lead to the minimization of some cost function (i.e., graph coloring) and generating optimal spill code [BGM+89, MD94] are NP-complete. Some approaches like [GH88] hence switch back and forth between the two phases, gradually refining the solution. Beyond our work presented in this chapter, [MPSR95] is one of the first approaches that considers both problems simultaneously.

2.1.6 Overview of the Chapter

First, we consider the problem of reordering the instructions of a basic block whose dependence graph is a DAG, in order to just minimize the register need (MRIS problem). We begin with heuristic methods based on so-called contiguous DAG schedules, a subset of the set of all possible schedules of a DAG. This includes an advanced enumeration algorithm that computes the optimal contiguous schedule and is practical for large DAGs up to 80 nodes.

Next, we present an advanced enumeration algorithm for computing an optimal solution to the MRIS problem based on dynamic programming that reduces the worst-case time bound for finding an optimal schedule from $O(n!)$ to $O(n^2 2^n)$. Tests for large basic blocks taken from real scientific programs and for suites of randomly generated basic blocks have shown that the algorithm makes the problem generally solvable for the first time also for medium-sized basic blocks with up to 40–50 instructions, a class that contains nearly all basic blocks encountered in real programs. The time complexity can be further reduced because we can exploit massive parallelism present in the algorithm.

We further show how this method can be adapted to solving the MTIS problem and the SMRTIS problem for target processors with delayed instructions and multiple functional units. As for such targets the two goals of scheduling, namely register space optimality and time optimality, typically conflict with each other, we extend our algorithm such that it can optimize either time or space consumption as the first priority, or one may specify a linear combination of time and space as the optimization goal.

Unfortunately, by these extensions for the SMRTIS problem, the algorithm is practical only for rather small basic blocks with up to 22 or 23 nodes. We complement the algorithm by a fast heuristic based on generating and testing random schedules, which can be used for larger basic blocks. We propose in Section 2.4.6 a generalization of the algorithm for time optimization such that more occasions for early pruning of suboptimal partial solutions can be exploited.

Another heuristic is based on pruning the solution space by splitting the basic block heuristically into multiple disjoint subblocks. The subblocks are scheduled separately by our enumeration method and the partial schedules are finally merged. It appears that this heuristic is not necessarily more successful than the naive random scheduling, and its run time and practicality is strongly dependent on the DAG structure and the sizes of the subDAGs.

Finally, we give an outlook of how to extend the main optimization algorithm to the optimal generation and scheduling of spill code where the number of available registers is exceeded, and shortly discuss the problems involved in allowing partial recomputation of some instructions to save register space.

We will conclude this chapter with a summary of related work on instruction scheduling.

2.2 Basic Definitions

2.2.1 DAGs and Schedules

We assume that we are generating code for a RISC-style single processor machine with general-purpose registers numbered consecutively (1, 2, 3, ...) and a countable sequence of memory locations. The arithmetic machine operations are three–address instructions like unary ($R_i \leftarrow \ominus R_j$) and binary ($R_i \leftarrow R_j \oplus R_k$) arithmetic operations on registers, loading a register R from a memory address a ($R \leftarrow$ `Load(`a`)`), and storing the contents of a register R to a memory address a (`Store(`a`)` $\leftarrow R$). For the first part of this chapter, we assume that each instruction takes one unit of time to execute, i.e. the result of an instruction can be used by the next instruction in the next time slot. The extension of our work for delayed instructions is discussed in Section 2.4.5.

Each input program can be partitioned into a number of basic blocks. On the machine instruction level, a *basic block* is a maximum-length sequence of three–address instructions that can only be entered via the first instruction and only be left via the last one. The data dependencies in a basic block can be described by a *directed acyclic graph (DAG)*. The nodes of the DAG represent as well the instructions of the basic block as the values computed by them. For simplicity we assume that all leaves of the DAG correspond to `Load` instructions, i.e. to values occurring as input to the basic block. A generalization of our algorithms to DAG leaves corresponding to constants or to values already residing in a register is straightforward. In general, DAG edges represent precedence constraints for the execution of instructions. For the sequel, we assume that DAG edges are caused by data flow, i.e. a DAG edge (u, v) connects instruction u to instruction v iff the value computed by u is used by v as an operand.

An example DAG is given in Figure 2.1. An algorithm to build the DAG for a given basic block can be found in [ASU86].

Definition 2.1 *For a DAG $G = (V, E)$, the* ***lower cone*** *(or just* ***cone*** *for short) of a node $v \in V$ in G, denoted by cone(G, v), is the subset $C \subset V$ such that for each $c \in C$ there is a path in G from c to v. The* ***cone DAG*** *of v, denoted by coneDAG(G, v), is the sub-DAG of G induced by its cone C, i.e. $(C, E \cap (C \times C))$. The* ***upper cone*** *of v in G, denoted by uppercone(G, v), is accordingly the subset $U \subset V$ such that for each $u \in U$ there is a path in G from v to u.*

Let $G = (V, E)$ be such a DAG with n nodes. A schedule of G is a total order of the nodes in V such that the partial order implied by the dependence arcs in E is preserved:

Definition 2.2 *A* ***(list) schedule*** *S of a DAG G is a bijective mapping of the nodes in V to the set of time slots$\{1, ..., n\}$ such that for all inner nodes $v \in V$ with children $v_1, ..., v_k$ holds $S(v) > S(v_i)$, $i = 1, ..., k$, i.e. v is computed only after all its children v_1,...,v_k have been computed.*

This implies that a schedule is *complete*, i.e. all nodes are scheduled at some time slot between 1 and n, and the schedule contains *no recomputations*, i.e., each node of the DAG is scheduled only once. Moreover, the schedule is *consistent* because the precedence constraints are preserved. Thus, a schedule is just a topological order of G.

The term ***list schedule*** for a total order of DAG nodes has been coined in [Cof76], where a list schedule is generated by a greedy topological sort traversal of the DAG. The name is due to the fact that the topological sorting algorithm maintains a set of nodes ready to be scheduled; if a total order among these is installed, e.g. by some heuristic priority function, these can be stored in a linear list. Hence, algorithms that compute a schedule by a topological sort traversal of the DAG are called ***list scheduling algorithms***.

In our implementations we represent a schedule S of length n as a list of nodes

$$[S^{-1}(1), S^{-1}(2), \ldots, S^{-1}(n)]$$

As the target processor considered by now has no delayed instructions and no multiple functional units, the (list) schedule is identical to the time schedule, and the execution time of any schedule is always equal to the number n of nodes.

2.2.2 Register Allocation for Basic Block Schedules

A *register allocation reg* for a given schedule maps the DAG nodes, in the order defined by that schedule, to a set of virtual machine registers such that the values computed by the DAG nodes are kept in registers from their definition up to their last use[2]:

Definition 2.3 *A mapping reg: $V \to \{1, 2, \ldots\}$ is called a* ***register allocation*** *for S, if for all nodes $u, v, w \in V$ the following holds: If u is a child of w, and $S(u) < S(v) < S(w)$, i.e. v is scheduled by S between u and w, then reg$(w) \neq$ reg$(u) \neq$ reg(v).*

[2]See also footnote 3 on page 24.

An *optimal register allocation* for a given schedule S uses the minimal number of registers, $m(S)$, that is still possible with that schedule:

Definition 2.4 *Let R_S denote the set of register allocations for a schedule S. Then,*

$$m(S) = \min_{\mathit{reg} \in R_S} \left\{ \max_v \mathit{reg}(v) \right\}$$

is called the ***register need*** *of the schedule S.*

An optimal register allocation for a given schedule can be computed in linear time, as described in Section 2.2.3.

A schedule is called *space-optimal* (or just *optimal* for short, as long as time constraints are not considered) if an optimal register allocation for this schedule uses not more registers than any other schedule for the DAG:

Definition 2.5 *A schedule S for a DAG G is called* ***space-optimal*** *if for all possible schedules S' of G holds $m(S') \geq m(S)$.*

In general, there will exist several space-optimal schedules for a given DAG. Recall that the problem of computing a space-optimal schedule for a DAG (the MRIS problem) is NP-complete [Set75].

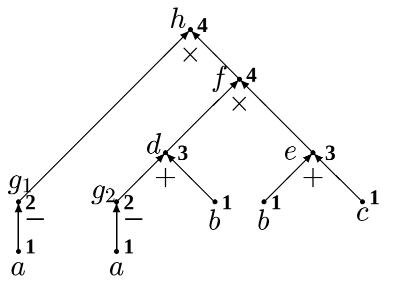
2.2.3 Optimal Register Allocation for a Given Schedule

If a schedule for the basic block is given, we can compute an optimal register allocation in linear time.

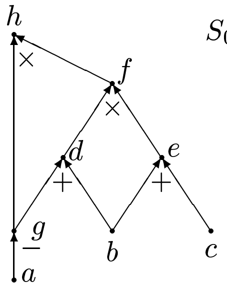
The technique of usage counts was first proposed in [Fre74]: Let *get_reg*$()$ be a function which returns an unoccupied register and marks it ‘occupied’. Let *free_reg*(r) be a function that marks register r ‘unoccupied’ again. *get_reg*$()$ is called each time a node is scheduled to hold the value computed by that node. *free_reg*$(\mathit{reg}(u))$ is called immediately before[3] the last parent of node u has been scheduled. To determine this event, we keep a usage counter for each node u which counts the number of u’s parent nodes already scheduled. Obviously, a register allocation for a given schedule can be computed with run time linear in the number of DAG edges, which is at most twice the number of instructions.

If one desires to keep register usage as *unequal* as possible (in order to obtain good candidates for a-posteriori spilling of registers) we propose the following method that keeps the list of unoccupied registers sorted by usage and hence requires an additional factor of $O(\log n)$ [C3,C7,J1]. Instead of *get_reg* we use a register allocation function called *first_free_reg* that allocates, for each node, the free register in this list with the smallest index. Since a new register is allocated only if there is no other free register left, the generated register allocation is optimal and the number of allocated registers is equal to the register need of the schedule.

[3]On most target architectures, the target register of an instruction may also be identical to one of the source registers (i.e., $\mathit{reg}(w) = \mathit{reg}(u)$ may be possible in Definition 2.3). Then, the register of u can be freed immediately *before* issuing the last parent of u. In the context of (strip-mined) vector instructions, however, the architecture may require registers to be mutually different [C2]. The latter variant was used in the example of Figure 2.1 and in Section 2.3.



a	$R_1 \leftarrow \texttt{Load}(a)$
g_2	$R_2 \leftarrow -R_1$
b	$R_1 \leftarrow \texttt{Load}(b)$
d	$R_3 \leftarrow R_2 + R_1$
b	$R_1 \leftarrow \texttt{Load}(b)$
c	$R_2 \leftarrow \texttt{Load}(c)$
e	$R_1 \leftarrow R_3 \times R_4$
f	$R_4 \leftarrow R_1 + R_2$
a	$R_2 \leftarrow \texttt{Load}(a)$
g_1	$R_3 \leftarrow -R_2$
h	$R_2 \leftarrow R_3 \times R_1$



S_0 :

a	$R_1 \leftarrow \texttt{Load}(a)$	c	$R_1 \leftarrow \texttt{Load}(c)$
g	$R_2 \leftarrow -R_1$	b	$R_2 \leftarrow \texttt{Load}(b)$
b	$R_1 \leftarrow \texttt{Load}(b)$	e	$R_3 \leftarrow R_1 + R_2$
d	$R_3 \leftarrow R_2 + R_1$	a	$R_1 \leftarrow \texttt{Load}(a)$
c	$R_4 \leftarrow \texttt{Load}(c)$	g	$R_4 \leftarrow -R_1$
e	$R_5 \leftarrow R_1 + R_4$	d	$R_1 \leftarrow R_4 + R_2$
f	$R_1 \leftarrow R_3 \times R_5$	f	$R_2 \leftarrow R_1 \times R_3$
h	$R_3 \leftarrow R_2 \times R_1$	h	$R_1 \leftarrow R_4 \times R_2$

FIGURE 2.1: Example: Suppose the expression tree for $(-a) \times ((-a+b) \times (b+c))$ has been scheduled using the Labeling algorithm of *Sethi/Ullman* with minimal register need 4. — Common subexpression elimination results in a DAG G, reducing the number of instructions to be executed. If G is scheduled according to S_0, now 5 registers are required. The better schedule on the right obtained by reordering the instructions needs only 4 registers, which is optimal.

This extended register allocation scheme uses a binary tree with the register 1,...,n as leaves. In each node, there is a flag *free* that indicates whether the subtree of this node contains a free register. In order to allocate a free register, *first_free_reg* walks along a path from the root to a free register by turning at each node to its leftmost child with a TRUE *free* flag. After switching the flag of the leaf found to FALSE, we traverse the path back to the root in order to update the flags. For each node on the path we set *free* to FALSE iff its two children have *free* $=$ FALSE. If a register is marked free again, *free_reg* must restore the *free* flags bottom–up on the path from this register back to the root in the same way by setting for each node *free* to TRUE if at least one child has a true *free* flag. The time for allocating or freeing a register is $O(\log n)$, hence the total time for computing the register allocation is $O(n \log n)$ for a DAG with n nodes.

The advantage of this allocation method is that the allocated registers usually differ more in their access rates since, in general, registers with a low index are used more often than registers with a high index. If the DAG nodes do not differ much in their outdegree, this results in an allocation scheme that is well suited for spilling registers: If we have fewer registers available in the target machine than the schedule requires, we would spill the registers with the largest indices [C2].

On the other hand, this simple strategy does not take the actual spill cost into account, which is proportional to the sum of the outdegrees of all DAG nodes mapped to a register.

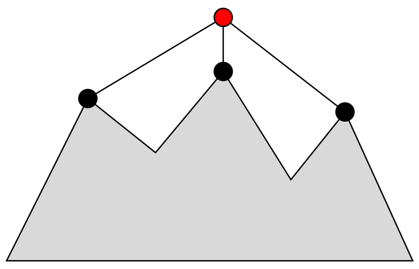


FIGURE 2.2: A DAG with $R > 1$ root nodes is made single-rooted by adding an artificial R-ary root node which corresponds to an empty R-ary operation.

Hence, the total spill cost may perhaps be improved by keeping the registers sorted in decreasing order of accumulated spill cost. But even with this strategy the register allocation computed may not be optimal with respect to the total spill cost, because the problem of determining a register allocation for a given schedule and a fixed number of registers with minimum total spill cost is NP-hard. Anyway, for pipelined or superscalar processors this problem is not really the interesting one, because the spill code must also be scheduled, which may change the original schedule. In Section 2.5 we will show on how to solve all aspects of the spilling problem optimally for the RCMTIS problem, such that the spill code is already taken into account in the scheduling phase.

2.3 Contiguous Schedules

In this section we assume for simplicity of presentation that the DAG $G = (V, E)$ representing the basic block under consideration has a single root node (with outdegree zero). Where this is not the case, the multiple root nodes are gathered by adding an artificial root node on top of the DAG as shown in Figure 2.2. In any schedule of the modified DAG the new root node will appear as the last instruction, as it depends on all other DAG nodes. Hence, it can be easily removed after the schedule has been computed.

Definition 2.6 *A schedule S of a DAG $G = (V, E)$ is called **contiguous**, if for each binary node $v \in V$ with children v_1 and v_2 the following restriction holds: If $S(v_1) < S(v_2)$), each $u \in$ cone$(G, v_2) -$ cone(G, v_1) (i.e., u is a DAG predecessor of v_2 but not of v_1), is scheduled after v_1, i.e. $S(v_1) < S(u)$. Vice versa, if $S(v_2) < S(v_1)$), each $u \in$ cone$(G, v_1) -$ cone(G, v_2) (i.e., u is a DAG predecessor of v_1 but not of v_2), is scheduled after v_2, i.e. $S(v_2) < S(u)$.*

A contiguous schedule has thus the property that for each binary node v, all nodes in the cone of one of v's children are scheduled first, before any other node belonging to the remaining cone of v is scheduled (see Figure 2.3). A generalization of this definition to k-ary nodes v is straightforward by determining the remaining cone DAGs of the k children of v according to a fixed linear order of the children. While general schedules can be generated by variants of topological-sort, contiguous schedules are generated by variants of *depth-first search*.

Obviously not all schedules of a (nontrivial) DAG are contiguous. As a consequence, as we shall see later, even a space-optimal contiguous schedule needs not be a space-optimal schedule.

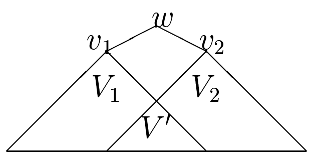


FIGURE 2.3: Example to the definition of a contiguous schedule. For any cone DAG G_v rooted at a binary node v, either all nodes in $V_1 = \textit{cone}(G_v, v_1)$ are scheduled before the remaining nodes in $V_2 - V_1$ are scheduled, or all nodes in $V_2 = \textit{cone}(G_v, v_2)$ are scheduled completely before the remaining nodes in $V_1 - V_2$ are scheduled.

First, let us consider the special case that the DAG is a tree.

2.3.1 Computing Space-Optimal Schedules for Trees

Definition 2.7 ***(tree node)***
(1) Each leaf is a tree node.
(2) An inner node is a tree node iff all its children are tree nodes and none of them has outdegree > 1.

Any tree node v has thus the property that its cone DAG $\textit{coneDAG}(G, v)$ is a tree. For tree nodes we compute labels recursively by a postorder tree traversal as follows:

Definition 2.8 ***(label** values for the nodes v of a tree)*
(1) For every leaf v, label$(v) = 1$.
(2) For unary v, label$(v) = \max\{\textit{label}(\textit{child}(v)), 1\}$.
(3) For binary v, label$(v) = \max\{2,\ \max\{\textit{label}(\textit{lchild}(v)), \textit{label}(\textit{rchild}(v))\} + q\}$
where $q = 1$ if label$(\textit{lchild}(v)) = \textit{label}(\textit{rchild}(v))$, and 0 otherwise.

The Labeling algorithm of *Sethi* and *Ullman* [SU70], see algorithm *treeSchedule* in Figure 2.4, generates a space-optimal schedule for a labeled tree by traversing, for each binary node, the subtree of the child node with the greater label value first. The algorithm exploits the fact that all registers used for the subtree evaluated first, except of that subtree's root node, can be reused for the nodes in the other subtree. One more register is thus only needed if the label values of both subtrees are identical. Hence, computing the labels by a bottom–up traversal of the tree and executing *treeSchedule* can be done in linear time.

The labeling algorithm can, with a minor modification to avoid duplicate evaluation of common subexpressions, be applied to DAGs as well, but this does not result necessarily in a space-optimal schedule. Indeed there are DAGs for which it produces suboptimal results [D1]. In any cases, *treeSchedule* can be used as a heuristic for DAGs [D1,C1].

2.3.2 Enumerating Contiguous Schedules for DAGs

Definition 2.9 *A **decision node** is a binary node which is not a tree node.*

Thus, all binary nodes that have at least one predecessor with more than one parent are decision nodes. In a tree, there are no decision nodes. For a general DAG let d be the number of decision nodes and b be the number of *binary* tree nodes. Then $k = b + d$ is the number

```
NodeList labelfs(node $v$)
 if $v$ is a leaf
   then return new NodeList($v$) fi
 if $v$ is a unary node
   then return labelfs(lchild($v$)) $\bowtie$ new NodeList($v$) fi
 if label(lchild($v$)) $>$ label(rchild($v$))
   then return labelfs(lchild($v$)) $\bowtie$ labelfs(rchild($v$)) $\bowtie$ new NodeList($v$)
   else return labelfs(rchild($v$)) $\bowtie$ labelfs(lchild($v$)) $\bowtie$ new NodeList($v$)
 fi
end labelfs;

Schedule treeSchedule( Tree $T = (V, E)$ )
 Schedule $S$ $\leftarrow$ new Schedule($|V|$);
 NodeList $L$ $\leftarrow$ labelfs( root($T$));
 for $i$ from 1 to $|V|$ do $S(i)$ $\leftarrow$ head($L$); $L$ $\leftarrow$ tail($L$) od
end treeSchedule
```

FIGURE 2.4: The labeling algorithm by Sethi and Ullman. When applied to a labeled tree T_v rooted at v, *labelfs* generates a node list that, if interpreted by *treeSchedule* as a schedule of T_v, uses exactly *label*(v) registers [SU70], which is optimal. The $\bowtie$ operator denotes list concatenation.

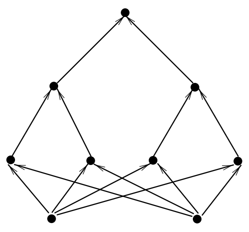


FIGURE 2.5: A DAG may have up to $n-2$ decision nodes. Here is an example for $n = 9$.

of binary nodes of the DAG. Note that a DAG may have up to $d = n - 2$ decision nodes, see Figure 2.5. In practice it appears that $0.5 \leq d/n \leq 0.75$ for most cases, see Tables 2.3 and 2.4.

The following properties of contiguous schedules are formally proven in [D1]:

Lemma 2.1 *For a tree T with one root and b binary nodes, there exist exactly 2^b different contiguous schedules.*

Lemma 2.2 *For a DAG with one root and k binary nodes, there exist at most 2^k different contiguous schedules.*

Lemma 2.3 *Let G be a DAG with d decision nodes and b binary tree nodes which form t (disjoint) subtrees $T_1, \ldots, T_t$. Let b_i be the number of binary tree nodes in T_i, $i = 1 \ldots t$, with $\sum_{i=1}^{t} b_i = b$. Then the following is true: If we fix an schedule S_i for T_i, then there remain at most 2^d different contiguous schedules for G.*

Corollary 2.4 *If we schedule all the tree nodes in a DAG G with d decision nodes by treeSchedule, there remain at most 2^d different contiguous schedules for G.*

If we throw at each decision node v a (0,1)-coin to determine which child of v to traverse first, and evaluate all the tree nodes in the DAG using *treeSchedule*, we obtain a random schedule. The ***randomized heuristic*** presented in [D1,C1] generates a user-specified number of such random schedules, plus the schedule that is implied by using *label* values (computed as above for the trees) as priorities. For each of these it determines the register need and returns the best schedule encountered.

If k random schedules are tested, the randomized heuristic takes time $O(k \cdot n \log n)$ for a DAG with n nodes. By choosing k suitably large, the probability that a bad schedule is returned can be made arbitrarily small, while the probability for encountering an optimal schedule increases accordingly.

In practice, this heuristic is probably sufficient. We have shown that for random DAGs, it reduces the register need by 30% on the average [C1], compared with a single random schedule of the DAG. The problem is, however, that in general one cannot know how far away from the optimum solution the reported schedule actually is.

Nevertheless, from our definitions above we can immediately derive an exhaustive search algorithm (see Figure 2.6) that enumerates all contiguous schedules of a DAG G by enumerating all 2^d bitvectors and uses a fixed contiguous schedule for the tree nodes of G.

While this algorithm still has exponential run time, for most practical cases the run time appears to be much smaller after clever pruning of the search space [C3,C7,J1]. The pruning strategy is based on the following observation: (consider the example DAG in Figure 2.7): Assume that the algorithm to generate a contiguous schedule decides to evaluate the left child f of the root h first (i.e., the decision bit of h is set to zero). Then node e appears in the schedule before g, since e is in the cone of f, but g is not. Therefore, there is no real decision necessary when node g is evaluated, because the child e of g is already evaluated. But because g is a decision node, the algorithm generates bitvectors containing 0s and 1s for the decision bit of g, although bitvectors that only differ in the decision bit for g describe the same schedule.

We say that g is *excluded* from the decision by setting the decision bit of h to 0, because the child e (and c) are already evaluated when the schedule of g starts. We call the decision bit of g *redundant* and mark it by an asterisk ($*$). [4]

The algorithm given in Figure 2.8 computes only those bitvectors that yield different schedules. We suppose again that tree nodes are evaluated by the labeling algorithm *labelfs*.

Table 2.2 shows the application of *descend* to the example DAG of Figure 2.7.

Definition 2.10 *Let N be the number of different contiguous schedules reported by the algorithm* descend. *We call $d_{\mathit{eff}} = \log N$ the* ***effective number of decision nodes*** *of G. Furthermore, let $d_\perp$ be defined as*

$$d_\perp = \min_{P \text{ path from some leaf to the root}} \#\text{decision nodes on } P$$

[4]Technically, this means that we use here "extended bitvectors" where each entry may have a value 0, 1, or $*$.

```
NodeList dfs( Node $v$, Bitvector $\beta$ )
 Let $v_1, \dots, v_d$ be the decision nodes of a DAG $G$, such that
 bitvector $\beta = (\beta_1, \dots, \beta_d) \in \{0,1\}^d$ represents the decisions made for them:
 if $v$ has already been visited  then return the empty list  fi
 mark $v$ as visited;
 if $v$ is a leaf  then return new NodeList($v$)  fi
 if $v$ is a tree node  then return labelfs($v$)  fi
 if $v$ is a unary node  then return  dfs(lchild($v, \beta$)) $\bowtie$ new NodeList($v$)  fi
 let $i$ be the index with $v = v_i$;    // $v$ is a decision node
 if  $\beta_i = 0$
   then return  dfs(lchild($v$)) $\bowtie$ dfs(rchild($v$)) $\bowtie$ new NodeList($v$)
   else return  dfs(rchild($v$)) $\bowtie$ dfs(lchild($v$)) $\bowtie$ new NodeList($v$)
 fi
end dfs

Schedule simple( DAG $G = (V, E)$ with root root)
 Schedule $S'$ = NULL;  $m'$ = $|V|$;
 forall $2^d$ different $\beta \in \{0,1\}^d$ do
   NodeList $L \leftarrow$ dfs($root, \beta$);
   let $S$ be the schedule corresponding to $L$;
   determine register need $m(S)$;
   if $m(S) < m'$ then $S' \leftarrow S$;  $m' = m(S)$ fi
 od
 return $S'$
end simple;
```

FIGURE 2.6: Algorithm *complete_search* determines a space-optimal contiguous schedule.

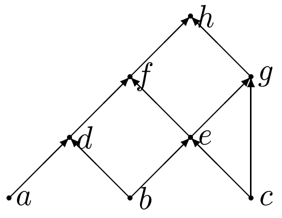


FIGURE 2.7: Example DAG.

For the example DAG of Figure 2.7, we have $N = 7$, hence $d_{\mathit{eff}} = \log 7$.
Obviously $d_{\mathit{eff}} \leq d$. Furthermore, we can show the following lower bound for d_{eff}:

Lemma 2.5 $d_{\perp} \leq d_{\mathit{eff}}$.

Proof: There must be at least as many bits set to 0 or 1 in each final bitvector as there are decision nodes on an arbitrary path from some leaf to the root, because no exclusion is possible on the path from the node being scheduled first to the root. The bitvector describing the path with the smallest number of decision nodes is enumerated by the algorithm, so the lower bound follows. □

This lower bound may be used to get a lower bound ($2^{d_{\perp}}$) for the number of schedules reported by *descend*. In the example above, the lower bound for d_{eff} is 2, since the path with

```
**procedure** *descend* ( **Bitvector** $\beta$, **int** $pos$ )
 **while** $\beta_{pos} = *$ **and** $pos < d$ **do** $pos \leftarrow pos + 1$ **od**
 **if** $pos \geq d$
 **then if** $\beta_{pos} = *$
   **then** *report*($\beta$)   // *new schedule found* //
   **else** // $\beta_{pos}$ *is empty:* //
     $\beta_d = 0$; *report*($\beta$);  // *new schedule found* //
     $\beta_d = 1$; *report*($\beta$);  // *new schedule found* //
   **fi**
 **else** // ($pos < d$:)
   $\beta_{pos} = 0$;
   mark exclusions of nodes $v_j$, $j \in \{pos + 1, ..., d\}$ by *lchild*($v_{pos}$) as $\beta_j \leftarrow *$;
   *descend*( $\beta$, $pos + 1$);
   $\beta_{pos} = 1$;
   mark exclusions of nodes $v_j$, $j \in \{pos + 1, ..., d\}$ by *rchild*($v_{pos}$) as $\beta_j \leftarrow *$;
   *descend*( $\beta$, $pos + 1$);
 **fi**
**end** *descend*;
```

FIGURE 2.8: The recursive function *descend* is the core of an algorithm that generates all *different* contiguous schedules of a DAG G. As before, $v_1, \ldots, v_d$ denote the decision nodes in reverse topological order (i.e., the root comes first). Function *descend* is called by *descend*(0^d, 1) where 0^d is a bitvector of size d initialized by zeroes. Function *report* computes the actual schedule from a bitvector β (in a similar way as *dfs* in Figure 2.6), compares it with the best schedule reported up to then, and stores the better one.

decision nodes $v_1, v_2, \ldots, v_5$:	h	f	g	d	e	
start at the root, $pos = 1$:	**0**		$*$			
propagate bits and asterisks, $pos = 2$:	0	**0**	$*$		$*$	
all bits set: first schedule reported	0	0	$*$	**0**	$*$	S_1
for $pos = 4$	0	0	$*$	**1**	$*$	S_2
'backtrack':	0	**1**	$*$	$*$		
	0	1	$*$	$*$	**0**	S_3
	0	1	$*$	$*$	**1**	S_4
'backtrack':	**1**	$*$		$*$		
	1	$*$	**0**	$*$		
	1	$*$	0	$*$	**0**	S_5
	1	$*$	0	$*$	**1**	S_6
'backtrack':	1	$*$	**1**	$*$	$*$	S_7

TABLE 2.2: For the example DAG of Figure 2.7, the algorithm *descend* executes the above schedule steps. Only 7 instead of $2^5 = 32$ contiguous schedules are generated.

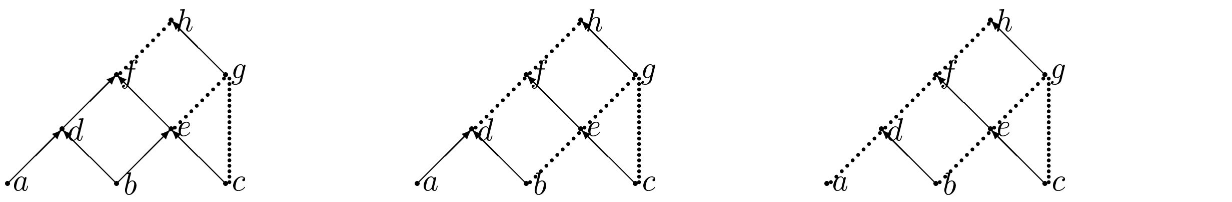


FIGURE 2.9: The example DAG is split in three steps by setting $\beta_1 = 0$, $\beta_2 = 0$, $\beta_4 = 0$. The edges between the generated subtrees are shown as dotted lines.

the least number of decision nodes is (c, g, h) which has two decision nodes.

2.3.3 Further Reducing the Number of Schedules

We now construct an algorithm that reduces the number of generated schedules further. The reduction is based on the following observation: Let v be a decision node with two children v_1 and v_2. Let $G(v) = (V(v), E(v)) =$ *coneDAG*(G, v) denote the cone DAG rooted at v, and $G(v_i) =$ *coneDAG*(G, v_i) the cone DAGs with root v_i, $i = 1, 2$. By deciding to evaluate v_1 before v_2, we decide to evaluate all nodes of $G(v_1)$ before the nodes in $G_{rest} = (V_{rest}, E_{rest})$ with $V_{rest} = V(v) - V(v_1), E_{rest} = E(v) \cap (V_{rest} \times V_{rest})$. Let $e = (u, w) \in E(v)$ be an edge with $u \in V(v_1), w \in V_{rest}$. The function *descend* marks w with a $*$. This can be considered as eliminating e: at decision node w, we do not have the choice to evaluate the child u first, because u has already been scheduled and will be held in a register until w is scheduled. Therefore, *descend* can be considered as splitting the DAG G into smaller subDAGs. We will see later that these subDAGs are trees after the splitting has been completed. The root of each of these trees is a decision node.[5] The trees are scheduled in reverse of the order in which they are generated. For the example DAG of Figure 2.7, there are 7 possible ways of carrying out the splitting. The splitting steps that correspond to schedule S_1 from Table 2.2 are shown in Figure 2.9.

If we look at the subDAGs that are generated during the splitting operation, we observe that even some of the intermediate subDAGs are trees which could be evaluated without a further splitting. E.g., after the second splitting step ($\beta_2 = 0$) in Figure 2.9, there is a subtree with nodes a, b, d which does not need to be split further, because an optimal contiguous schedule for the subtree can be found by a variant of *labelfs*. By stopping the splitting operations in these cases, the number of generated schedules can be reduced from 7 to 3 for the example DAG.

Depending on the structure of the DAG, the number of generated schedules may be reduced dramatically when splitting the DAG into trees. An example is given in Figure 2.10. In order to evaluate the generated trees we need a modified labeling algorithm that is able to cope with the fact that some nodes of the trees must be held in a register until the last reference from any other tree is resolved. Such an algorithm is given in Section 2.3.5. Before applying the new labeling algorithm, we explicitly split the DAG in subtrees $T_1 = (V_1, E_1), \ldots, T_k = (V_k, E_k)$. We suppose that these subtrees must be evaluated in this order. The splitting procedure is described in detail in the next section. After the splitting, we

[5]As we will see later, the root of the last generated tree is not a decision node.

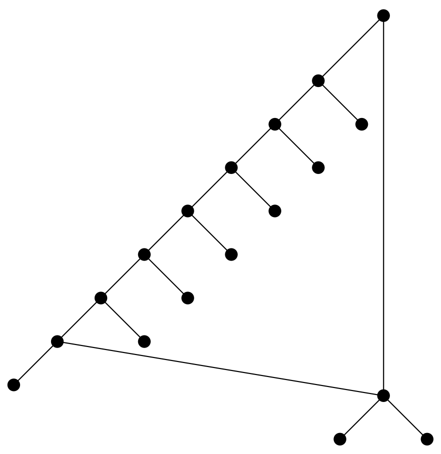


FIGURE 2.10: The DAG to the left has 8 decision nodes. When using the function *descend*, only one node gets an asterisk, i.e. 2^7 schedules are generated. When using the improved algorithm *descend2* presented in this section, only 2 schedules are generated: the first one evaluates the left child of the root first, the second one evaluates the right child first.

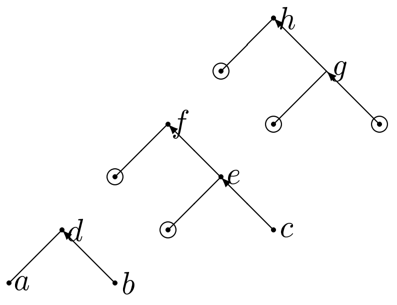


FIGURE 2.11: The example DAG is split into 3 subtrees by setting $\beta_1 = 0$, $\beta_2 = 0$, $\beta_4 = 0$. The newly introduced import nodes are marked with a circle. They are all non-permanent.

introduce additional import nodes which establish the communication between the trees. The resulting trees to the second DAG in Figure 2.9 are given in Figure 2.11.

We present the modified labeling algorithm in Section 2.3.5 with the notion of import and export nodes:

Definition 2.11 *An **export node** of a tree T_i is a node which has to be left in a register because another tree $T_j (j > i)$ has a reference to v, i.e., T_j has an import node which corresponds to v.*

*An **import node** of T_i is a leaf which is already in a register R because another tree $T_j (j < i)$ that has been evaluated earlier has left the corresponding export node in R.*

Therefore, an import node needs not to be loaded in a register and does not appear again in the schedule. For each import node, there exists a corresponding export node. Note that two import nodes $v_1 \neq v_2$ may have the same corresponding export node.

We distinguish between two types of import nodes:

- A ***permanent import node*** v can be evaluated without being loaded in a register. v can*not* be removed from the register after the parent of v is evaluated, because there is another import node of T_i or of another tree T_j that has the same corresponding export node as v and that has not been evaluated yet.

- A ***nonpermanent import*** node v can also be evaluated without being loaded into a register. But the register that contains v can be *freed* after the parent of v has been evaluated, because all other import nodes that have the same corresponding export node as v are already evaluated.[6]

Let the DAG nodes be $V = V_1 \cup \ldots \cup V_k$. We describe the import and export nodes by the characteristic functions *exp*, imp_p and imp_{np} that map V to $\{0,1\}$, and the function $corr$ that maps V to V. These are defined as follows:

$$exp(v) = \begin{cases} 1 & \text{if } v \text{ is an export node} \\ 0 & \text{otherwise} \end{cases}$$

$$imp_p(v) = \begin{cases} 1 & \text{if } v \text{ is a permanent import node} \\ 0 & \text{otherwise} \end{cases}$$

$$imp_{np}(v) = \begin{cases} 1 & \text{if } v \text{ is a nonpermanent import node} \\ 0 & \text{otherwise} \end{cases}$$

$$corr(v) = u, \text{ if } u \text{ is the corresponding export node to } v$$

The definition of import and export nodes implies

$$exp(v) + imp_p(v) + imp_{np}(v) \leq 1 \text{ for each } v \in V_i$$

2.3.4 Splitting the DAG into Subtrees

We now describe how the DAGs are split into subtrees and how the import and export nodes are determined. We derive a recursive procedure *descend2* that is a modification of *descend*. *descend2* generates a number of schedules for a given DAG G by splitting G into subtrees and evaluating the subtrees with a modified labeling scheme. Among the generated schedules are all optimal schedules. We first describe how the splitting is executed.

Let d be the number of decision nodes. The given DAG is split into at most d subtrees to generate an schedule. After each split operation, export nodes are determined and corresponding import nodes are introduced as follows: Let $v = v_{pos}$ be a decision node with children v_1 and v_2 and let $G(v), G(v_1)$ and G_{rest} be defined as in the previous section. We consider the case that v_1 is evaluated before v_2 ($\beta_{pos} = 0$). Let $u \in V(v_1)$ be a node for which an edge $(u,w) \in E(v)$ with $w \in V_{rest}$ exists. Then u is an export node in $G(v_1)$. A new import node u' is added to G_{rest} by setting $V_{rest} = V_{rest} \cup \{u'\}$ and $E_{rest} = E_{rest} \cup \{(u',w)\}$. u' is the corresponding import node to u. If u has already been marked in $G(v_1)$ as export node, then u' is a permanent import node, because there is another reference to u (from another tree) that is evaluated later. Otherwise, u' is a nonpermanent import node. If there are other edges $e_i = (u, w_i) \in E(v)$ with $i = 1, \ldots, k$ and $w_i \in V_{rest}$, then new edges $e_i' = (u', w_i)$ are added to E_{rest}. If $k \geq 1$, G_{rest} is not a tree and will be split later on.

One splitting step is executed by the function *split_dag* whose pseudocode is given in [J1]. We omit the details here because this method is only applicable to register space optimization and hence an in-depth description is beyond the scope of this book. It is sufficient to note that

[6]This partitioning of the import nodes is well defined, since the order of the T_i is fixed.

the worst-case time complexity of *split_DAG* is linear in the DAG size. *split_dag* is called by the recursive procedure *descend2* that visits the decision nodes in reverse topological order (in the same way as *descend*). For each decision node v (not yet marked by a $*$) with children v_1 and v_2, *descend2* executes the two possible split operations by two calls to *split_dag*, which build, for each of the two possible orderings of the children, the two (remaining) cone DAGs G_{left} and G_{right} (see Figure 2.12) and also compute the values of exp, imp_p, imp_{np} and $corr$ for each node in these cone DAGs for each variant. If one of these cone DAGs is a tree, all decision nodes in the tree are marked with a $*$ so that no further split is executed for these decision nodes. The root of the tree is added to *roots*, which is a set of nodes that is empty at the beginning. If all decision nodes have been marked, i.e. a new schedule has to be reported, the trees which have their roots in *roots* are evaluated according to *ord* with the modified labeling scheme *labelfs2* presented in Section 2.3.5.

The algorithm is started with a call *descend2*$(0^d, 1, G)$ where again 0^d is a zeroed bitvector with d positions, and the decision nodes $v_1, \ldots, v_d$ of the DAG G are supposed to be sorted in reversed topological order (the root first).

By fixing the traversal order of the trees in the *top_sort* call (before computing and reporting the schedule), we also determine the type of the import nodes[7], i.e., which import nodes return a free register when being scheduled. An import node is nonpermanent if it is the last reference to the corresponding export node. Otherwise it is permanent: The register cannot be freed until the last referencing import node is computed.

2.3.5 Evaluating Trees with Import and Export Nodes

By repeatedly splitting a DAG, *descend2* generates a sequence of trees $T_1 = (V_1, E_1)$, ..., $T_k = (V_k, E_k)$ with import and export nodes. It remains to describe how an optimal schedule is generated for these trees. With the definitions from Section 2.3.3 we define two functions *occ* and *freed*:

$$occ : V \to \{0,1\} \qquad with \qquad occ(v) = \sum_{w \text{ is a proper predecessor of } v} exp(w)$$

counts the number of export nodes in the subtree $T(v)$ with root v (excluding v), i.e. the number of registers that remain occupied after $T(v)$ has been evaluated.

$$\mathit{freed} : V \to \{0,1\} \qquad with \qquad \mathit{freed}(v) = \sum_{w \text{ is a proper predecessor of } v} imp_{np}(w)$$

counts the number of import nodes of the second type in $T(v)$, i.e. the number of registers that are freed after $T(v)$ has been evaluated.

We define for each node v of a tree T_i, $1 \leq i \leq k$, a label *label*(v) that specifies the number of registers required to evaluate v as follows:

[7]If two import nodes v_1 and v_2 of the same tree T_i have the same corresponding export node, then the type is determined according to the traversal order of T_i as described in the nect section. For the moment we suppose that both nodes are permanent; Appendix A.1 handles the other case.

```
procedure descend2 ( Bitvector $\beta$, int $pos$, DAG $G$ )
 while $\beta_{pos} = *$ and $pos \le d$ do $pos \leftarrow pos + 1$ od;
 if $pos = d + 1$
 then $ord \leftarrow top\_sort(roots)$;  $S \leftarrow$ new NodeList();
   for $i = 1$ to $d$ do $S \leftarrow S \bowtie labelfs2(ord(i))$ od; report(S);
 else
   $\beta_{pos} = 0$; $G_1$ = copy($G$);
   mark exclusions of nodes $v_j, j \in \{pos + 1, ..., d\}$ by $lchild(v_{pos})$ as $\beta_j = *$;
   $G_{left} \leftarrow coneDAG(G_1, lchild(v_{pos}))$;
   if $G_{left}$ is a tree
   then mark all decision nodes in $G_{left}$ with a $*$; $roots = roots \cup \{lchild(v_{pos})\}$ fi;
   $G_{right} \leftarrow split\_dag(v_{pos}, lchild(v_{pos}), rchild(v_{pos}), G_1)$;
   if $G_{right}$ is a tree then
     mark all decision nodes in $G_{right}$ with a $*$;  $roots = roots \cup \{v_{pos}\}$; fi;
   descend2( $\beta$, $pos + 1, G_1$);
   $\beta_{pos} = 1$; $G_2$ = copy($G$);
   mark exclusions of nodes $v_j, j \in \{pos + 1, ..., d\}$ by $rchild(v_{pos})$ as $\beta_j = *$;
   $G_{right} \leftarrow coneDAG(G_2, rchild(v_{pos}))$;
   if $G_{right}$ is a tree then
     mark all decision nodes in $G_{right}$ with a $*$; $roots = roots \cup \{rchild(v_{pos})\}$ fi;
   $G_{left} \leftarrow split\_dag(v_{pos}, rchild(v_{pos}), lchild(v_{pos}), G_2)$;
   if $G_{left}$ is a tree then
     mark all decision nodes in $G_{left}$ with a $*$; $roots = roots \cup \{v_{pos}\}$ fi;
   descend2( $\beta$, $pos + 1, G_2$);
 fi
end descend2;
```

FIGURE 2.12: The recursive *descend2* routine. *top_sort* is a function that sorts the nodes in its argument set in topological order according to the global DAG. Hence, if there are nodes v, v_1, v_2, w_1, w_2 where $v = v_{pos}$ is a decision node with $\beta_{pos} = 0$ and $(v_1, v), (v_2, v) \in E$ and w_1 is a predecessor of v_1 and w_2 is a predecessor of v_2, then $ord(w_1) < ord(w_2)$. Accordingly, if $\beta_{pos} = 1$, then $ord(w_2) < ord(w_1)$. *copy* is a function that yields a copy of the argument DAG.

If v is a leaf, then

$$label(v) = 2 - 2 \cdot (imp_p(v) + imp_{np}(v)).$$

For an inner node v with two children v_1 and v_2, let S_j denote the subtree with root $v_j, j = 1, 2$. Being restricted to contiguous schedules, we have two possibilities to evaluate v: If we evaluate S_1 before S_2, we use

$$m_1 = \max(label(v_1), label(v_2) + occ(v_1) + 1 - freed(v_1))$$

registers, provided that v_1 (v_2) can be evaluated with label(v_1) (label(v_2)) registers. After S_1 is evaluated, we need $occ(v_1)$ registers to hold the export nodes of S_1 and one register to hold v_1.

On the other hand, we free *freed*(v_1) registers when evaluating S_1. If we evaluate S_2 before S_1, we use

$$m_2 = \max(\mathit{label}(v_2), \mathit{label}(v_1) + \mathit{occ}(v_2) + 1 - \mathit{freed}(v_2))$$

registers. We suppose that the best traversal order is chosen and set

$$\mathit{label}(v) = \min(m_1, m_2)$$

With this label definition, the labeling algorithm introduced above, now called *labelfs2* [J1], can be reused for trees with import and export nodes.

Appendix A.1 proves that a call *labelfs2*(v) generates an optimal contiguous schedule of the subtree rooted at v that uses exactly *label*(v) registers. Hence, we conclude with the following theorem:

Theorem 2.6 *For a given DAG, the algorithm descend2 generates a contiguous schedule that uses no more registers than any other contiguous schedule.*

2.3.6 Experimental Results

We have implemented *descend* and *descend2* and have applied them to a great variety of randomly generated test DAGs with up to 150 nodes and to large DAGs taken from real application programs, see Tables 2.3 and 2.4. The random DAGs are generated by initializing a predefined number of nodes and by selecting a certain number of leaf nodes. Then, the children of inner nodes are selected randomly. The following observations can be made:

- *descend* considerably reduces the number of contiguous schedules that would be enumerated by the naive algorithm.
- *descend2* often leads to a large additional improvement over *descend*, especially for DAGs where *descend* is not so successful in reducing the number of different contiguous schedules.
- *descend2* works even better for DAGs from real application programs than for random DAGs.
- In almost all cases, the computational effort of *descend2* seems to be justified. This means that, *in practice*, an *optimal* contiguous schedule (and thus, contiguous register allocation) can be computed in acceptable time even for large DAGs.

n	d	N_{simple}	$N_{descend}$	$N_{descend2}$
24	12	4096	146	5
25	14	16384	1248	3
28	16	65536	748	22
27	17	131072	744	15
28	19	524288	630	32
33	21	2097152	1148	98
36	24	16777216	2677	312
38	26	67108864	6128	408
39	27	134217728	1280	358
42	29	536870912	6072	64
42	31	2^{31}	2454	152
46	34	2^{34}	4902	707
54	39	2^{39}	30456	592
56	43	2^{43}	21048	4421

n	d	N_{simple}	$N_{descend}$	$N_{descend2}$
20	14	16384	160	10
28	16	65536	784	8
29	18	262144	938	32
30	21	2097152	1040	64
37	23	8388608	13072	24
38	24	16777216	11924	56
45	27	134217728	100800	18
41	29	536870912	74016	364
41	31	2^{31}	3032	142
41	31	2^{31}	3128	180
44	33	2^{33}	40288	435
46	34	2^{34}	40244	1008
48	37	2^{37}	21488	1508
53	42	2^{42}	79872	3576

TABLE 2.3: Measurements excerpted from a test series for large random DAGs. The number of contiguous schedules generated by the algorithms *simple*, *descend* and *descend2* are given for typical examples.

Source	DAG	n	d	N_{simple}	$N_{descend}$	$N_{descend2}$	$T_{descend}$	$T_{descend2}$
LL 14	second loop	19	10	1024	432	18	0.1 sec.	$<$ 0.1 sec.
LL 20	inner loop	23	14	16384	992	6	0.2 sec.	$<$ 0.1 sec.
MDG	calc. $\cos(\theta), \sin(\theta), ...$	26	15	32768	192	96	$<$ 0.1 sec.	$<$ 0.1 sec.
MDG	calc. forces, first part	81	59	2^{59}	—	7168	—	13.6 sec.
	subDAG of this	65	45	2^{45}	—	532	—	0.9 sec.
	subDAG of this	52	35	2^{35}	284672	272	70.2 sec.	0.8 sec.
	subDAG of this	44	30	2^{30}	172032	72	42.9 sec.	0.3 sec.
	subDAG of this	24	12	4096	105	8	$<$ 0.1 sec.	$<$ 0.1 sec.
SPEC77	mult. FFT analysis	49	30	2^{30}	131072	32768	20.05 sec.	21.1 sec.

TABLE 2.4: Some measurements for DAGs taken from real programs (LL = Livermore Loop Kernels; MDG = Molecular Dynamics, and SPEC77 = atmospheric flow simulation, both from the Perfect Club Benchmark Suite). The table also gives the run times (1992) of the algorithms *descend* and *descend2*, implemented on a SUN SPARC station SLC. The tests show that for large DAGs *descend* is too slow, but the run times required by *descend2* remain really acceptable.

2.3.7 Summary of the Algorithms for Contiguous Schedules

We started with the simple randomized heuristic and the simple enumeration algorithm that evaluates only the tree nodes of the DAG by a labeling algorithm and generates 2^d contiguous schedules, where d is the number of decision nodes of the DAG. We presented two refined variants of the simple enumeration algorithm. The first variant is the exclusion of redundant decision nodes as performed by procedure *descend*. The second variant is the splitting of the DAG in subtrees (performed by *descend2*) and scheduling these by the modified labeling algorithm *labelfs2*. The experimental results confirm that this variant generates only a small number of contiguous schedules, even for large DAGs. Among the generated schedules are all schedules with the least register need (with respect to contiguous schedules). Therefore, by using *descend2* we find the optimal contiguous schedule in a reasonable time even for large DAGs. The dramatic reduction in schedules generated makes *descend2* suitable for the use in optimizing compilers, especially for time-critical regions of the source program.

2.3.8 Weaknesses of Contiguous Schedules

However, the algorithms for computing an optimal contiguous schedule may not find an optimal schedule because it might be noncontiguous. There are DAGs for which a general schedule exists that uses fewer registers than every contiguous schedule, e.g. the subDAG of MDG in Table 2.4 with 24 nodes[8] or the example DAG given in Figure 2.13.

Moreover, if the time behaviour of a schedule is to be considered in the presence of delayed instructions or multiple functional units, contiguous schedules tend to be inferior to noncontiguous ones, because at contiguous schedules it is much more typical that a parent node is scheduled directly after its child, which may cause a data hazard.

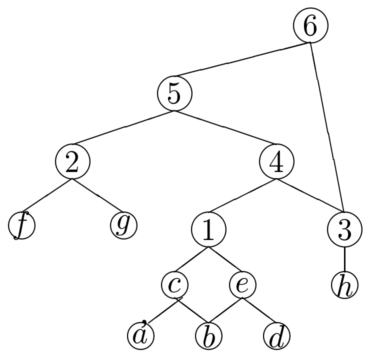


FIGURE 2.13: This DAG can be scheduled noncontiguously (in the order a, b, c, d, e, 1, f, g, 2, h, 3, 4, 5, 6) using 4 registers, less than each contiguous schedule: Any schedule using 4 registers must schedule the subDAG with root 1 first. A contiguous schedule can do this only by scheduling the left child of node 6, the right child of node 5 and the left child of node 4 first. In a contiguous schedule, nodes h and 3 must be scheduled after 1 and thus the values of nodes 3 and 4 must be held in two registers. But in order to schedule nodes f, g and 2, three more registers are required, thus the contiguous schedule uses at least five registers.

[8]For the subDAG of MDG with $n = 24$ nodes in Table 2.4, there is a noncontiguous schedule that uses 6 registers, while the computed contiguous schedule takes 7 registers.

2.4 Computing Optimal Schedules

Let N_G denote the number of different schedules S for a DAG G with n nodes. N_G obviously depends on the structure of the DAG. Clearly N_G is less than $n!$, the number of permutations of the instructions. Our hope is that this is a very coarse upper bound and that N_G is not too high for small and medium-sized DAGs. Then we could just generate *all* N_G possible schedules for a given DAG G and select an optimal one among them.

2.4.1 Enumerating Schedules

Let us begin with a naive approach using an algorithm for topological sorting to generate these schedules. ***Topological sorting*** performs n steps. Each step selects one of the nodes, say v, with indegree zero, schedules it, and deletes all edges from v to its parents. Note that some of v's parents may get indegree 0, thus they become selectable in the following step.

We use an array `S[1:n]` of nodes of G to store the current (partial) schedule S. Let z_0 be the set of all leaves of G, i.e. the nodes with initial indegree zero, and let *INDEG*$_0$ be an array containing the initial indegree of every node. The call *nce*$(z_0, \mathit{INDEG}_0, 1)$ of the recursive algorithm ***nce*** given in Figure 2.14 yields all possible schedules of G.

It is easy to see that the algorithm has a recursive nesting depth of n. For each recursive call, a copy of the current set z and of the current array *INDEG* must be generated, so the required space is $O(n^2)$. The iterations of the **for all** loop over z could be executed in parallel, if each processor has its own copy of S. If this loop is executed in sequential, copying can be avoided; thus space $O(n)$ will suffice. The run time of *nce* depends on the structure of the given DAG G and of the number of possibilities for the choice of the next node for S, i.e. the cardinality of z in each step of the algorithm.

The total run time of *nce* is proportional to the total number of generated schedules. Thus, the run time is bounded by $O(n!\cdot n)$, because for each call we either update the *INDEG* array *or* print a schedule.[9] The *real* number of schedules generated will be considerably smaller than the worst case bound $n!$ because only a small fraction of the permutations are valid schedules. Nevertheless, algorithm *nce* is impractical for DAGs with more than 15 nodes; our prototype implementation of *nce* already needs several days for DAGs of size around 20.

As a straightforward extension of *nce*, the algorithm could check after each scheduling decision whether the remaining DAG induced by the yet unscheduled nodes is a tree or forest [Sei00]. In that case, we need no longer follow the recursive computation of *nce* but can fall back to our refined tree scheduling method with *labelfs2* (see Section 2.3.5) and determine a space-optimal remainder of the schedule in time polynomial in the size of the tree or forest.

In order to improve the exhaustive search performed by *nce*, we must have a closer look at it. *nce* builds a decision tree, called the ***selection tree***, $T = (Z', H')$. The set Z' of nodes, called the *selection tree nodes*, contains all the *instances* z' of the zero-indegree sets of DAG nodes that occur as the first parameter z in a call to *nce* during the execution of the algorithm. A directed edge $h = ({z'}_1, {z'}_2) \in H'$, called *selection tree edge*, connects two selection tree nodes ${z'}_1, {z'}_2 \in Z'$ iff there is a step in *nce* that directly generates ${z'}_2$ from ${z'}_1$ by selecting

[9] We can easily eliminate a factor of n if the DAG has only one root r: This node r must always be the last node in a schedule, i.e. $S(r) = n$, terminating the recursion at nesting level $n-1$.

```
**Schedule** $S \leftarrow$ **new Schedule**$(|V|)$;
**Schedule** $S_{opt} \leftarrow$ some arbitrary schedule of $G$
**int** $n \leftarrow |V|$;
**int** $m_{opt} \leftarrow n$;  // upper bound for register need

**function** selection( **Node** $v$, **Set** $z$, **Array** INDEG)
  // select DAG node $v$ to be scheduled next in current $S$:
  let $P$ be the set of all parent nodes of $v$ in $G$
  **for all** $w \in P$ **do** $INDEG'[w] \leftarrow INDEG[w] - 1$ **od**;
  **for all** $u \notin P$ **do** $INDEG'[u] \leftarrow INDEG[u]$ **od**;
  let $P'$ be the subset of those nodes of $P$ which have indegree 0;
  $z' \leftarrow z - \{v\} \cup P'$;
  **return** $(z', INDEG')$;
**end** selection

**function** nce (**set** $z$,  **int[]** INDEG,  **int** pos)
  **if** $pos = n + 1$  // $S$ is a complete schedule of $G$
  **then**  **if** $m(S) < m_{opt}$ **then** $S_{opt} \leftarrow S$;  $m_{opt} \leftarrow m(S)$ **fi**
  **else**
    **for all** $v \in z$ **do**
      $(z', INDEG') \leftarrow$ selection$(v, z, INDEG)$;
      $S(pos) \leftarrow v$;
      nce $(z', INDEG', pos + 1)$;
    **od**
  **fi**
**end** nce
```

FIGURE 2.14: Algorithm ***nce*** enumerates all schedules of a DAG $G = (V, E)$.

a DAG node v from z'_1 and scheduling it. The selection tree edge h is labeled $\ell(h) = v$ by the DAG node v selected. Clearly, the selection tree has one root that corresponds to the initial set z_0 of zero-indegree DAG nodes. All leaves of the selection tree are instances of the empty set of DAG nodes. The number of selection tree edges leaving a selection tree node $z \in Z'$ is equal to the cardinality of z. According to the algorithm *nce*, the order of the labels $\ell((z_{j-1}, z_j))$ of the edges on any path $\pi = (z_0, ..., z_n)$, read from the root z_0 to a leaf z_n, corresponds one-to-one to a valid schedule $S_\pi : V \to \{1, ..., n\}$ of the DAG G with $S_\pi(\ell((z_{j-1}, z_j))) = j$, $j = 1, ..., n$; and for each schedule S of G there exists a unique corresponding path π_S in T from z_0 to some leaf.

Let *scheduled*(z) denote the set of DAG nodes that have already been scheduled when function *nce* is called with first parameter z. In particular, all DAG predecessors of the DAG nodes in z belong to *scheduled*(z), see Fig. 2.15. Furthermore, we denote by $L(z)$ the length of the path from the root z_0 to z, i.e. $L(z) = |scheduled(z)|$.

An already scheduled node of the DAG is ***alive*** at the end of a partial schedule of the DAG if it is an operand of a node that has not yet been scheduled, and must hence be held in a

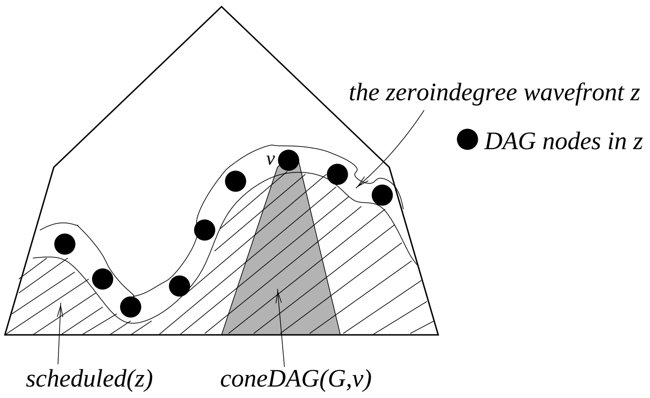


FIGURE 2.15: A snapshot of a DAG being processed by algorithm *nce*.

register:

Definition 2.12 *The* ***set of alive nodes*** *of a schedule S of a DAG G for a selection node z is*

$$\textit{alive}(z) = \{u \in \textit{scheduled}(z) : \ \exists (u,v) \in E, \ v \notin \textit{scheduled}(z)\}$$

and the number of alive nodes is $\textit{occ}(z) = |\textit{alive}(z)|$.

Hence, $\textit{occ}(z)$ denotes the number of values which are currently to be held in registers (assuming as usually that an optimal register allocation is used). Note that $\textit{occ}(z)$ is the same for all schedules of $\textit{scheduled}(z)$. Hence, $\textit{occ}(z)$ is a lower bound for the register need of a schedule of $\textit{scheduled}(z)$.

As example, for the DAG in Figure 2.16 we have
$\textit{scheduled}(\{c,d\}) = \{a,b\}$, $L(\{c,d\}) = 2$, $\textit{occ}(\{c,d\}) = 2$;
$\textit{scheduled}(\{a,b,c\}) = \emptyset$, $L(\{a,b,c\}) = 0$, $\textit{occ}(\{a,b,c\}) = 0$;
$\textit{scheduled}(\emptyset) = V = \{a,b,c,d,e,f,g,h\}$, $L(\emptyset) = n = 8$, $\textit{occ}(\emptyset) = 0$.

We have observed that in such a selection tree, there occur generally several different instances $z' \in Z'$ of the same set z. We will modify the algorithm *nce* in such a way that these multiple occurrences of z are eliminated and replaced by a single node named z. Thus, the computation of the modified algorithm will result in a *selection DAG* rather than a selection tree. Let us first introduce some notation.

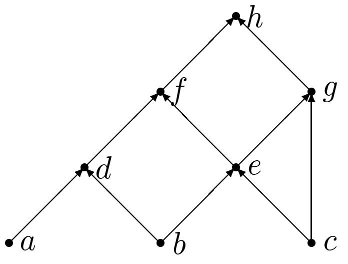
Definition 2.13 *A* ***selection DAG*** *$D = (Z,H)$ is a directed acyclic graph. The set Z of* ***selection nodes*** *contains all* different *sets z generated by nce. A* ***selection edge*** *$h = (z_1, z_2) \in H$ connects two selection nodes $z_1, z_2 \in Z$ iff there is a step in nce that directly generates z_2 from z_1 by selecting a DAG node $v \in z_1$.*

Note that a selection DAG has a single root, namely z_0, and a single sink, the empty set $\emptyset$.

Figure 2.16 shows an example DAG G and the selection DAG computed for it.

Each selection edge (z_1, z_2) is annotated with the DAG node $v = \ell((z_1, z_2))$ that is selected when going from z_1 to z_2.

By $G_z = (\textit{scheduled}(z), \ E \cap (\textit{scheduled}(z) \times \textit{scheduled}(z)))$ we define for each $z \in Z$ the subgraph of G induced by $\textit{scheduled}(z)$. Note that $G = G_\emptyset$.



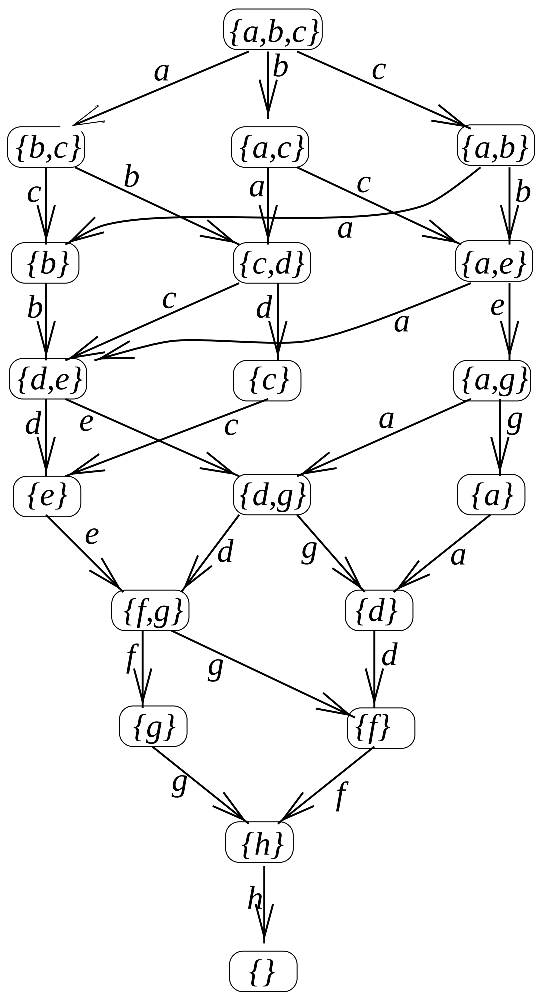


FIGURE 2.16: An example DAG and its selection DAG

Lemma 2.7 *Each path $\pi = (z_0, z_1, ..., z_{n-1}, z_n = z)$ of length n in the selection DAG $D = (Z, H)$ from the root z_0 to a selection node $z \in Z$ corresponds one-to-one to a schedule of scheduled(z) which is given by the mapping $S : V \to \{1, ..., n\}$ with $S(\ell((z_{j-1}, z_j))) = j$ for $j = 1, ..., n$.*

Proof: by induction on the length n of π. Initially, the path $\pi = (z_0)$ consists only of the root z_0 and has length $n = 0$, with *scheduled*$(z_0) = \emptyset$. Clearly, π corresponds one-to-one to the empty schedule. Now assume $n > 0$.
$\Rightarrow$: Let z be an inner node of D. z has some direct predecessor y in D. In y a DAG node $\ell((y, x)) = v \in y$ was selected by *nce*. Thus, *scheduled*$(y) =$ *scheduled*$(x) - \{v\}$ and $L(y) = L(x) - 1 = n - 1$. Consider the prefix path $\pi' = (z_0, ..., z_{n-1} = y)$ of π to y. We apply the induction hypothesis to π', yielding a schedule $S_{\pi'}$ for *scheduled*(y). We generate a schedule S_π for *scheduled*(z) by just adding $S_\pi(v) = n$ to $S_{\pi'}$.
$\Leftarrow$: Consider a schedule S of n DAG nodes. Let $v = S^{-1}(n)$. We define $V_z = \{S^{-1}(1),...,S^{-1}(n)\}$ and $V_y = V_z - \{v\}$. Let S_y be some arbitrary schedule for V_y. By the induction hypothesis there is a path $\pi_y = (z_0, ..., z_{n-1})$ in D to some selection node $z_{n-1} = y$ with *scheduled*$(y) = V_y$. Since v can be scheduled next (i.e. at position n), there must exist a selection edge (y, z) to some selection node z with $\ell((y, z)) = v$, *scheduled*$(z) = V_z =$ *scheduled*$(y) \cup \{v\}$ and $L(z) = n$. Thus, $\pi_S = (z_0, ..., y, z)$

is the path corresponding to schedule S. □

This means that subpaths corresponding to different partial schedules of the same subsets of DAG nodes end up in the same selection node. For instance, the paths inscripted a, b, c and b, c, a in Figure 2.16 both end up in selection node $\{d, e\}$. If we read the edge labels along any path from the root to the sink of the selection DAG, we obtain a valid schedule of G.

Corollary 2.8 *All paths π in the selection DAG $D = (Z, H)$ from the root to a selection node $z \in Z$ have equal length $L(z)$.*

Corollary 2.9 *D is leveled, i.e. the nodes z of D can be arranged in $n+1$ levels $L_0, L_1, ..., L_n$ such that the selection edges in D connect only selection nodes in neighbored levels L_{i-1}, L_i, $i = 1, ..., n$.*

Proof: Set $i = L(z)$ in corollary 2.8. □

Up to now, the selection DAG is nothing more than a compressed representation of the selection tree, as it allows to reproduce all schedules generated by algorithm *nce*. Now we will present and prove a key lemma that enables using the selection DAG as the underlying structure of a dynamic programming algorithm to minimize the register need.

A direct consequence of Lemma 2.7 is that for a (partial) schedule S of a DAG node set *scheduled*(z), corresponding by Lemma 2.7 to a path $\pi = (z_0, z_1, ..., z_n = z)$, we may exchange any length-k prefix of π, with $k < n$, by taking a different path towards z_{k+1}, resulting in a new path $\pi' = (z_0, z'_1, ..., z'_k, z_{k+1}, ..., z_n = z)$, i.e. a new schedule S' of *scheduled*(z). A key observation is that for both S and S', after executing the kth instruction (and thus arriving at the same zeroindegree set z_{k+1}), the same set *alive*(z_{k+1}) of nodes will be held in registers. Hence, it seems reasonable to change to a prefix path that minimizes the register need for the resulting schedule. We exploit this property now to show that it suffices to keep *one* arbitrary optimal schedule of *scheduled*(z) for each selection node z:

Lemma 2.10 *Consider all direct predecessors $y_1, ..., y_q \in L_{i-1}$ of a selection node $z \in L_i$. Let $v_j = \ell((y_j, z))$, $j = 1, ..., q$, and let S_{y_j} denote an arbitrary optimal schedule for scheduled(y_j), $j = 1, ..., q$. We construct schedules $S_z^{(j)}$ for scheduled(z) by just adding $S_z^{(j)}(v_j) = i$ to schedule S_{y_j}, $j = 1, ..., q$. Then, one of these schedules, $S_z^{(j')}$, with $m(S_z^{(j')}) \leq m(S_z^{(j)})$ for all $j = 1, ..., q$, is an optimal schedule of scheduled(z).*

Proof: (indirect) It is sufficient to consider only the subset J consisting of just those indices j where S_{y_j} contributes to schedules $S^{(j)}$ with *minimal* register need, i.e., for each $j \in J$ we have $m(S_z^{(j)}) = m' = \min_{1 \leq k \leq q} m(S_z^{(k)})$. We show: If m' is *not* equal to the (global) optimum m^* for *scheduled*(z), i.e. if no $S_z^{(j)}$ is optimal for *scheduled*(z), then there is at least one $j \in J$ where the schedule S_{y_j} used to build $S_z^{(j)}$ was not optimal. Comparing the register need of S_{y_j} and $S_z^{(j)}$, we classify the indices $j \in J$ into two classes: those where the register need remains the same after scheduling v_j, and those where it has increased.

Class 1: $m(S_{y_j}) = m'$. For these cases, a register was reused to store v_j. Hence, suboptimality of $S_z^{(j)}$ implies suboptimality of S_{y_j} for each j in class 1.

Class 2: $m(S_{y_j}) = m' - 1$. In these cases, a new register had to be used for v_j, as all other $m' - 1$ registers had been occupied, i.e. *occ*$(y_i) = m' - 1$ (because we always use an optimal register allocation). As *occ*(y_i) is a lower bound for the register need, any schedule S of *scheduled*(y_j) that uses only $m(S) = m' - 1$ registers would be optimal for *scheduled*(y_j). (Note that, by the construction, such a schedule exists for each j in class 2, namely S_{y_j}.) But this cannot hold simultaneously for all j in class 2, because then all $S^{(j)}$ must be optimal as well, which is a contradiction to our assumption that $m' > m^*$. □

For each selection node z we further define:
Σ_z as the set of all schedules for G_z;
S_z as one of the schedules in Σ_z with minimal register need, and
$m_z = m(S_z)$ as this minimal register need. Note that $m_\emptyset$ is then the register need of an optimal schedule for G.

The definition of Σ_z and the selection of S_z from Σ_z is well-defined due to Lemma 2.7 and Lemma 2.10.

The new algorithm ***ncc*** (see Figure 2.17) constructs the selection DAG instead of the selection tree. *ncc* generates the selection nodes levelwise. At any creation of a new zeroindegree set z', it inspects whether a selection node for the same DAG node set has already been created before. Because D is leveled (see corollary 2.9), the algorithm needs only to inspect already generated selection nodes in the current level L_{i+1}. If such a selection node z' does not yet exist, a new selection node z' is created and $S_{z'}$ is initialized to the current schedule S. Otherwise, if the selection node z' already exists, *ncc* computes whether the current schedule S or the schedule $S_{z'}$ uses fewer registers, and sets $S_{z'}$ to the better schedule of these two.

Finally, each selection node z is annotated by an optimal schedule S_z of *scheduled*(z), and consequently $S_\emptyset$ is an optimal schedule for the DAG G.

From Lemma 2.10 the correctness of *ncc* follows by induction:

Theorem 2.11 *Algorithm ncc computes an optimal schedule for the DAG G.*

Theorem 2.12 *The worst-case time complexity of algorithm ncc is* $O(n^2 2^n)$.

Proof: The number of selection nodes is bound by the number of subsets of V, which is 2^n. Each zeroindegree set has at most n elements. Hence, the call to ***selection*** is executed at most $n2^n$ times. The function ***selection*** uses at most linear time to compute the new zeroindegree set and the update of the indegree array. Assuming that we use, for each list L_i, a binary search tree[10] for storing the selection nodes in lexicographic order of the zeroindegree sets, the ***lookup*** call, executed at most $n2^n$ times, takes time logarithmic in the maximum number of items stored in list L_{i+1}, which is less than 2^n, hence a ***lookup*** call takes linear time in the worst case. The same holds for the ***insert*** call. □

In practice, *ncc* performs considerably better than this worst case scenario. One reason for this is that not all 2^n subsets of V can occur as zeroindegree sets. Also, the number of possible selections for a zeroindegree set is just its size, which is usually much smaller than our conservative estimation n.

[10] A naive linear search, as adopted in the prototype implementation, is only suitable for small sets of selection nodes. In practice, searching in large sets of selection nodes could be additionally sped up by hashing.

```
function ncc ( DAG $G$ with $n$ nodes and set $z_0$ of leaves)
 $L_i \leftarrow$ empty List $\forall i > 0$;   $L_0 \leftarrow$ new List($z_0$);   $S_{z_0} \leftarrow \emptyset$;
 for level $L_i$ from $L_0$ to $L_{n-1}$ do
   for all $z \in L_i$ do
     for all $v \in z$ do
       $(z', INDEG') \leftarrow$ selection$(v, z, INDEG)$;  // selecting $v$ produces a selection node $z'$
       $S \leftarrow S_z \bowtie \{v\}$;   // get schedule $S$ of scheduled$(z')$ from $S_z$ by adding $S(v) = i+1$
       $z'' \leftarrow L_{i+1}$.lookup$(z')$;
       if $(z'')$  // $z'$ already exists in $L_{i+1}$
       then   // compare $S$ and $S_{z'}$:
         if $m(S) < m(S_{z''})$ then $S_{z''} \leftarrow S$ else forget $S$ fi
       else $L_{i+1}$.insert$(z')$;  $S_{z'} \leftarrow S$
       fi
     od
   od
 od
 return $S_\emptyset$ in $L_n$.first()
end ncc
```

FIGURE 2.17: Algorithm ***ncc*** constructs the selection DAG for a DAG G and computes an optimal schedule.

The comparison for equality and for lexicographic precedence of zeroindegree sets can be done very fast in practice by representing zeroindegree sets as bitvectors.

2.4.2 Improvement

We improve the algorithm *ncc* by the following modification: Instead of entering all selection nodes of the same level into the same set L_i, we subdivide L_i into sets L_i^0, L_i^1, ..., L_i^K where K is some upper bound of the minimal number of registers required for G.[11] Now we store in L_i^k exactly those selection nodes z of L_i that have register need $m_z = k$. Now, we no longer have to look up all L_i^k, $k = 1, ..., K$ when looking for equality of z sets because of data dependencies:

Lemma 2.13 *The predecessors of a selection node in L_{i+1}^k are located in either L_i^{k-1} or L_i^k.*

Proof: Appending a DAG node v to an existing schedule S with register need $m(S)$ giving a new schedule S' yields $m(S) \leq m(S') \leq m(S) + 1$ since we could be required to allocate a new register for v that was previously unused, and a composed schedule S' can never take less registers than any of its sub-schedules. □

This leads to the data dependencies in the selection graph illustrated in Figure 2.19.

We use this theorem to free storage for levels as soon as they are not required any more.

[11] K could be a priori determined by using contiguous schedule techniques or by just choosing a random schedule of G. If storage is not too scarce, we can conservatively choose $K = n$.

```
function ncv ( DAG $G$ with $n$ nodes and set $z_0$ of leaves)
 $L_i^k \leftarrow$ empty list $\forall i\ \forall k$;  $L_0^0 \leftarrow$ new List$(z_0)$;  $S_{z_0} = \emptyset$;
 for $k = 1, ..., K$ do    // outer loop: over space axis
   for level $i$ from 0 to $n-1$ do
     for all $z \in L_i^{k-1}$ do
       for all $v \in z$ do
         $(z', INDEG') \leftarrow$ selection$(v, z, INDEG)$;
         $m \leftarrow m(S)$ with $S \leftarrow S_z \bowtie v$;    // by adding $S(v) = i+1$
         if $(!L_{i+1}^{k-1}$.lookup$(z'))$
           if $m = k-1$
             $L_{i+1}^k$.remove$(z')$;       // (if any: move $z'$ from $L_{i+1}^k$ to $L_{i+1}^{k-1}$)
             $L_{i+1}^{k-1}$.insert$(z')$;  $S_{z'} \leftarrow S$;
           else if $(!\ L_{i+1}^k$.lookup$(z'))$
             $L_{i+1}^k$.insert$(z')$;  $S_{z'} \leftarrow S$;  fi
           fi
         fi
       od
     od
   od
   if $L_n^{k-1}$.nonempty$()$
     then return the schedule $S_z$ for some $z \in L_n^{k-1}$ fi
   fi
 od
end ncv
```

FIGURE 2.18: Algorithm ***ncv***.

The data dependencies impose a partial order of the selection steps. Hence, we can change our algorithm's order of constructing selection nodes as long as there is no conflict with these data dependencies. For instance, we can change the priority of the time slot axis over the register need axis, which corresponds, roughly speaking, just to a loop interchange. The resulting algorithm ***ncv*** (see Figure 2.18) can finish computing selection nodes as soon as the (first) selection node is entered into level L_n, because it represents a valid schedule with minimal register need—just what we want! Thus, expensive computation of later unused, inefficient partial schedules is deferred as far as possible—if we are lucky, to a point that needs not be considered any more because a solution has been found meanwhile.

2.4.3 Parallelization

There are several possibilities to exploit parallelism in *ncv*:

- exploiting the structure of the data dependencies among the lists L_i^k (see Figure 2.19): all lists L_i^{j-i}, $i = 1, 2, ...$ (the lists on the jth "wavefront" diagonal) can be computed concurrently in step j, for $j = 1, ..., n+K$.

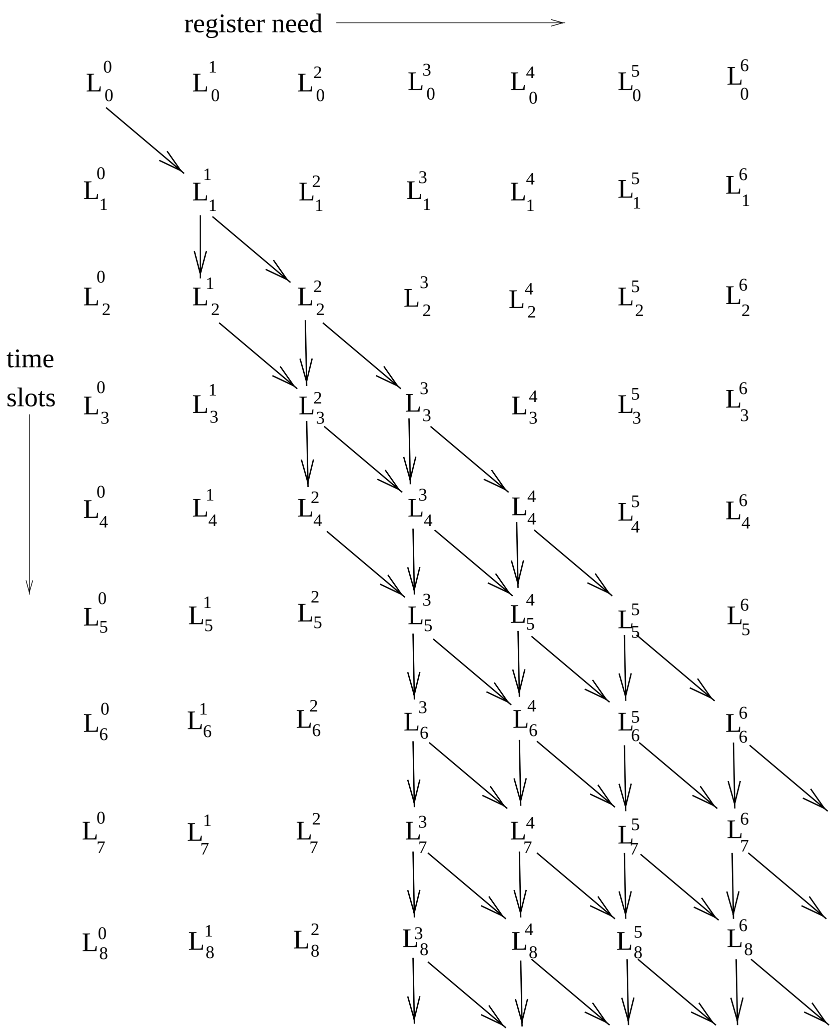


FIGURE 2.19: Data dependencies among the sets L_i^k of selection nodes.

- further subdivision of a list L_i^k into sublists of roughly equal size (hashing) and expanding the selection nodes in these sublists concurrently. Note that this requires a parallel list data structure that guarantees that the same selection node is not erroneously inserted twice at the same time.
- parallelization of the lookup routine (according to previous item).
- parallelization of the expansion step itself.

Particularly, the second and third item promise the exploitation of massive parallelism.

2.4.4 Experimental Results

We have implemented *nce*, *ncc* and *ncv*, and have applied them to hundreds of randomly generated test DAGs and to larger DAGs taken from real application programs. Some of the

n	T_{ncv}	$m_\emptyset$
24	0.8 sec	8
26	1.5 sec	7
26	0.8 sec	7
27	0.1 sec	8
28	1.4 sec	8
30	4.6 sec	8
31	22.5 sec	7
31	48.6 sec	7
32	17.5 sec	8
32	1:14.9 sec	8
33	4.6 sec	10
34	25.3 sec	7
34	54.3 sec	9
35	34.4 sec	10
36	14.9 sec	10
36	32.4 sec	10
37	3:00.8 sec	9
38	56.1 sec	10
38	2:38.8 sec	9
38	28.8 sec	9
38	7.0 sec	10
38	3:35.8 sec	9
39	10:01.7 sec	10
39	34.4 sec	9
39	43.1 sec	10
39	3:39.6 sec	9
39	1:44.8 sec	10
41	14:28.1 sec	9
41	1:48.6 sec	11
41	1:09.3 sec	10
42	23.4 sec	10
42	46.9 sec	11
43	43:30.6 sec	9
43	23:51.7 sec	9
44	3:44.6 sec	11
44	5:20.7 sec	12
44	18.9 sec	11
44	10:30.9 sec	10
45	58:01.7 sec	10
46	36:46.0 sec	12
46	55:59.8 sec	10
47	22:30.0 sec	11
49	1:09:37.2 sec	10
50	32:58,1 sec	12
51	19:01.2 sec	13

TABLE 2.5: *ncv* applied to some random DAGs. For all these DAGs, *nce* failed because of running out of space and time. (Measurements taken 1996 on a SUN SPARC-10)

experimental results are shown in Tables 2.5 and 2.6. All measurements were taken on a SUN SPARC-20.

The random DAGs are generated by initializing a predefined number of nodes and by selecting a certain number of them as leaf nodes. Then, the children of inner nodes are selected randomly. We observed:

- *nce* is not practical for DAGs with more than 20 nodes and often takes inacceptable time for DAGs with 15 to 20 nodes.
- *ncc* reduces the number of different contiguous schedules considerably. It is, roughly spoken, practical for DAGs with up to 40 nodes, sometimes also for larger DAGs, stongly depending on the DAG's structure. A timeout feature, controlled e.g. by a compiler option, should be provided for practical use that switches to a heuristic method if the deadline is exceeded..
- *ncc* is particularly suitable for "slim" DAGs because the zero-indegree sets z always remain small.
- *ncv* defers the combinatorial explosion to a later point of time but, of course, cannot always avoid it for larger and more complex DAGs. For some DAGs in the range of 41...50 nodes and for nearly all DAGs with more than 50 nodes *ncv* runs out of space and time.

2.4.5 Simultaneous Optimization of Register Space and Execution Time

Up to now, we assumed that there are no delay slots to be filled since the result of an operation was available at the beginning of the next machine cycle. However, modern processors often have delayed instructions with one or more delay slots. For instance, in most processors the `Load` instructions are delayed such that the processor may be kept busy while the value

Source	DAG	n	T_{ncv}	$m_\emptyset$
LL 14	second loop	19	0.12 sec	4
LL 20	inner loop	23	0.48 sec	6
MDG	calculation of $\cos, \sin$,...	26	0.44 sec	7
SPEC77	spherical flow	27	0.93 sec	7
SPEC77	multiple FFT analysis	49	23:26.0 sec	7

TABLE 2.6: *ncv* applied to some DAGs taken from real programs (LL = Livermore Loop Kernels; MDG = molecular dynamics, and SPEC77 = atmospheric flow simulation; the last two are taken from the Perfect Club Benchmark Suite). *nce* failed for all DAGs because of running out of space and time. (Measurements taken 1996 on a SUN SPARC-10)

is fetched from the slower memory. These delay slots may thus be filled by subsequent operations that do not use the result of the delayed operation. If there are no such operations available, a `NOP` instruction has to be inserted to fill the delay slot. We will show how our technique can be easily extended to cope with delay slots.

For each DAG node v, let *delay*$(v) = \Delta($
$ttype(v))$ denote the number of delay slots required by the operation performed at v. For a given schedule S, we compute a mapping T of the DAG nodes to the set $\{1, 2, ...\}$ of physical time slots as given in Figure 2.20. The algorithm returns the time *time*(S) which is required to execute the basic block reordered according to schedule S. The number of `NOP` instructions implied by S is then just *time*$(S) - n$.

A schedule is time-optimal if its execution on the target processor takes not more cycles than any other schedule:

Definition 2.14 *A schedule S of a DAG G is called* **time-optimal** *iff time$(S) \leq$ time(S') for any schedule S' of G.*

We are faced with the problem that we now get a second optimization goal: minimizing *time*(S), i.e. the number of `NOP`s, in addition to minimizing the number $m(S)$ of registers. When minimizing simultaneously for space and time, comparing two schedules S_1

```
function time ( Schedule $S$ of a DAG $G$ with $n$ nodes )
Initially we set $t \leftarrow 1$. Then,
for the DAG nodes $u_i$ in the order induced by $S$, i.e., $u_i = S^{-1}(i)$, $i = 1, ..., n$ do
  if $u_i$ has a left child $l$ then $t = \max(t, T(l) + delay(l) + 1)$ fi
  if $u_i$ has a right child $r$ then $t = \max(t, T(r) + delay(r) + 1)$ fi
  $T(u_i) = t$; $t = t + 1$;
od
return $\max_i(T(u_i) + delay(u_i))$; end time
```

FIGURE 2.20: Algorithm *time* computes the time slots and the execution time of a schedule S on a single-issue pipelined processor. A similar formulation of this algorithm is given e.g. in [Muc97, Chapter 17].

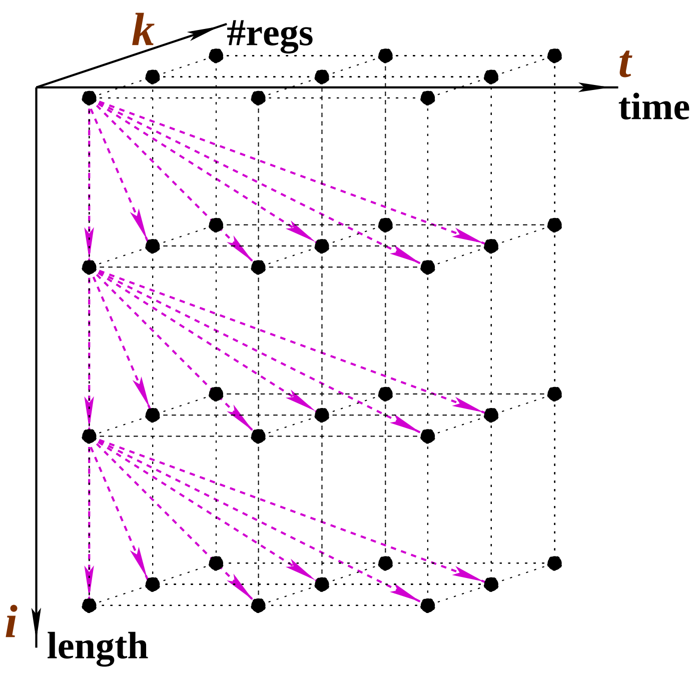


FIGURE 2.21: Three-dimensional arrangement of the selection nodes, with dependencies (shown for one node only in each level) among the lists $L_i^{k,t}$ of selection nodes, here for *maxdelay* $= 2$. This space may be traversed in any order that does not conflict with the dependencies.

and S_2 becomes more difficult. Clearly, if $\mathit{time}(S_1) < \mathit{time}(S_2)$ and $m(S_1) \leq m(S_2)$, or if $\mathit{time}(S_1) \leq \mathit{time}(S_2)$ and $m(S_1) < m(S_2)$, we prefer S_1 over S_2. In the context of our previous algorithm, this means that S_1 would be kept in the lists of schedules, and S_2 would be thrown away. But what happens if S_1 and S_2 have both equal register need and equal number of `NOP`s, or, even worse, if $\mathit{time}(S_1) > \mathit{time}(S_2)$ but $m(S_1) < m(S_2)$ (or vice versa)? The only safe method is to keep both S_1 and S_2 in the lists for later expansion. This clearly increases the computational work.

But we apply the same trick as above to defer computationally expensive expansions to a later point in time. In addition to the previous partition of the selection nodes into lists L_i^k (see Figure 2.19), we add a third dimension to the space of lists of selection nodes z, namely execution time $\mathit{time}(S_z)$ (see Figure 2.21). Specifically, a list $L_i^{k,t}$ contains the selection nodes z with $|\mathit{scheduled}(z)| = i$ and $m_z = k$ and $\mathit{time}(S_z) = t$. The structure of data dependencies among the $L_i^{k,t}$, as shown in Figure 2.21, results from

Lemma 2.14 *Let maxdelay* $= \max_v \mathit{delay}(v)$. *The predecessors of a selection node in* $L_{i+1}^{k,t}$ *are located in* $L_i^{k-1,\vartheta}$ *or* $L_i^{k,\vartheta}$, *with* ϑ *ranging from* $t -$ *maxdelay to* d.

Proof: Evidence regarding k was already given in Lemma 2.13. Appending a DAG node v to an existing schedule S with execution time $\mathit{time}(S)$, giving a new schedule S', yields $\mathit{time}(S) \leq \mathit{time}(S') \leq \mathit{time}(S) + \mathit{maxdelay}$ since v can be placed safely *maxdelay* time slots after the last time slot occupied by S. □

Another, even more problematic difficulty arises if the execution time is to be minimized. For the previous algorithm, *ncv*, all space-optimal schedules for the same set of nodes were equivalent also when later used as sub-schedules in subsequent stages of the selection DAG (see Lemma 2.10). Thus, *ncv* needed to keep only one optimal schedule for each $z \in L_i^k$. For time optimization with delayed instructions, though, Lemma 2.10 does not hold: whether a schedule S_1 for a zeroindegree set z is superior to another schedule S_2 for z cannot be decided immediately by comparing just the "current" time consumption *time*(S_1) and *time*(S_2). Instead, it depends on *subsequent*, not yet scheduled instructions, whether S_1 or S_2 or both of them should be kept. Informally spoken, when optimizing execution time by just considering the "past", *ncv* may keep a "wrong" optimal schedule and may throw away the "right" ones, but this will become apparent only in the "future", i.e. at a later stage of the scheduling algorithm.

This is illustrated in the following example (see Figure 2.16): When selection node $\{d, e\}$ has been completed, *ncv* keeps only one optimal schedule for *scheduled* $(\{d, e\}) = \{a, b, c\}$, e.g. (b, c, a). The optimal solution found by *ncv* may follow a path through $\{d, e\}$. But if the final schedule should still require only 4 registers, *ncv must* continue with node d. But the schedule (b, c, a, d) enforces a NOP after issuing a if the DAG leaves have one delay slot (delayed Load). Thus the final schedule computed may be suboptimal w.r.t. execution time because the decision to keep only (b, c, a) at $\{d, e\}$ was made too early.

To repair this deficiency, we adopt a brute-force approach and simply keep, for each selection node z, *all* schedules that may still lead to a time-optimal schedule of the DAG, rather than only one, and some of these may even be nonoptimal for *scheduled*(z). Clearly, this increases optimization time (and space) considerably, since all these schedules for *scheduled*(z) have to be taken into consideration now when expanding selection node z. An improvement of this straightforward strategy is discussed in Section 2.4.6.

As we have two different optimization goals (number of registers and number of NOPs) we have to consider trade-offs between them. We can select one of them as primary goal of optimization and the other one as a secondary goal, which determines the order of stepping through the search space $\{L_i^{k,t}\}_{i,k,t}$. Or we formulate a mixed criterion (e.g., a weighted sum of k and d) that results in the algorithm traversing the search space in a wavefront manner. It is up to the user to formulate his favorite optimization goal.

We implemented algorithm ***ncn***, the modification of *ncv* for delayed instructions. The delays can be defined arbitrarily. For instance, the processor of the SB-PRAM [KPS94] has a delayed Load with one delay slot; all other operations perform in one cycle. We decided to minimize execution time as a primary goal and register need as a secondary goal. It is very easy now to backtrack the algorithm if one is willing to trade more NOPs for decreased register space. The results for several DAGs at different delay configurations are given in Table 2.7. It appears that the program runs quickly out of space for $n > 25$ in the case of delayed Loads of 1 delay slot, and for $n > 20$ if more delay slots are assumed. This is due to the unavoidable (but perhaps optimizable) replication of schedules described above. Nevertheless this is still better than *ncc*. Clearly, our test implementation of *ncn* still wastes a lot of space. Section 2.4.6 proposes the concept of time–space profiles, which may help to reduce the space and thus time requirements of *ncn*, especially for the case that execution time is to be minimized with higher priority. In any case *ncn* appears to be feasible at least for small DAGs.

In general, optimization seems to be the faster, the more limitations and dependencies there are. Thus it is a straightforward idea to extend our method for processors with multiple functional units.

In the presence of multiple functional units it becomes possible to exploit fine grained

DAG	n	delay slots for `Loads`	avg. delay slots for compute	T_{ncn}	min. reg. need	#nops at min. reg. need	reg. need at min. #nops	min. #nops
LL 14	19	1	0	2.2 sec	4	4	5	0
		2	0.5	15.2 sec	4	11	6	1
LL 20	23	1	0	18.5 sec	6	0	6	0
random	17	1	0	0.7 sec	5	1	6	0
random	19	1	0	6.4 sec	5	2	6	0
random	19	1	0	14.8 sec	5	4	6	0
random	20	1	0	30.3 sec	6	2	7	0
random	21	1	0	11.8 sec	8	0	8	0
random	25	1	0	10.4 sec	7	0	7	0
random	15	2	0.5	0.1 sec	5	3	6	1
random	16	2	0.5	4.1 sec	5	3	6	1
random	16	2	0.5	1.1 sec	6	1	7	0
random	17	2	0.5	10.2 sec	5	7	6	1
random	19	2	0.5	62.5 sec	5	9	8	0
random	21	2	0.5	5.5 sec	6	2	7	0
random	22	2	0.5	45.3 sec	6	5	7	0

TABLE 2.7: Real and random DAGs, submitted to *ncn* with two target machine constellations: (1) delayed `Load` with one delay slot, and (2) delayed `Load` with 2 delay slots and delayed `Compute` with 0 or 1 delay slots (randomly chosen with probability 0.5). Columns 6 and 7 show register need and number of `NOP`s if register need is minimized; columns 8 and 9 show the results if the number of `NOP`s is minimized. For DAGs of more than 25 resp. 22 nodes our *ncn* implementation ran out of space. Interesting tradeoffs between register need and execution time occur mainly for small DAGs.

parallelism. Consider a schedule S and two DAG nodes u, v such that v is scheduled directly after u. If u is not a DAG predecessor of v, u and v could be executed in parallel, provided that both are ready to be issued and functional units are available for the operations to be performed at u and v.

In contrast to VLIW machines, today's superscalar microprocessors do not rely on compilers to schedule operations to functional units. Scheduling is done on-line in hardware by a dispatcher unit. The dispatcher is able to issue several subsequent instructions from the instruction stream in the same clock cycle as long as there are (1) enough functional units available to execute them, and (2) there are no structural hazards such as control dependencies or data dependencies that would prohibit parallel execution These dependences are determined on-line by the dispatcher. If a subsequent instruction v is control or data dependent on a previous one, u, that has just been issued in this cycle, or if all suitable functional units are busy, the later instruction has to be delayed.

We see that this online scheduling performed by the dispatcher is indeed bounded by the rather small lookahead. Thus, it is a reasonable idea to reorder the instruction stream at compile time to facilitate the dispatcher's job. In order to optimize the schedule in this context, we need exact knowledge of the hardware features such as number, type, and speed of functional units, and the dispatcher's functionality which is then just simulated during optimization, as we already did for the pipeline behaviour in the previous section.

We have extended algorithm *ncn* for this case. Only function *time* must be adapted. The result is an optimal schedule S with issuing-time slots T and a mapping F of the DAG nodes to the functional units, such that the overall time required by S is minimized. T and F are computed as side-effects of the modified function *time* which just imitates the dispatcher's functionality.[12]

Nevertheless, care must be taken for superscalar processors where the order of instructions given by the list schedule may be reversed by the dispatcher (*out-of-order issue*) or the result writing phases may be interchanged (*out-of-order completion*). For instance, if a node v_j that occurs in the list schedule S later than a node v_i but is executed earlier (because the functional unit for v_j is free while v_i must wait), it may happen that v_j overwrites a register that was not yet freed as it is an operand of v_i, i.e. a conflict of the time schedule with the register allocation. In that case, the register allocation and the time schedule cannot longer be computed independently of each other from the list schedule. Either the register allocation must be computed as a function of the time schedule (if the first optimization goal is time) or the time schedule as a function of the register allocation (otherwise), such that execution of v_j (as mentioned above) writing to register r can start only after all instructions v_i occurring earlier in S reading an operand from register r have read them (*in-order issue*). In general, it is sufficient to consider these constraints just for the cycle where v_j writes to r, usually in its last delay slot (*in-order completion*).

In practice, it is thus not always possible to exactly predict the execution time for advanced superscalar processors, because the hardware may internally allow out-of-order issue, out-of-order execution, or out-of-order completion of instructions and applies on-line dependence analysis, reorder buffers and hidden registers to reestablish the semantics implied by the program, such that the actual execution time may differ from the idealized processor model used by the compiler for the prediction in the *time* function. In our implementation of *time* for superscalar processors and the measurements in Tables 2.12 and 2.13 we assumed a target processor with (from the assembler programmer's point of view) in-order issue and in-order completion. The results apply as well to out-of-order issue architectures as long as there are enough internal registers to buffer results so that these can be written back in program order.

[12]Additional choices may exist here for multi-issue architectures where the instructions may be placed explicitly by the programmer at arbitrary time slots, such as for VLIW architectures. For example, consider two instructions u and v where u precedes v immediately in a schedule S, and u and v are to be executed on different units, U_u and U_v. If u is a child of v, there is no choice, v must be executed when the result computed by u is available. Instead, if v does not depend on u, they may be (in an in-order issue system, see the following discussion) issued simultaneously if both units are free. For that case, a schedule S' derived from S by interchanging u and v would result in the same time schedule. On the other hand, there is no means to express that one of u or v should be delayed to the latest time slot where it can still be issued without changing the overall execution time, with the goal of reducing register pressure between the issue slots of u and v. As both S and S' are considered by the optimization algorithm, this "flushright" option could be exploited by installing an arbitrary total order to among the functional units. If $to(U_u) < to(U_v)$, S implies that u and v are issued in the same time slot, while S' implies that u is flushed to the right after the overall schedule has been computed. Otherwise, S implies that v is flushed to the right, while S' implies that u and v are issued in the same time slot.

2.4.6 Time–Space Profiles

The naive strategy of keeping all optimal schedules for a selection node described above can be improved if we summarize all schedules that are "compatible" with respect to their time behaviour and keep a single schedule for them, as they are all equivalent if used as subschedules for subsequent selections. This "compatibility" is formally defined by *time profiles*, as an extension of the selection node concept.

Definition 2.15 *A **time profile** of a schedule S for an **out-of-order multi-issue** pipelined processor, consisting of f functional units U_1,...,U_f where unit U_j has maximum delay D_j, is a tuple*

$$\begin{aligned} P(S) \;\; = \;\; & (t_1, u_{1,t_1-D_1+1}, ..., u_{1,t_1}, \\ & t_2, u_{2,t_2-D_2+1}, ..., u_{2,t_2}, \\ & \ldots, \\ & t_f, u_{f,t_f-D_f+1}, ..., u_{f,t_f}) \end{aligned}$$

that consists of the f time profiles for each unit U_j, where t_j is the time slot where the last DAG node assigned to unit U_j is scheduled in S, and $u_{j,i}$ denotes the DAG node executed by unit U_j in time slot i, for $t_j - D_j + 1 \leq i \leq t_j$. Some of the $u_{j,i}$ entries may be NULL (–) where no node could be executed by U_j at time i. For a zero-delay unit U_j, only the t_j entry is stored.

*A **time profile** of a schedule S for a **single-issue** or **in-order multi-issue** pipelined processor, consisting of f functional units U_1,...,U_f where unit U_j has maximum delay D_j, is a tuple*

$$P(S) = (t, u_{1,t-D_1+1}, ..., u_{1,t}, u_{2,t-D_2+1}, ..., u_{2,t}, \ldots, u_{f,t-D_f+1}, ..., u_{f,t})$$

that consists of the f time profiles for each unit U_j, where t is the time slot where the last DAG node in S is scheduled, and $u_{j,i}$ denotes the DAG node executed by unit U_j in time slot i, for $t - D_j + 1 \leq i \leq t$. Again, some of these $u_{j,i}$ may be NULL (–) where no node could be executed by U_j at time i. For a zero-delay unit U_j, no $u_{j,i}$ entry is stored.

*The time profile of a schedule S is extended to a **time–space profile** of S by adding the register need m of S as another component of the tuple.*

Hence, a time profile contains all the information required to decide about the earliest time slot where the instruction selected next can be scheduled. All schedules for a selection node (up to now characterized by a zeroindegree set only) with the same time profile are equivalent if used as a prefix subschedule in subsequent selections. Note also that by a zero-indegree set z, its "space profile" *occ*(z) is implicitly given.

For retrieving selection nodes, the zeroindegree set is now no longer sufficient as a unique key:

Definition 2.16 *An **extended selection node** is a pair (z, P), consisting of a zero-indegree set z and a (time or time–space) profile P.*

If the register need is used as an optimization criterion, it must be a key component of extended selection nodes as well. Hence, when solving RCMTIS or SMRTIS, an extended selection node covers only schedules with the same time and space consumption.

Not all possible combinations of zero-indegree sets and time profiles occur in practice. Technically, extended selection nodes can be retrieved efficiently e.g. by hashing.

The previous algorithm *ncc* can now be generalized to extended selection nodes as follows: For each extended selection node (z, P), only one schedule S, which is optimal for that node, needs to be stored. When creating a new schedule S' from a schedule S stored in an extended selection node (z, P) by appending a DAG node v, its profile $P(S')$ can be computed incrementally from $P = P(S)$. If an extended selection node for the new zero-indegree set with the same key $(z', P(S'))$ does already exist, S' can be ignored. Otherwise, a new extended selection node $(z', P(S'))$ is created and annotated with S' as the currently best schedule for that node.

Note that the extended selection nodes can again be arranged in lists, like the $L_i^{k,t}$ defined above for *ncn*. Hence, the dependency structure and traversal strategy used for *ncv* is now applied to the extended selection nodes accordingly.

As an example, consider again the DAG of Figure 2.16. Let us assume now that we have an single-issue target processor with $f = 2$ functional units, where unit U_1 is not delayed and unit U_2 is delayed by one cycle. For our example we assume that operations b and e are delayed by one CPU cycle, i.e. to be executed on unit U_2, while the other operations are to be executed on unit U_1. The six possible schedules for the selection node $\{d, e\}$ have the same register need (3), and there exist just two different time profiles, namely $(3, b)$ if b was scheduled third, and $(3, -)$ if b was scheduled first or second. Hence, it is sufficient to store only the two extended selection nodes $(\{d, e\},\ (3, b, 3))$ and $(\{d, e\},\ (3, -, 3))$, and keep just one schedule in each of them.

The extension of *ncc* for extended selection nodes is based upon two key observations. First, the correctness of the algorithm is installed by the following equivalent of Lemma 2.10:

Lemma 2.15 *All schedules whose corresponding paths in the extended selection DAG end up in the same extended selection node (z, P) are equivalent with respect to their time and space consumption if used as a prefix of schedules derived in subsequent scheduling decisions.*

Second, the algorithm can safely ignore extended selection nodes with an inferior time profile, as long as only time consumption is considered as optimization goal. For simplicity we define *time-inferiority* of a time profile here only for a single-issue architecture:

Definition 2.17 *An extended selection node (z, P) with time profile $P = (t, u_{1,t_1}, ..., u_{f,t})$ is **time-inferior** iff there exists an extended selection node (z, Q) with time profile $Q = (t', v_{1,t_1}, ..., v_{f,t})$ for the* same *zeroindegree z, such that $t' < t$ and $u_{i,j} = v_{i,j}$ for all i, j, $1 \le i \le f$, $t_i \le j \le t$.*

Definition 2.18 *An extended selection node (z, P) with time–space profile $P = (t,\ u_{1,t_1},\ ...,\ u_{f,t},\ m_u)$ is **space-inferior** iff there exists an extended selection node (z, Q) with time–space profile $Q = (t', v_{1,t_1}, ..., v_{f,t}, m')$ for the* same *zeroindegree z, such that $m' < m$.*

Lemma 2.16 *When optimizing execution time only, time-inferior extended selection nodes need not be further expanded and can be ignored.*

This is intuitively clear: Consider two schedules S_1 for z with time profile P and S_2 for z with time profile Q. Assume that (z, P) is time-inferior to (z, Q). All schedules generated in later scheduling stages that have S_1 as prefix will have a higher execution time than schedules with the same suffix that have S_2 as prefix, because empty delay slots within S_1 or S_2 cannot be filled with subsequent instructions.

Lemma 2.16 allows the algorithm to prune irrelevant paths as early as possible.

When optimizing execution time and register space simultaneously, inferiority of a selection node must take the register need into account as well. Pruning (z, P) is then only safe if P is time-inferior *and* space-inferior.

Figure 2.22 shows a run of the extended *ncc* algorithm for a single-issue architecture with two units, as in the example above. Only the execution time is optimized. We see that the number of generated extended selection nodes remains reasonably small in this case.

The generalization of our method for extended selection nodes, as described in this section, is very new and has not yet been implemented; this is left for future work.

2.4.7 A Randomized Heuristic

As a byproduct of the optimization algorithms considered so far, we can easily derive a randomized heuristic, in the same way as we did for the contiguous schedules in Section 2.3.2.

A ***random schedule*** is obtained by selecting, in each step of the basic topological search mechanism, one of the nodes from the zeroindegree wavefront randomly. As usual, its register need and execution time can be determined in time $O(n)$. In the random scheduling heuristic ***ncrand***, we repeat this process k times, for a user-specified constant k, and return the best of the schedules encountered. If k is large enough, the quality of the computed schedule is in most cases quite close to the optimum, although the computation also for very large DAGs and k in the order of some 1000 takes only a few seconds on an ordinary workstation.

For large k, *ncrand* may be parallelized in a straightforward way by assigning the computation of k/p random schedules to each of p available processors, and determining the best schedule encountered by a global minimum reduction over integer pairs consisting of register need and execution time.

As far as time optimization is desired, we can, unfortunately, compare with the optimal solution computed by *ncn* only for rather small DAGs (up to about 22 instructions). On the other hand, for very small DAGs (with $n < 12$) *ncv* is even faster than the heuristic with $k = 4096$.

Tables 2.8 and 2.9 show the results produced by *ncrand* for randomly generated DAGs for two types of target processors, with different optimization priorities. We have focused on DAGs with up to 25 nodes because otherwise *ncn* runs out of space and time, and we could then not tell how far away the reported solutions are from the optimum.

Tables 2.10 and 2.11 show the results delivered by *ncrand* for some DAGs taken from scientific application programs, again with different optimization priorities. Some results for the same DAGs with a target processor with multiple functional units are given in Tables 2.12 and 2.13.

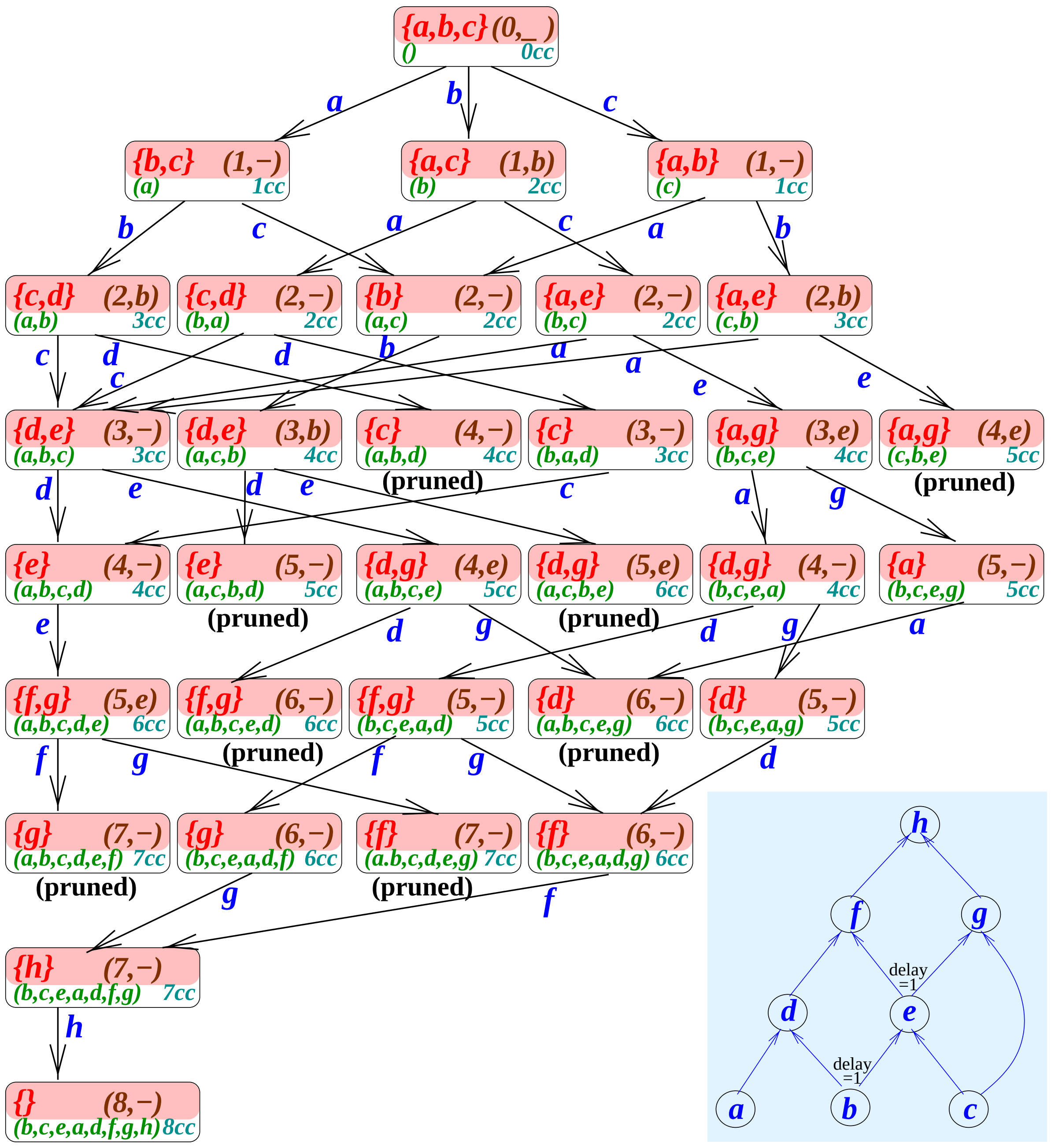


FIGURE 2.22: The extended selection DAG constructed by the generalization of *ncc* for extended selection nodes if applied to the example DAG in the bottom right corner (instructions b and e are delayed by one cycle). The schedule stored in each extended selection node and its execution time are also shown.

DAG size	register need,#nops of best random schedule, $k = 1$	4	16	64	256	1024	4096	optimum (by *ncn*)
15	7,5	6,4	6,4	6,3	**5,3**	5,3	5,3	5,3
16	7,3	7,2	6,3	6,2	**5**,6	5,3	5,2	5,1
18	**5**,3	5,3	5,3	5,3	5,3	**5,1**	5,1	5,1
18	7,4	7,3	6,3	6,3	6,1	**5**,5	**5,4**	5,4
19	7,2	7,2	7,1	6,0	**5**,4	5,3	**5,1**	5,1
19	8,4	**6**,5	6,5	6,3	6,3	6,2	6,2	6,1
20	9,4	8,2	8,2	**7**,3	7,3	7,2	**7,1**	7,1
20	7,3	7,3	7,3	7,2	6,3	6,1	6,1	5,2
22	9,4	9,4	8,3	**7**,3	7,2	7,2	**7**,2	7,1
22	8,3	7,4	7,2	7,2	7,1	**6**,6	6,6	6,5
22	7,4	7,4	7,3	7,1	7,0	**6**,2	6,2	6,0
24	9,5	9,2	8,3	8,3	**7**,4	7,2	**7,0**	7,0
24	11,4	10,5	6,6	6,6	6,6	6,6	6,3	—

TABLE 2.8: Optimization of register need before execution time by *ncrand*, for randomly generated DAGs. The target is a RISC processor with delayed load with latency 1.

DAG size	#nops,register need of best random schedule, $k = 1$	4	16	64	256	1024	4096	optimum (by *ncn*)
12	6,7	2,7	2,6	1,7	1,7	1,7	**0,7**	0,7
12	6,8	2,10	2,9	1,9	1,9	**0**,9	0,9	0,6
13	6,7	5,6	3,7	3,7	**2**,7	2,7	2,6	2,5
14	4,6	4,6	3,8	**1**,8	1,7	1,7	**1,6**	1,6
15	5,7	3,8	**1**,8	1,8	1,8	1,8	1,7	1,6
16	4,8	3,8	2,8	2,8	2,8	**1**,8	1,8	1,6
17	7,7	3,7	1,8	**0**,8	0,8	**0,7**	0,7	0,7
19	2,9	2,9	1,9	1,9	**0**,8	0,8	0,8	0,7
20	8,7	7,9	4,9	3,9	**2**,8	2,8	2,8	2,7
21	2,11	2,11	2,10	2,10	2,9	**0**,9	0,9	0,7
21	5,8	5,8	5,8	3,8	1,8	1,8	1,8	0,7
22	6,9	6,9	2,9	2,8	1,8	1,8	**0**,9	0,6
22	10,10	6,9	4,10	2,10	2,10	1,10	**0**,10	0,8
22	7,8	3,7	3,7	2,8	2,8	**1**,7	1,7	1,6
23	5,9	5,9	2,11	1,11	1,9	**0**,9	0,9	0,7
24	6,11	5,11	5,10	2,10	2,10	1,9	1,9	—
25	5,10	5,10	3,11	2,10	2,9	**1**,10	1,10	1,8
26	6,12	3,12	3,12	1,12	1,12	1,11	1,11	—
27	5,12	5,11	2,11	1,11	1,11	1,10	1,10	0,8

TABLE 2.9: Optimization of execution time before register need by *ncrand*, for randomly generated DAGs. The target is a RISC processor with delayed compute instructions with latency 0 or 1 (determined randomly by coin flipping) and delayed load with latency 2.

We can see from these data that in many cases the best of up to $k = 4096$ random schedules is already optimal with respect to the first-priority optimization goal, while in most cases the heuristic fails to report a solution that is optimal with respect to both the first-priority and the second-priority optimization goal.

2.4.8 Heuristic Pruning of the Selection DAG

The disadvantage of the randomized method just presented is that it is completely unaware of locally optimal partial solutions, as e.g. explored in the algorithms *descend2* or *ncv*. If the DAG is very large, the probability of encountering an optimal schedule by chance is very small if only a fixed number of schedules is generated and tested. On the other hand, the

DAG		register need,#nops of best random schedule, with							optimum
name	size	$k = 1$	4	16	64	256	1024	4096	(by *ncn*)
LL14	19	9,8	8,6	8,6	7,10	7,5	**6**,5	6,5	6,1
LL20	23	9,9	9,8	8,7	8,6	8,5	8,5	7,8	—
INTRAF	26	10,7	10,6	9,7	8,5	7,9	7,8	7,8	—
SPEC77	49	12,10	12,10	12,10	12,10	12,6	12,6	12,4	—
SPHER	27	8,9	8,9	8,5	7,10	7,10	7,10	7,5	—

TABLE 2.10: Optimization of register need before execution time by *ncrand*, for DAGs taken from some scientific codes. The target is a RISC processor with delayed load with latency 1.

DAG		#nops,register need of best random schedule, with							optimum
name	size	$k = 1$	4	16	64	256	1024	4096	(by *ncn*)
LL14	19	8,9	6,8	6,8	5,8	4,8	4,8	3,8	1,6
LL20	23	9,9	5,10	5,9	4,9	4,9	4,9	3,9	—
INTRAF	26	7,10	6,10	4,10	4,10	2,10	2,9	1,9	—
SPEC77	49	10,12	5,13	4,13	4,13	4,13	2,14	2,13	—
SPHER	27	9,8	7,9	4,9	2,9	2,9	1,9	1,9	—

TABLE 2.11: Optimization of execution time before register need by *ncrand*, for DAGs taken from some scientific codes. The target is a pipelined RISC processor with delayed compute instructions with latency 0 or 1 (determined randomly by coin flipping) and delayed load with latency 2.

optimization algorithm *ncv* turned out to be impractical for medium-sized and large DAGs (with more than 22 or 23 nodes).

We propose a heuristic which is a compromise between these two extremal approaches. This heuristic is based on a divide-and-conquer approach.

The given DAG G is split "horizontally" into two subDAGs G_1 and G_2 of approximately half the size of G (see Figure 2.23. Each subDAG is scheduled separately. The lower subDAG G_1 is handled first, such that the list schedule S_2 being computed for the upper subDAG G_2 is

DAG		register need,time of best random schedule, with							optimum
name	size	$k = 1$	4	16	64	256	1024	4096	(by *ncn*)
LL14	19	8,23	6,24	6,24	6,21	5,23	5,23	**4,24**	4,24
LL20	23	9,25	8,24	8,23	7,26	7,24	7,23	6,28	—
INTRAF	26	9,27	7,29	7,29	7,27	6,25	6,25	6,21	—
SPEC77	49	13,47	12,39	12,39	12,39	11,44	11,44	11,37	—
SPHER	27	8,27	8,24	7,27	6,25	6,25	6,25	6,25	—

TABLE 2.12: Optimization of register need before execution time by *ncrand*, for DAGs taken from some scientific codes. The target is a superscalar processor with three different functional units, where the delays are 2 for load, 0 and 1 for arithmetics. For LL14, *ncn* ran about twice as long as the heuristic algorithm with 4096 random schedules (4 seconds on a SUN SPARC-10).

DAG		time,register need of best random schedule, with							optimum
name	size	$k = 1$	4	16	64	256	1024	4096	(by *ncn*)
LL14	19	23,8	19,7	19,7	19,7	19,7	18,6	18,6	17,6
LL20	23	25,9	24,8	23,8	23,8	22,8	22,8	21,7	—
INTRAF	26	27,9	27,9	22,9	22,9	21,9	21,8	20,9	—
SPEC77	49	47,13	39,12	39,12	39,12	38,13	36,13	36,12	—
SPHER	27	27,8	24,8	23,8	23,8	22,9	21,9	21,8	—

TABLE 2.13: Optimization of execution time before register need, for DAGs taken from some scientific codes. The target is a superscalar processor with three functional units: delayed compute instructions with latency 0 and 1, and delayed load with latency 2.

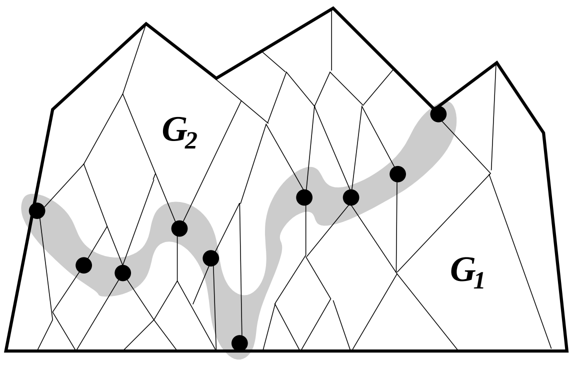


FIGURE 2.23: Splitting a DAG into two subDAGs along a bisector defined by a zeroindegree set b.

just appended to the best schedule S_1 found for the lower subDAG G_1, resulting in a merged schedule S for G. Hence, the time and space requirements at the merge point of the two subschedules are properly taken into account.

The splitting and merging method can be applied recursively to each subDAG until the resulting subDAGs reach a handy size that can be managed by *ncv*. A similar recursive DAG splitting approach has been used by Paul et al. [PTC77] to derive an upper bound on the minimum register need.

The bisector between the two subDAGs must be some zeroindegree set b generated by a top-sort traversal of the DAG. If register space is the first optimization criterion, a zeroindegree set b with small *alive*(b) set is preferable because the size of *alive*(b) is a lower bound for the register need of the overall schedule. On the other hand, a small zeroindegree set may not necessarily be the best choice as a bisector with respect to the execution time of the resulting schedule. Instead, a larger zeroindegree set may allow to keep more functional units busy at the merge point of S_1 and S_2. Also, a small execution time of S_1 will probably lead to a rather small total execution time. We conclude that the quality of b strongly depends on the primary optimization goal desired by the user. It is exactly this heuristic choice of b which may cause the overall algorithm to return a suboptimal solution. In other words, if we knew an optimal bisector in advance (i.e., a zeroindegree set appearing in the middle of an optimal schedule that we cannot compute because the DAG is too large), this divide-and-conquer method would produce an optimal solution as well.

Splitting the DAG G with bisector b and applying *ncv* resp. *ncn* recursively to the subDAGs corresponds to an aggressive pruning of the original selection DAG D: At level $k_b =$

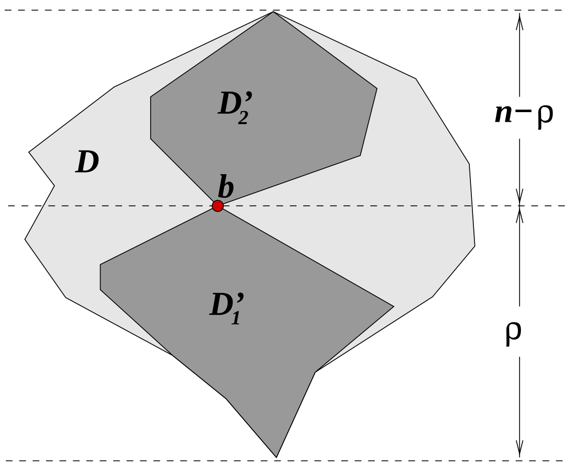


FIGURE 2.24: Bisecting a DAG G with bisector b prunes the selection DAG D for G such that it gets the shape of an hourglass with b as articulation point.

$|\mathit{scheduled}(b)|$ there is only one single selection node, namely for the zeroindegree set b. The pruned selection DAG D' contains only those nodes of D that are in the lower cone or in the upper cone of b in D:

$$D' = (Z', E') \text{ with } Z' = \mathit{cone}(D, b) \cup \mathit{uppercone}(D, b), \quad E' = E \cap (Z' \times Z')$$

The resulting selection DAG has thus the shape of an hourglass, where b is the articulation point (see Figure 2.24). The lower part D'_1 of the pruned selection DAG D' is induced by $\mathit{cone}(D, b)$ and the upper part D'_2 by $\mathit{uppercone}(D, b)$.

In our implementation of this divide-and-conquer approach, we split the DAG implicitly in several horizontal subDAGs with a fixed size σ for each subDAG, based on the pruning mechanism mentioned above applied to the *ncn* implementation. [13] It appears that the space and time consumption of the modified algorithm, now called *ncsplit*, strongly depends on σ but also on the structure of the DAG being processed. For instance, for the 49-node DAG in SPEC77 we obtain a fairly good schedule in only 6 seconds for $\sigma = 4$, but already for $\sigma = 6$ our implementation runs out of space, where the problem occurs already in the bottommost subDAG. The explanation for this behaviour is that larger DAGs have more leaves and hence yield many selection nodes in the first σ levels. Nevertheless, we found that solving the space problem by splitting the DAG unevenly (according to the expected load), by starting with a small σ value and stepwise increasing it while working upwards in the DAG, does not produce good schedules on the average. On the other hand, a larger value of σ does not generally imply a better result or a slower optimization, because the DAG is split differently. Table 2.14 shows some results.

Comparing the results delivered and optimization time and space consumed by *ncsplit* and by *ncrand*, we find that *ncrand* with suitably large k produces results of similar quality as *ncsplit*, but is more economic and predictable with respect to optimization time and space. On the other hand, for large DAGs a run of *ncsplit* even with a very small σ value appears to be more successful than many random schedules.

[13] In contrast to the explicit splitting described above, we simply throw, after reaching a pruning level σ, all selection nodes of that level away except for an optimal one (with regard to the primary optimization goal).

DAG		time,register need : *ncsplit* optimization time , with					optimum
name	size	$\sigma = 6$	$\sigma = 8$	$\sigma = 10$	$\sigma = 12$	$\sigma = 16$	(by *ncn*)
LL14	19	18,5 : 3s	17,7 : 9s	**17,6** : 18s	17,7 : 13s	17,6 : 13s	17,6 : 23s
LL20	23	22,8 : 9s	20,8 : 90s	20,7 : 183s	—	—	—
INTRAF	26	20,10 : 1s	18,7 : 37s	17,8 : 94s	17,8 : 14s	17,9 : 60s	—
SPEC77	49	34,13 : 6s *	—	—	—	—	—
SPHER	27	22,8 : 4s	20,9 : 10s	20,8 : 17s	20,10 : 31s	21,7 : 54s	—
random	30	26,9 : 14s	26,10 : 77s	22,10 : 683s	—	—	—

TABLE 2.14: Results and optimization times of the *ncsplit* algorithm, optimizing execution time as primary and register need as secondary goal, applied to some DAGs taken from scientific codes, for a superscalar target processor with three functional units, where load has 2 delay slots and arithmetics has zero or one delay slot. * The only run for the SPEC77 DAG that did not run out of space used $\sigma = 4$. The best random schedule for the random DAG with 30 nodes, needing 25 cycles and 10 registers, was computed by *ncrand* in 12s (with $k = 4096$).

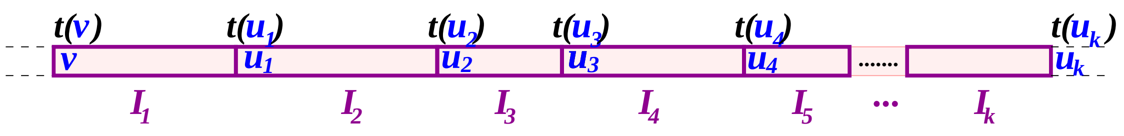


FIGURE 2.25: The live range of a virtual register with k uses can be structured into k intervals.

2.5 Extension for Optimal Scheduling of Spill Code

As each instruction in a basic block produces at most one result value, a DAG node corresponds to a virtual register containing this result value. The virtual register is set once when the instruction is executed, and used as many times as there are DAG nodes that reference it. The time schedule determines for each DAG node v the time slot $t(v)$ where its virtual register will be set, and the order and the time slots $t(u_1)$,...,$t(u_k)$ of the k parent nodes $u_1, ..., u_k$ of v that use v, if there are any. For simplicity of presentation, we number the parent nodes u_j as they appear in the schedule, such that $t_0 < t(u_1) < ... < t(u_k)$. Hence, the ***live range*** of v is the interval $[t(v), t(u_k)[$ which can be partitioned in k intervals $I_1 = [t(v), t(u_1)[$, ...,$I_k = [t(u_{k-1}), t(u_k)[$, as shown in Figure 2.25.

Graph coloring methods for register allocation build a *register interference graph*, where the nodes are the live ranges of the virtual registers, and an edge connects two live ranges if they overlap in time. If the interference graph can be colored with c colors such that no two live ranges connected by an edge have the same color, then c physical registers (corresponding to the different colors) are sufficient to store the virtual registers of the program. In some cases, register allocation can be used to avoid unnecessary `move` instructions by assigning the source and the target virtual register of the `move` to the same physical register (*register coalescing* [CAC+81, Cha82, GA96]).

The classical graph coloring approach to register allocation applied ***spilling*** to the entire live range of a virtual register: This means that it is keet permanently in the main memory instead of in a register; it is stored (more or less) immediately after the definition point $t(v)$ and reload (more or less) immediately before each use $t(u_j)$ (which again requires an available register for a very short time if the target machine does not offer instructions with direct addressing of operands residing in memory, which is the common case today.) and these load and store instructions must be inserted into the existing schedule. Spilling heuristics like [CAC$^+$81, Cha82, CH84, BGM$^+$89] classify live ranges by a cost estimate that considers the execution frequency (frequent reloading is less acceptable in inner loops where instructions are frequently executed) and the degree of the live range in the register interference graph (with how many other live ranges it overlaps) which may be related to its length $t(u_k) - t(v)$. The ideal spilling candidate has a low execution frequency and a high degree in the interference graph. After a spill has been executed, the corresponding store and load instructions are generated, and the register interference graph is rebuilt.

Advanced variants of graph coloring [BCT92] do not consider entire live ranges of virtual registers as atomic units for spilling but split them up into smaller parts; in the extreme case we arrive at the single intervals I_j.

In any case, the problem is that a-posteriori scheduling of the spill code may compromise the quality of the given schedule, which would not be the case if the necessary addtional load and store instructions would have been already known before scheduling.

We do not go into further details here, because this problem is not within the focus of this book, but we would like to mention that the computation of *optimal* spill code for basic blocks is possible as a straightforward extension of our algorithms given above. The idea is as follows:

In addition to investigating just all the different possibilities for selecting a node from the zeroindegree list, additional selection nodes corresponding to schedules for variants of the DAG with inserted store and load nodes are also generated. For a DAG node with outdegree k, this will produce 2^k different selection nodes, because there are 2^k different possibilities of spilling or not spilling the k intervals I_1, ...,I_k. An example for $k = 2$ is shown in Figure 2.26.

Hence, a selection node stores not only the zeroindegree list and the array of current indegrees, but also a list of nodes that have been spilled and a list of additional reload nodes that are conceptually added to the zeroindegree list. The test for equality of selection nodes has to be adapted accordingly.

From a heuristic point of view, splitting long live ranges into just two subranges, with a single additional store and reload node per spilled live range, could be sufficient in many cases, as it is up to the subsequent scheduling decisions to keep the uses within each subrange closely together in order to make the two new subranges substantially smaller than the original live range.

Due to the addition of store and reload nodes, the term *list schedule* must be redefined as a sequence containing the n DAG nodes and some store and reload instructions in some topological order. The definition of the levels of the selection DAG must be slightly adapted by introducing intermediate levels that contain the selection nodes with schedules that end with a store or reload operation of a spilled node, while the "classical" levels still contain the selection nodes where an "ordinary" DAG node was selected last. *scheduled*(z) denotes (as before) only the ordinary DAG instructions scheduled so far, while *spillinst*(z) contains the spill-caused store and reload instructions selected since the last "ordinary" DAG node has

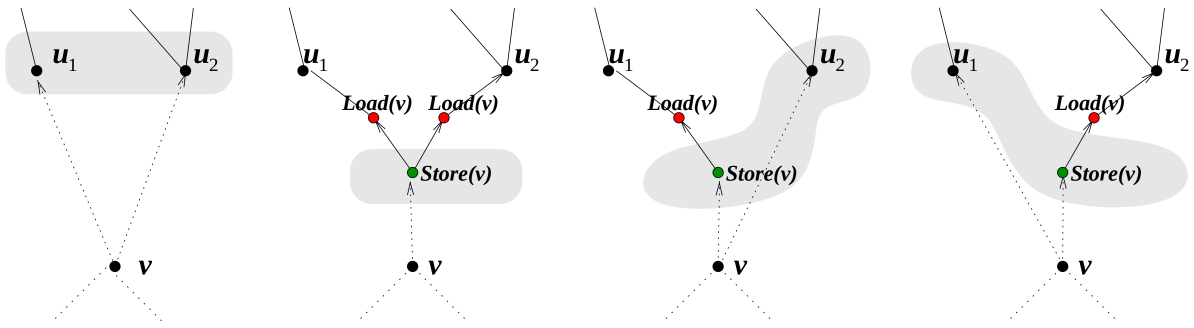


FIGURE 2.26: If all possibilities for spilling single intervals should be considered, for selecting a DAG node with outdegree k from the zeroindegree wavefront 2^k different selection nodes will be created. This is an example for $k = 2$. The newly generated nodes and edges are not part of the DAG but stored implicitly in the corresponding selection node. The shaded area contains the new zeroindegree nodes. The first variant is the usual one generated by standard *ncv*; no interval is spilled. The second one means that both intervals I_1 and I_2 are spilled; hence, two reload instructions are necessary. In the third and fourth variant, only one interval is spilled, while the other one is not, hence only one reload instruction must be inserted. The store instruction is not necessary for leaves. Note that the newly introduced edges from Store to Reload nodes specify precedence constraints caused by data flow in the main memory rather than in the registers, and must thus be handled differently by the register allocation function *get_reg*.

been selected in *scheduled*(z).

Spilling increases thus the length and (maybe) the execution time of a schedule, while a possible gain in reduced register need will become visible only later when scheduling the parent nodes; hence the selection nodes corresponding to spillings will be considered in *ncv* and *ncn* later than the “classical” selection nodes that contain no spillings. This delay of the combinatorial explosion due to the added possibilities is highly desirable, because the spilling variants should only be taken into consideration if the finally determined optimum schedule does exceed the number of available registers.

The effect of postponing the spilling variants may be enforced if another dimension is added to structure the domain of selection nodes, namely the size of *spillinst*(z), the number of spill-caused store and reload instructions selected since the last “ordinary” DAG node has been selected. This solves also the problem of how to organize the intermediate levels of the selection DAG. The “classical” selection nodes z with $|$*scheduled*$(z) = k|$ are entered in the zero-spill-instructions entry $(k, 0)$ of the new two-dimensional level structure. Selection nodes created by selecting a spill-caused instruction at level (k, s) are now entered in level $(k, s + 1)$, while a selection node created by selecting an ordinary instruction is added to level $(k + 1, 0)$. The other two axes for register need and execution time can again be used for a further structuring of the solution space, as in the original algorithm.

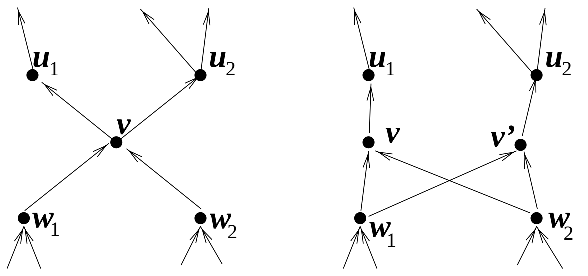


FIGURE 2.27: Recomputing for a live range interval of a single DAG node v corresponds to a conceptual replication of v.

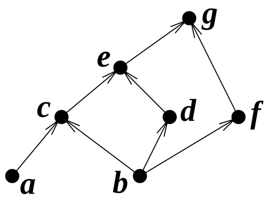


FIGURE 2.28: By recomputing (or spilling) b after its second use, the register need is reduced from 3 to 2 registers if the nodes of this DAG are scheduled in the order a, b, c, d, e, b, f, g.

2.6 Extension for Partial Recomputations

2.6.1 Recomputing Single DAG Nodes

Before discussing the general problem of recomputing entire sets of DAG nodes, we have a look at the simpler problem of recomputing a single DAG node v where the children of v (if there are any) are not being recomputed. Obviously, recomputing v makes only sense if the outdegree k of v is larger than 1. Recomputing v after its ith use, $1 \leq i < k$, means that the children of v cannot free their registers even if v is their last use. Instead, their values must remain in their registers until v has been recomputed for the last time. As with spilling, recomputation refers to the live range intervals of DAG nodes with more than one parent. The recomputation is equivalent to a restructuring of the DAG such that the node to be recomputed is duplicated (see Figure 2.27).

Recomputing a single DAG node v can be, at some extent, regarded as a generalization of spilling v, where the repetition of the instruction v corresponds to the reload instruction while the equivalent of the store instruction is the fact that the registers of nodes whose parents are recomputed cannot be freed before the last recomputation of these nodes. Recomputing and spilling of a leaf node has the same effect, as can be seen from the example in Figure 2.28.

With the described extension for generating optimal spill code, we get also a mechanism for integrating partial recomputation of single DAG nodes. The idea is again that the conceptual introduction of new DAG nodes and edges to account for the recomputation is coded internally in the selection nodes.

2.6.2 Recomputing for Sets of Instructions

Nevertheless, the technical integration of recomputing for multiple nodes as an entity in our framework is harder, because the leveling structure of the selection DAG is now compromised, as either zero-indegree sets are no longer unique, or the selection DAG becomes cyclic. Moreover, recomputing may be nested if, in a set of nodes to be recomputed, one of these nodes is selected again for recomputation. This nesting of recomputation makes the technical representation of the current status of the DAG (indegrees, conceptually replicated nodes and edges) difficult, and will additionally blow up the number of alternatives for scheduling. It is not only this combinatorial explosion that caused us to refrain from further developing this extended form of recomputing. We are also in doubt whether recomputing of larger sets of instructions will be of any use to remarkably improve the register need if the considerable increase in execution time (which is exponential in the worst case [PT78]) is taken into account. If the memory latency is not extraordinarily high, spilling is probably the better alternative to reduce the register need.

2.7 Related Work

2.7.1 Approaches to Optimal Instruction Scheduling

There exist several polynomial-time algorithms for the optimal solution of restricted scheduling problems, such as scheduling trees and / or for very specific target architectures. Surveys of early work on scheduling for pipelined processors are given in Lawler et al. [LLM$^+$87] and Krishnamurthy [Kri90].

For the general problem formulation, one may adopt branch-and-bound methods based on list scheduling, as we did in our first approach (*nce*), and exploit structural properties of the DAG to prune the search space. An alternative consists in modeling the desired scheduling problem as an integer linear program. Both strategies take exponential time in the worst case.

Optimal scheduling for trees Hu [Hu61] developed an algorithm that determines a time-optimal schedule of a tree for a k-issue architectures with k identical units. For each node of the tree, its *level* denotes its distance from the root of the tree. A k-way topological sort is now performed, where as many as possible (up to k) instructions are taken simultaneously from the current zeroindegree set and issued together in the same cycle. If there are more than k instructions in the zeroindegree set, the level values are used as priorities, i.e. k instructions with higher level are selected first. If there are $k' < k$ instructions in the current zeroindegree set, $k - k'$ units will be idle in that cycle.

For processors with two functional units, one for arithmetics and one for loads, Bernstein, Jaffe, and Rodeh [BJR89] propose a dynamic programming algorithm based on [AJ76] for approximatively solving the MTIS problem for trees. Their algorithm runs in time $O(n \log n)$ and produces a schedule that is within a factor of $\min(1.091, 1 + (2 \log n)/n)$ off the optimal execution time. Register need is not considered.

For pipelined RISC processors with a delayed load of 1 delay cycle, Proebsting and Fraser [PF91] solve the MTIS problem for trees where loads (and only loads) occur only at the tree leaves. Spill code is generated if necessary. The algorithm can be used as a heuristic for processors with delays greater than 1. An extension of [PF91] for the case where leaves may contain values already residing in registers has been proposed by Venugopal and Srikant [VS95]. They give an approximation algorithm that delivers a schedule that takes at most one more cycle and one more register than the optimum schedule. For the same processor type, Kurlander, Proebsting, and Fraser [KPF95] give a variation of the Sethi–Ullman labeling algorithm [SU70]. Their algorithm runs in linear time and produces optimal results (both for register need and execution time) for trees. It can be used as a heuristic for DAGs. Spill code is generated where the number of available registers is exceeded.

Optimal scheduling for DAGs Vegdahl [Veg92] applies an idea similar to *ncc* for combining all schedules of the same subset of nodes, to solve the MTIS problem. He first constructs the entire selection DAG, which is not leveled in his approach, and then applies a shortest path algorithm. Obviously this method is, in practice, applicable only to small DAGs. In contrast, we compute also partial schedules with partial costs (space and time) and thus construct only those nodes of the selection DAG which may belong to an optimal path (schedule). Vegdahl's method directly generalizes to Software Pipelining in the presence of loops. However, register requirements are not considered at all. But in particular for the register space axis, our pruning strategy appears to be most successful (see the run times of *ncv* compared to these of *ncn*).

Chou and Chung [CC95] consider the MTIS problem for a superscalar target processor. They construct and traverse a solution space that corresponds to our selection *tree*. In contrast to our *nce* formulation, they do not use list schedules as an intermediate representation of schedules, but proceed processor-cycle-wise, selecting in each cycle as many instructions as possible in a greedy manner. Multiple possibilities for selection occur as soon as there are for a class of instructions more selectible instructions than there are units available for them. In these situations, all possibilities are considered. Several relations that are derived from the DAG structure are used for pruning the selection tree. They report results for random DAGs with up to 20 nodes.

For pipelined RISC processors with a maximum delay of one cycle, Bernstein and Gertner [BG89] give an algorithm for solving the MTIS problem for DAGs. It is based on a reduction of the problem to scheduling tasks on two parallel processors which is solved optimally by the Coffman–Graham algorithm [CG72] in time $O(n\alpha(n))$ [Set76] where α is the inverse of the Ackermann function, an extremely slowly growing function that is smaller than five for all practical applications. Spill code is not generated. The problem for delays greater than one is NP-complete [HG83].

Yang, Wang, and Lee [YWL89] use a branch-and-bound method with pruning strategies for the MTIS problem for DAGs and multiple functional units, for the special case that all functional units are identical and all instructions take unit time with no delays. They additionally assume that the number of available units varies over time but are explicitly given.

Scheduling methods based on integer linear programming Zhang [Zha96] provided the first formulation of instruction scheduling as an integer linear programming problem (*SILP*) whose number of variables and number of inequalities is, in the worst case, quadratic in the number of instructions in the basic block, but can usually be reduced substantially by exploiting properties of the DAG structure. Kästner [Käs97] integrates register allocation into the SILP framework and implements a system for instruction scheduling and register allocation for the digital signal processor ADSP-2106x. He compares this approach to another variant [GE92, GE93] of modeling instruction scheduling and register allocation as an integer linear programming problem called OASIC. The OASIC modeling method incurs in the worst case a quadratic number of binary variables and a cubic number of inequalities for the instruction scheduling part and, depending on the DAG structure, an up to exponential number of inequalities for the register allocation part. For both SILP and OASIC modeled scheduling problems, the resulting integer linear program instances are solved using the commercial solver CPLEX. By using approximations, e.g. by rounding, the complexity of solving the integer linear program can be reduced at the cost of a reduction of the quality of the resulting schedule. It appears that for small DAGs an exact solution can be computed (within a few minutes up to a few hours of CPU time). However, the solvability and the solution times depends considerably on the DAG structure; it varies much more with the DAG size than does our own method for optimal instruction scheduling. For instance, one 16-instruction basic block is scheduled within a few seconds, while the optimal solution of a 18-instruction problem must be aborted after 24 hours. Generally the SILP modeling approach appears to be faster in practice than the OASIC approach. An empirical comparison with several heuristics for instruction scheduling based on list scheduling is given by Langenbach [Lan97]. It turns out that approaches based on integer linear programming are only practical for small basic blocks if an optimal solution is to be found.

At about the same time, integer linear programming has also been used by Leupers and Marwedel [LM97] to solve the MTIS problem for multiple-issue DSP processors.

Another approach based on integer linear programming, applied to solve the MRIS problem, has been reported in [GYZ$^+$99], mainly in order to evaluate a heuristic scheduling algorithm.[14]

In a recent paper [WLH00] on optimal local instruction scheduling by integer linear programming, Wilken, Liu, and Heffernan propose several simplifications of the DAG before deriving the ILP formulation.

2.7.2 Heuristics for Instruction Scheduling

Various heuristic methods have been proposed in the literature.

Lawler et al. [LLM$^+$87] generalize Hu's algorithm [Hu61] for time-optimal scheduling of trees to k-issue architectures with k identical pipelined units. They show also that for a DAG and a machine with one or more identical pipelines, any list scheduling heuristic that prioritizes the nodes by their level in the DAG, such as Hu's algorithm [Hu61] if applied to a DAG, produces (in linear time) a schedule whose execution time is within a factor 2 of the optimum. This is a generalization of a result by Coffman and Graham [CG72] on time-optimal scheduling for a 2-issue architecture with two identical processors. Based on the NP-completeness of time-optimal scheduling for k identical machines where $k > 1$ is part of the input [GJ79] and on the fact that scheduling a DAG for a (single-issue) pipeline of length $k + 1$ is at least as hard as scheduling it for a (k-issue) architecture with k identical units [LVW84], they also provide an NP-completeness result for time-optimal scheduling for pipelines of depth $k > 1$. Moreover, they show that time-optimal scheduling for a multi-issue architecture with two typed pipelines (each instruction is to be executed on a uniquely

[14]Although the experimental results in [GYZ$^+$99] do not explicitly mention this, one can conclude that their method, implemented using the commercial solver CPLEX, seems to cope with DAGs of size up to about 20 to 25.

determined pipeline) is NP-complete as well, even if both pipelines are of length one and the DAG consists only of linear chains of instructions.

Palem and Simons [PS90, PS93] give an $O(n^2)$ time algorithm for time-optimal scheduling the instructions of a DAG for a single-issue, two-unit processor where U_1 is not delayed and U_2 is delayed by one cycle. The algorithm can be used as a heuristic for other configurations. Sarkar and Simons [SS96] propose an extension of this algorithm that allows for a better connection of the schedule of a basic block with the schedules of its successor basic blocks in a restricted global scheduling framework. They add the preference to move idle cycles as far as possible toward the end of the schedule within each basic block, without changing its execution time. The algorithm produces time-optimal global schedules for single-issue processors with instructions delayed by at most one cycle, and can be used as a heuristic, otherwise.

Motwani, Palem, Sarkar and Reyen [MPSR95] present an integrated approach to solve the RCMTIS problem for DAGs heuristically in polynomial time. Their method is applicable to pipelined RISC processors and superscalar processors. Spill code is generated and scheduled if necessary.

Another heuristic method for solving the MTIS problem for a superscalar or VLIW processor has been proposed by Natarajan and Schlansker [NS95]. It applies a heuristic top-sort traversal based on a critical path length criterion.

Kiyohara and Gyllenhaal [KG92] solve the RCMTIS problem for superscalar and VLIW processors with arbitrary delays heuristically. They consider the special case of DAGs in unrolled loops. Spill code is generated if necessary.

Goodman and Hsu [GH88] try to break the phase-ordering dependence cycle and solve the RCMTIS problem by a mix of several heuristics and switching back and forth between different optimization goals (MTIS and MRIS) depending on the current register pressure. Freudenberger and Ruttenberg [FR92] extend this approach to trace scheduling for pipelined VLIW processors.

Brasier, Sweany, Beaty and Carr [BSBC95] propose a framework for combining instruction scheduling and register allocation. They consider the register interference graph before instruction scheduling (early register assignment) and the register interference graph after instruction scheduling (late register assignment). The former is generally a subgraph of the latter, as scheduling may add edges due to increased operand lifetimes in a total execution order. Their algorithm adds to the former graph the edges of the difference graph in a heuristically chosen order step by step and observes a cost function which is a combination of execution time and register pressure. When the overall cost gets too high, they fall back to early register assignment.

Silvera, Wang, Gao and Govindarajan [SWGG97] propose a list scheduling heuristic for out-of-order multi-issue superscalar processors with a given size of the instruction lookahead window. The method tries to rearrange the instructions and reuse registers without decreasing the parallelism available to the processor in the instruction lookahead window, compared with the default order of the instructions in the original intermediate representation. For most of their test programs their technique results in improvements of register need or execution time (or both); in some cases, however, the computed schedule needs even more space or time than the default schedule.

Govindarajan, Zhang and Gao [GZG99] et al. [GYZ$^+$99] propose a heuristic algorithm for the MRIS problem that exploits the observation that in node chains in a DAG the register of an operand can be reused for the result and hence such chains can be handled as an entity for register allocation. The idea is to split the DAG in a set of linear chains, assign registers to chains, and then solve for the MTIS problem with a modified list scheduling algorithm. The scheduling algorithm may deadlock for the minimum amount of registers; in that case, it must try again with one more register.

Gibbons and Muchnick [GM86] give a heuristic for the harder problem where *structural hazards* (*pipeline interlocks*) can occur also between data-unrelated instructions if the same functional sub-unit is needed in some pipeline stage at the same time by two instructions. The algorithm is based on a heuristic backwards top-sort traversal of the DAG, starting with the root nodes as zerooutdegree set, and using critical path analysis to minimize the execution time. At each selection step, the nodes ready to be scheduled in the zerooutdegree set are ordered by a heuristically defined priority value that depends on possible interlocks with already scheduled instructions, time behaviour etc.; the node with highest priority is selected. Ertl and Krall [EK92] propose a hybrid list scheduling algorithm that switches back and forth between two heuristics, depending on the currently available amount of instruction-level parallelism. A generative approach for fast detection of pipeline interlocks in superscalar processors for a given program has been described by Proebsting and Fraser [PF94].

Kerns and Eggers [KE93] propose a technique called balanced scheduling, which is an extension of list-scheduling based heuristics like [GM86] for architectures where `Load` instructions may have variable latency. The priority weight for selection of a `Load` instruction is increased by a value that accounts for the number of instructions that may be executed concurrently with that instruction, in order to provide for sufficent padding of subsequent `Load` delay slots. An improvement is given by Lo and Eggers [LE95].

Further scheduling heuristics developed for specific compilers are given e.g. by Warren [War90] for the IBM R-6000 system and Tiemann [Tie89] for the GNU C compiler.

2.8 Summary

We have presented several algorithms for solving the NP-complete optimization problem of computing a schedule of a basic block whose precedence constraints are given in the form of a DAG. The initial goal was to minimize the number of registers used. Later on we also considered to minimize the execution time on the target processor, or a combined criterion of register need and execution time.

We have demonstrated two fundamental ways of traversing the space of possible schedules. The first one restricts the solution space to contiguous schedules, which are generated by depth-first-search traversals of the DAG. We have developed an exhaustive search strategy that prunes the search space to generate only the schedules that are most promising (with regard to register need). In practice, the method can be applied to DAGs with up to 80 nodes. Hence, in nearly all cases that occur in practice, a space-optimal contiguous schedule can be computed in a reasonable time, and a contiguous schedule is sufficient for space optimization. As a byproduct we also obtained an $O(n \log n)$ time randomized heuristic that may be applied for very large DAGs. On the other hand, focusing on contiguous schedules seems to be a too large restriction as soon as delayed instructions or multiple functional units do matter.

The second way considers all schedules as permitted by the DAG edges, which are generated by topological ordering of the DAG nodes. The basic method, *nce*, naively enumerates all topological orderings. It is made more practical by several refinement steps. First, the solution space is constructed bottom–up by a dynamic programming method, where partial solutions for the same subsets of nodes are combined (*ncc*). The search space can be pruned further (*ncv*) by discretizing it according to schedule length, register need, and execution time (*ncn*) of the locally best partial schedule computed so far, and traversing it in a way that is most promising for the desired optimization goal. In order to speed up the optimization of execution time or the combined optimization of execution time and register need, the concept of time profiles and time–space profiles has been introduced as a generalization of the previous algorithms.

With these improvements, our method becomes practically feasible for the first time to solve this classical optimization problem for small and also for medium-sized DAGs. Hence, the algorithms may be used in optimizing compiler passes for important basic blocks. Regarding DAGs of medium size, a timeout should be specified, depending on the DAG size and the importance of this particular DAG for the overall execution time of the program. If the algorithm exceeds this deadline without the solution being completed, one may, for the MRIS problem, still switch to the optimal algorithm for contiguous schedules or to one of the heuristics based on generating and testing random schedules (randomized contiguous scheduling for MRIS, and random scheduling by *ncrand* for MTIS and MRIS). If solving the MRIS problem, we should immediately restrict ourselves to contiguous schedules for DAGs with more than 50 nodes, since it is not very probable that the solution can be completed before the deadline is met.

Fortunately, nearly all the DAGs occurring in real application programs are of small or medium size and can thus be processed by the *ncv* algorithm. Note also that for test compilations, any legal schedule is sufficient, which can be determined in linear time. The optimization algorithms presented here need only be run in the final, optimizing compilation of an application program. Moreover, they need only be applied to the most time-critical parts of a program, e.g. the bodies of innermost loops.

We have also proposed several ideas how to exploit massive parallelism in *ncv*.

Furthermore, we have proposed and examined several heuristics to allow handling also large DAGs.

In principle it is possible to integrate optimal spilling of register values and recomputations of individual DAG nodes into our framework. Nevertheless the straightforward generalization of our method to computing optimal schedules including optimal spill code blows up the time and space requirements dramatically; some heuristic restrictions seem to be unavoidable here.

Future research based upon our enumeration approach may address global scheduling techniques, other than trace scheduling [Fis81], that extend our method beyond basic block boundaries. For instance, it may be interesting to combine our algorithms with software pipelining techniques in the presence of loops, maybe in a way similar to [Veg92]. We expect the concept of time-space profiles to be useful in this context, as it allows to describe precisely the time and register requirements at the basic block boundaries. A global scheduling method could hence switch back and forth between local and global levels of optimization, maybe even in a multi-level hierarchical way, following the control dependence structure of the input program.

The *ncrand* heuristic may be extended by a greedy method like [GM86] that uses a critical path analysis and priority estimations to generate one more list schedule, maybe also in the reverse direction, i.e. starting with the DAG roots as the last nodes in the schedule and working backwards towards the leaves.

Finally, the integration of instruction scheduling and register allocation with the—up to now, separate—instruction selection phase could be an interesting topic of future research.

All these issues will be further developed in a long-term follow-on project starting in January 2001 at the University of Linköping, Sweden. This project, led by the author, will focus on code generation problems for digital signal processors (DSPs), especially for DSPs with an irregular architecture where standard heuristics and the state-of-the-art, phase-decoupled methods produce code of poor quality. The main solution engine will be based upon the time-space profile method introduced in this chapter, extended by further new techniques that allow to cope with non-homogeneous register sets and similar constraints imposed by irregular hardware. A long-range goal of this project is to actually build a retargetable code generation and optimization system for a wide range of standard and DSP processors, which also includes irregular architectures. Another goal of this project is to establish our dynamic programming approach as a powerful antagonist to the approaches based on integer linear programming described in Section 2.7.1.

Acknowledgements

The modified labeling algorithm presented in Sections 2.3.3 to 2.3.5 was jointly developed with Thomas Rauber [C3,C7,J1].

For helpful comments on earlier versions of the description of the dynamic programming algorithms *ncv* and *ncn* [C12,J4], which are given in Section 2.4, the author thanks Anton M. Ertl, Arno Formella, and Helmut Seidl.

Chapter 3

Automatic Comprehension and Parallelization of Sparse Matrix Codes

3.1 Introduction

Matrix computations constitute the core of many scientific numerical programs. A matrix is called *sparse* if so many of its entries are zero that it seems worthwhile to use a more space-efficient data structure to store it than a simple two-dimensional array; otherwise the matrix is called *dense*. Space-efficient data structures for sparse matrices try to store only the nonzero elements. This results in considerable savings in space for the matrix elements and time for operations on them, at the cost of some space and time overhead to keep the data structure consistent. If the spatial arrangement of the nonzero matrix elements (the *sparsity pattern*) is statically known to be regular (e.g., a blocked or band matrix), the matrix is typically stored in a way directly following this sparsity pattern; e.g., each diagonal may be stored as a one-dimensional array.

Irregular sparsity patterns are usually defined by runtime data. Here we have only this case in mind when using the term "sparse matrix". Typical data structures used for the representation of sparse matrices in Fortran77 programs are, beyond a *data array* containing the nonzero elements themselves, several *organizational variables*, such as arrays with suitable row and/or column index information for each data array element. Linked lists are, if at all, simulated by index vectors, as Fortran77 supports no pointers nor structures. C implementations may also use explicit linked list data structures to store the nonzero elements, which supports dynamic insertion and deletion of elements. However, on several architectures, a pointer variable needs more space than an integer index variable. As space is often critical in sparse matrix computations, explicit linked lists occur rather rarely in practice. Also, many numerical C programs are written in a near-Fortran77 style because they were either directly transposed from existing Fortran77 code, or because the programming style is influenced by former Fortran77 projects or Fortran77-based numerics textbooks.

Matrix computations on these data structures are common in practice and often parallelizable. Consequently, numerous parallel algorithms for various parallel architectures have been invented or adapted for sparse matrix computations over the last decades.

Bik and Wijshoff [BW96] suggested that the programmer expresses, in the source code, parallel (sparse) matrix computations in terms of dense matrix data structures, which are more elegant to parallelize and distribute, and let the compiler select a suitable data structure for the matrices automatically. Clearly this is not applicable to (existing) programs that use hardcoded data structures for sparse matrices.

While the problems of automatic parallelization for *dense* matrix computations are, meanwhile, well understood and sufficiently solved, (e.g. [BKK93, ZC90]), these problems have been attacked for *sparse* matrix computations only in a very conservative way, e.g., by runtime parallelization techniques such as the inspector–executor method [MSS$^+$88] or runtime analysis of sparsity patterns for load-balanced array distribution [UZSS96]. This is not astonishing because such code looks quite awful to the compiler, consisting of indirect array indexing or pointer dereferencing which makes exact static access analysis impossible.

In this chapter we describe SPARAMAT, a system for concept comprehension that is particularly suitable to *sparse* matrix codes. We started the SPARAMAT project by studying several representative source codes for implementations of basic linear algebra operations like dot product, matrix–vector multiplication, matrix–matrix multiplication, or LU factorization for sparse matrices [Duf77, Gri84, Kun88, SG89, Saa94, ZWS81] and recorded a list of basic computational kernels for sparse matrix computations, together with their frequently occurring syntactical and algorithmic variations.

3.1.1 Basic Terminology

A ***concept*** is an abstraction of an externally defined procedure. It represents the (generally infinite) set of concrete procedures coded in a given programming language that have the same type and that we consider to be equivalent in all occurring calling contexts. Typically we give a concept a *name* that we associate with the type and the operation that we consider to be implemented by these procedures.

An ***idiom*** of a concept c is such a concrete procedure, coded in a specific programming language, that has the same type as c and that we consider to implement the operation of c.

An ***occurrence*** of an idiom i of a concept c (or short: an occurrence of c) in a given source program is a fragment of the source program that matches this idiom i by unification of program variables with the procedure parameters of i. Thus it is legal to replace this fragment by a call to c, where the program objects are bound to the formal parameters of c. The (compiler) data structure representing this call is called an ***instance*** I of c; the formal parameters, that is, the fields in I that hold the program objects passed as *parameters* to c are called the ***slots*** of I. Beyond the Fortran77 parameter passing, SPARAMAT allows procedure-valued parameters as well as higherdimensional and composite data structures to occur as slot entries.

After suitable preprocessing transformations (inlining all procedures) and normalizations (constant propagation), the intermediate program representation—abstract syntax tree and/or program dependence graph—is submitted to the concept recognizer. The concept recognizer, described in Section 3.4, identifies code fragments as concept occurrences and annotates them by concept instances.

When applied to parallelization, we are primarily interested in recognizing concepts for which there are particular parallel routines available, tailored to the target machine. In the back-end phase, the concept instances can be replaced by suitable parallel implementations. The information derived in the recognition phase also supports automatic data layout and performance prediction [D2,J2].

3.1.2 Problems in Program Comprehension for Sparse Matrix Computations

One problem we were faced with is that there is no standard data structure to store a sparse matrix. Rather, there is a set of about 15 competing formats in use that vary in their advantages and disadvantages, in comparison to the two-dimensional array which is the “natural” storage scheme for a dense matrix.

The other main difference is that space-efficient data structures for sparse matrices use either indirect array references or (if available) pointer data structures. Thus the array access information required for safe concept recognition and code replacement is no longer completely available at compile time. Regarding program comprehension, this means that it is no longer sufficient to consider only the declaration of the matrix and the code of the computation itself, in order to safely determine the semantics of the computation. Code can only be recognized as an occurrence of, say, sparse matrix–vector multiplication, subject to the condition that the data structures occurring in the code really implement a sparse matrix. As it is generally not possible to statically evaluate this condition, a concept recognition engine can only *suspect*, based on its observations of the code while tracking the live ranges of program objects, that a certain set of program objects implements a sparse matrix; the final *proof* of this hypothesis must either be supplied by the user in an interactive program understanding framework, or equivalent runtime tests must be generated by the code generator. Unfortunately, such runtime tests, even if parallelizable, incur some overhead. Nevertheless, static program flow analysis (see Section 3.4.5) can substantially support such a *speculative* comprehension and parallelization. Only at program points where insufficient static information is available, runtime tests or user prompting is required to confirm (or reject) the speculative comprehension.

3.1.3 Application Areas

The expected benefit from successful recognition is large. For automatic parallelization, the back-end should generate two variants of parallel code for the recognized program fragments: (1) an optimized parallel library routine that is executed speculatively, and (2) a conservative parallelization, maybe using the inspector–executor technique [MSS$^+$88], or just sequential code, which is executed nonspeculatively. These two code variants may even be executed concurrently and overlapped with the evaluation of runtime tests: If the testing processors find out during execution that the hypothesis allowing speculative execution was wrong, they abort and wait for the sequential variant to complete. Otherwise, they abort the sequential variant and return the computed results. Nevertheless, if the sparsity pattern is static, it may be more profitable to execute the runtime test once at the beginning and then branching to the suitable code variant.

Beyond automatic parallelization, the abstraction from specific data structures for the sparse matrices also supports program maintenance and debugging, and could help with the exchange of one data structure for a sparse matrix against another, more suitable one. For instance, recognized operations on sparse matrices could be replaced by their counterparts on dense matrices, and thus, program comprehension may serve as a front-end to [BW96]. Or, the information derived by concept recognition may just be emitted as mathematical formulas, for instance in LaTeX format, typeset in a mathematical textbook style, and shown in a graphical editor as annotations to the source code, in order to improve human program understanding.

The SPARAMAT implementation focuses on sparse matrix computations coded by indirect array accesses. This is because, in order to maintain an achievable goal in a university project, it is necessary to limit oneself to a language that is rather easy to analyze (FORTRAN), to only a handful of sparse matrix formats (see Section 3.2), and to a limited set of most important concepts (see Section 3.3.1. For this reason, pointer alias analysis of C programs, as well as concepts and matching rules for pointer-based linked list data structures, are beyond the scope of this project. Due to the flexibility of the generative approach, more concepts and templates may be easily added by any SPARAMAT user. Furthermore, it appears that we can reuse some techniques from our earlier PARAMAT project [D2,J2,J6] more straightforwardly for indirect array accesses than for pointer accesses.

3.1.4 Overview of the Chapter

This chapter is structured as follows: Section 3.2 contains a guided tour of common sparse matrix storage schemes. Section 3.3 summarizes concepts for (sparse) matrix computations. Section 3.4 discusses the concept recognition strategy, and Section 3.5 describes our implementation. The concept specification language CSL is presented in Section 3.6. We briefly review related work in Section 3.7, identify possible directions for future work in Section 3.8, and conclude with Section 3.9. Further examples are given in Appendix B.

3.2 Vectors and (Sparse) Matrices

3.2.1 Basic Terminology: Vectors and Matrices

It is important for the following discussion to distinguish between program data structures, such as multidimensional arrays, and mathematical entities that correspond to bulk accesses to these data structures, such as vectors and matrices. In particular, the extent and/or dimensionality of an access may differ from the extent or dimensionality of the data structure being accessed.

A ***vector*** is an object in the intermediate program representation that summarizes a *one-dimensional view of some elements of an array*. For instance, a vector of reals accessing the first 5 elements in column 7 of a two-dimensional array `a` of reals is represented as `V(a,1,5,1,7,7,0)`. For ease of notation we assume that the “elements” of the vector itself are consecutively numbered starting at 1. `IV(...)` denotes integer vectors.

An ***indexed vector*** summarizes a one-dimensional view of some elements of an array whose indices are specified in a second (integer) vector, e.g. `VX(a,IV(x,1,n,2))`.

A ***matrix*** summarizes a *two-dimensional view of an array* according to the conventions of a specific storage format. Dense matrices appear as a special case of sparse matrices.

3.2.2 An Overview of Data Structures for Sparse Matrices

Now we summarize some general storage formats for sparse matrices based on index vectors, which are the most frequently occurring in Fortran77 codes. Formats for special, more regular sparsity patterns, such as for band matrices, block sparse matrices, or skyline matrices, are not considered here. The abbreviations of format names are partially adapted from SPARSKIT [Saa94]. More details can be found, for instance, in the SPARSKIT documentation [Saa94], the TEMPLATES collection of linear algebra algorithms [BBC$^+$94], and in Zlatev's textbook on algorithms on sparse matrices [Zla91].

- **DNS** (*dense storage format*): uses a two-dimensional array `A(N,M)` to store all elements. Due to the symmetric access structure of the two-dimensional array, a *leading dimension* flag `ld` tells us whether the matrix access is transposed or not. In the following notation, we summarize all data referring to the dense matrix access as an object

  ```
  DNS( a, 1,n,1, 1,m,1, ld )
  ```

  Example: In FORTRAN, DNS-matrix–vector multiplication may look like

  ```
  DO i = 1, n
     b(i) = 0.0
     DO j = 1, m
        b(i) = b(i) + a(i,j) * x(j)
     ENDDO
  ENDDO
  ```

- **COO** (*coordinate format*): A data array `a` stores the `nz` nonzero matrix elements in arbitrary order, and integer vectors `row(nz)` and `col(nz)` hold for each nonzero element its row and column index. The object representing the matrix access is summarized as

  ```
  COO( V(a,1,nz,1), IV(row,1,nz,1), IV(col,1,nz,1), nz )
  ```

  Example: COO-Matrix–vector multiplication may look like

  ```
  DO i = 1, n
     b(i) = 0.0
  ENDDO
  DO k = 1, nz
     b(row(k)) = b(row(k)) + a(k) * x(col(k))
  ENDDO
  ```

  The COO format occurs, for instance, in the SLAP package [SG89].

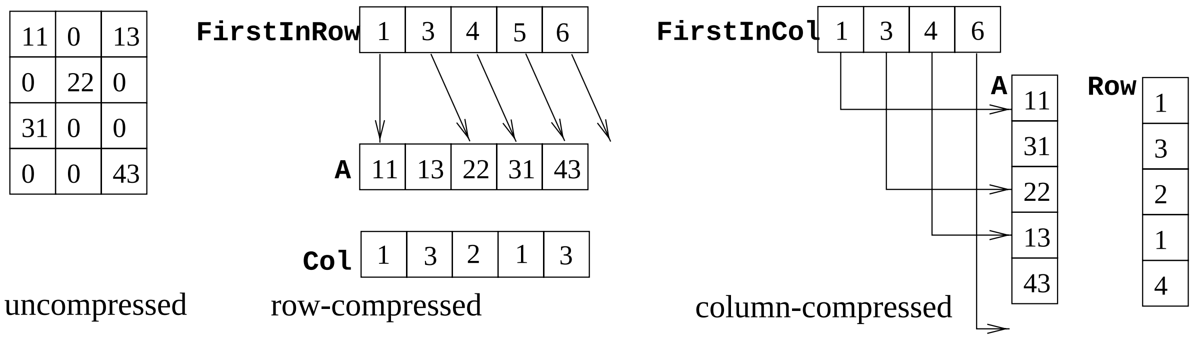


FIGURE 3.1: Row-compressed (CSR) and column-compressed (CSC) storage formats for sparse matrices.

- **CSR** (*row-compressed sorted storage format*): A data array `a` stores the `nz` nonzero matrix elements a_{ij} in row-major order, where within each row the elements appear in the same order as in the dense equivalent. An integer vector `col(1:nz)` gives the column index for each element in `a`, and an integer vector `firstinrow(1:n+1)` gives indices to `a` such that `firstinrow(`i`)` denotes the position in `a` where row i starts, $i = 1,...,$`n` and `firstinrow(n+1)` always contains `nz+1` (see Figure 3.1). Thus, `firstinrow(`$i+1$`)-firstinrow(`i`)` gives the number of nonzero elements in row i. A CSR matrix object is summarized as

```
CSR(V(a,firstinrow(1),firstinrow(n+1)-1,1),
    IV(firstinrow,1,n+1,1),
    IV(col,firstinrow(1),firstinrow(n+1)-1,1),
    n,
    nz)
```

  Example: An idiom of a matrix vector multication for CSR format may look like

```
DO i = 1, n
   b(i) = 0.0
   DO k = firstinrow(i), firstinrow(i+1)-1
      b(i) = b(i) + a(k) * x(col(k))
   ENDDO
ENDDO
```

  Such storage formats are typical for Fortran77 implementations. CSR is used, for instance, in the SLAP package [SG89].

- **CUR** (*row-compressed unsorted storage format*): like CSR, but the order of nonzeros within each row is not important. CUR is used e.g. as the basic format in SPARSKIT [Saa94]. CUR matrix–vector multiplication looks identical to the CSR version.

- **XSR / XUR**: an extension of CSR / CUR that additionally stores in an n-element integer array `lastinrow` for each compressed row its last index within the data array `A`. This makes row interchanges and row reallocations due to fill-in more efficient. XUR is used e.g. in Y12M [ZWS81].

- **MSR** (*modified row-compressed storage format*): like CSR, but the elements of the main diagonal of the matrix are stored separately and regardless of whether they are zero or not. This is motivated by the fact that, often, most of the diagonal elements are a priori known to be nonzero, and are accessed more frequently than the other elements. Typically the diagonal elements are stored in the first `n` elements of `a` and `a(n+1)` is unused. The column indices of the diagonal elements need not be stored, thus the elements of the array `firstinrow` of CSR are stored in the first `n+1` entries of a two-purpose integer array `fircol`. The remaining nonzero elements are stored in `a(n+2:nz+1)` and their column indices in `fircol(n+2:nz+1)`. A MSR matrix object is thus given as

  ```
  MSR(V(a,1,fircol(n+1)-1,1), IV(fircol,1,n+1,1), n, nz)
  ```

  MSR is used, for instance, in the sparse matrix routines of the *Numerical Recipes* [PTVF92].

  Example: Matrix–vector multiplication may look as follows (routine `sprsax()` from [PTVF92]):

  ```
  DO i = 1, n
      b(i) = a(i) * x(i);
      DO k = fircol(i), fircol(i+1)-1
         b(i) = b(i) + a(k) * x(fircol(k))
      ENDDO
  ENDDO
  ```

- **CSC** (*column-compressed format*): similar to CSR where the `a` contains the nonzero elements in column-major order and the other two arrays are defined correspondingly (see Figure 3.1). Thus, CSC format for a matrix A is equivalent to the CSR format for A^T, and vice versa. A CSC matrix object is summarized by

  ```
  CSC( V( a, firstincol(1), firstincol(n+1)-1, 1),
       IV( firstincol, 1, n+1, 1),
       V( row, firstincol(1), firstincol(n+1)-1, 1),
       n, nz)
  ```

  Example: CSC-Matrix–vector multiplication may look like

  ```
  DO i = 1, n
     DO k = firstincol(i), firstincol(i+1)-1
        b(row(k)) = b(row(k)) + a(k) * x(i)
     ENDDO
  ENDDO
  ```

CSC is used, for instance, in the Harwell `MA28` package [Duf77].

- **MSC** (*modified column-compressed storage format*): A MSC matrix object is similar to the CSC representation, but the elements of the main diagonal of the matrix are stored separately, as for MSR.

- **JAD** (*jagged diagonal format*): First the rows of the matrix are permuted to obtain decreasing numbers n_i of nonzero elements for each row i. The data array `a(1:nz)` is filled as follows: The first nonzero element of each row i (the first "jagged diagonal") is stored in `a(`i`)`, the second nonzero element of each row i in `a(n+`i`)` etc. The overall number `njd` of jagged diagonals is at most `n`. An integer array `col(1:nz)` holds the column index of each element in `a`. An integer array `firstinjdiag(1:n)` holds indices into `a` resp. `col` indicating the beginning of a new jagged diagonal; thus `firstinjdiag(`$k+1$`)-firstinjdiag(`k`)` gives the number n_k of elements belonging to the kth jagged diagonal. Thus, a JAD matrix object is given by

```
JAD( V(a,firstinjdiag(1), firstinjdiag(njd+1),1),
     IV(firstinjdiag,1,njd+1,1),
     V(col,firstinjdiag(1),firstinjdiag(njd+1)-1,1),
     n, nz, njd )
```

  Example: JAD-Matrix–vector multiplication may look like

```
DO r = 1, n
   b(r) = 0.0
ENDDO
DO k = 1, njd
   DO i = firstinjdiag(k), firstinjdiag(k+1)-1
      r = i - firstinjdiag(k)
      b(r) = b(r) + a(i) * x(col(i))
   ENDDO
ENDDO
```

  followed by re-permutation of vector `b` if necessary.

- **LNK** (*linked list storage format*): The data array `a(1:maxnz)` holds the `nz` nonzero elements in arbitrary order, the integer array `col(1:maxnz)` gives the column index of each nonzero element. An integer array `nextinrow(1:maxnz)` links the elements belonging to the same row in order of increasing `col` index. A zero `nextinrow` entry marks the last nonzero element in a row. The list head element of each row i is indexed by the ith element of the integer array `firstinrow(1:n)`. Empty rows are denoted by a zero `firstinrow` entry. If required by the application, a similar linking may also be provided in the other dimension, using two more index vectors `nextincol(1:nz)` and `firstincol(1:n)`. Thus, a singly-linked LNK matrix object is summarized by

```
LNK( VX( a, IV(firstinrow,1,n)),
     IV(firstinrow,1,n),  IV(nextinrow,1,n),
     VX( col, IV(firstinrow,1,n)), n, nz, maxnz )
```

Example: LNK-Matrix–vector multiplication may look like

```
DO i = 1, n
   b(i)=0.0
   k = firstinrow(i)
   WHILE (k.GT.0)
      b(i) = b(i) + a(k) * x(col(k))
      k = nextinrow(k)
   ENDWHILE
ENDDO
```

The LNK format requires more space than the previously discussed sparse matrix formats, but it supports efficient dynamic insertion and deletion of elements (provided that `a` and `nextinrow` have been allocated with sufficient space reserve, `maxnz`).

While matrix–vector multiplication codes for a sparse matrix look quite simple and seem to be somehow identifiable by concept matching techniques, implementations of matrix–matrix multiplication or LU decomposition look quite unstructured. This is mainly due to the fact that in the course of these algorithms, some matrix elements may become nonzero which were originally zero (*fill-in*), and thus additional storage has to be allocated for inserting them. Thus, the sparsity pattern may change in each step of these algorithms, while at matrix–vector multiplication, the sparsity pattern (and thus, the organizational variables) is read-only.

A simple work-around to cope with a limited number of fill-ins is to store fill-ins in a separate temporary data structure, or respectively, to allocate slightly more space for the data array and the index vectors. This is e.g. applied in SPARSE [Kun88].

There are also many possibilities for slight modifications and extensions of these data structures. For instance, a flag may indicate symmetry of a matrix. Such changes are quite ad-hoc, and it seems generally not sensible to define a new family of concepts for each such modification. For instance, in the Harwell routines MA30, the sign bit of the row resp. column indices is “misused” to indicate whether a new column or row has just started, thus saving the `firstinrow` resp. `firstincol` array when sequentially scanning through the matrix. Clearly such dirty tricks make program comprehension more difficult.

A main consequence that arises from these data structures is that the comfortable symmetry present in the two-dimensional arrays implementing dense matrices (DNS) is lost. Hence, we must explicitly distinguish between transposed and nontransposed matrix accesses, and between rowwise and columnwise linearization of the storage for the nonzero matrix elements.

Linked list data structures (e.g., the LNK format) cause operations on them, such as traversal or insert/delete, to be inherently sequential. Thus they are particularly good candidates to be completely replaced by other data structures that are more suitable for exploiting parallelism, e.g. linked lists with multiple heads for parallel access. Data structure replacement for a sparse matrix is possible if all operations on it have been recognized and if alias analysis can guarantee that there are no other variables which may be used to access one of these linked list elements in an unforeseen way.

3.3 Concepts

This section gives a survey of concepts that are frequently encountered in sparse matrix codes. Although this list is surely not exhaustive, it should at least illustrate the application domain. The extension of this list by more concepts to cover an even larger part of numerical software may be a subject for future work.

We have developed a concept specification language that allows one to describe concepts and matching rules on a level that is (more or less) independent from a particular source language or compiler. A concept specification consists of the following components: its name (naming conventions are discussed below), an ordered and typed list of its parameters, and a set of matching rules (called *templates*). A matching rule has several fields: a field for structural pattern matching, specified in terms of intermediate representation constructs (loop headers, conditions, assignments, and instances of the corresponding subconcepts), fields specifying auxiliary predicates (e.g., structural properties or dataflow relations), fields for the specification of pre- and postconditions for the slot entries implied by this concept (see Section 3.4), and a field creating a concept instance after successful matching. For an example specification see Figure 3.2. The details of the concept specification language will be described in Section 3.6.

Naming conventions for concepts Our naming conventions for concepts are as follows: The *shape* of operands is denoted by shorthands S (scalar), V (vector), VX (indexed vector), and YYY (matrix in some storage format YYY). The result shape is given first, followed by a mnemonic for the type of computation denoted by the concept, and the shorthands of the operands. Most concepts are type-polymorphic, that is, they are applicable to computations on real or integer data. Where an explicit type specification is necessary, special integer concepts and objects are prefixed with an `I` while their real counterparts are not.

Basic concepts include memory access concepts and constant constructor concepts.

Memory access concepts include all accesses to variables or scalar, vector or matrix accesses to arrays. *Constant constructor concepts* represent scalar-, vector- or matrix-valued compile-time constant expressions.

We extend our earlier PARAMAT approach [D2,J2,J6] to representing concepts and concept instances in several aspects.

Automatic type inference The basic concepts are typed: the integer versions of these concepts are prefixed with an `I`, while the real versions are not.

Nonbasic concepts are not typed. The type (integer or real) of a nonbasic concept instance is automatically inferred from the memory access concepts occurring as concept instance parameters by obvious type inference rules.

```
concept SDOTVV {
  param(out)  $r: real;
  param(none) $L: range;
  param(in)   $u: vector;
  param(in)   $v: vector;
  param(in)   $init: real;

  templateVertical {
    pattern {
       node  DO_STMT $i = $lb:$ub:$st
       child    INCR($rs,MUL($e1,$e2))
    }
    where {
           $e1->isSimpleArrayAccess($i)
           && $e2->isSimpleArrayAccess($i)
           && $s->isVar()
           && $i->notOccurIn($s)
    }
    instance SDOTVV($rs, newRANGE($i,$lb,$ub,$st),
                     newVector($e1,$i,$lb,$ub,$st),
                     newVector($e2,$i,$lb,$ub,$st),
                     $rs)
  }
  templateHorizontal {
    pattern {
      sibling($s) SINIT($x,$c)
         fill($f)
         node($n) SDOTVV($r1,$L1,$u1,$v1,$init1)
    }
    where($s) { $x->array() == $init1->array() }
    where($f) {
      notOutSet = $x;
      notInSet  = $x;
      inSet     = $init1;
    }
    instance($s) EMPTY()
    instance($n) SDOTVV( $L1, $u1, $v1, $r1, $c )
  }
}
```

FIGURE 3.2: A CSL specification for the SDOTVV concept (simple dot product) with two templates.

Operator parameters Some concepts like VMAPVV (elementwise application of a binary operator to two operand vectors) take an operator as a parameter. This makes hierarchical program comprehension slightly more complicated, but greatly reduces the number of different concepts, and allows for a more lean code generation interface.

Functional composition As a generalization of operator parameters, we allow, to some extent, the arbitrary functional composition of concepts to form new concepts. This idea is inspired by the work of Cole on algorithmic skeletons [Col89]. Nevertheless, it turned out to be useful if there exist at least some “flat” concepts for important special cases, such as SDOTVV for dot product, VMATVECMV for matrix–vector multiplication, etc. These may be regarded as “syntactic sugar” but are to be preferred as they enhance readability and speed up the program comprehension process.

No in-place computations Most of our concepts represent not-in-place computations. In general, recognized in-place computations are represented by using temporary variables, vectors, or matrices. This abstracts even further from the particular implementation. It is the job of the back-end to reuse (temporary array) space where possible. In other words, we try to track *values* of objects rather than memory locations. Where it is unavoidable to have accumulating concepts, they can be specified using accumulative basic operations like `INCR` (increment) or `SCAL` (scaling).

Concept instances as parameters Nesting of concept instances is a natural way to represent a treelike computation without having to specify temporary variables. As an example, we may denote a `DAXPY`-like computation, $\vec{b} \leftarrow \vec{b} + 3.14 \cdot \vec{c}$, as

```
VMAPVS( V(tmp,1,n,1), MUL, V(c,1,n,1), VCON(3.14,n) )
VINCRV( V(b,1,n,1), V(tmp,1,n,1) )
```

which is closer to the internal representation in the compiler, or as

```
VINCR( V(b,1,n,1), VMAPVS( , MUL, V(c,1,n,1), VCON(3.14,n) ) )
```

which is more readable for humans. If the computation structure is a directed acyclic graph (DAG), then we may also obtain a DAG of concept instances, using temporary variables and arrays for values used multiple times. In order to support nesting, our notation of concept instances allows to have the result parameter (if there is exactly one) of a concept instance appear as the "return value" of a concept instance, rather than as its first parameter, following the analogy to a call to a function returning a value.

Nesting of concept instances is important to simplify and reduce the number of concepts. Hence, a concept instance may occur as a parameter in a slot of another concept instance, provided that the type of the concept instance matches the slot type.

Without the nesting feature, the addition of multiple sparse matrix formats would have significantly increased the complexity and number of concepts. For instance, each combination of a matrix computation and a format for each of its matrix parameters would result in a different concept. Instead, the format particulars are gathered into a special memory access concept instance and stored as a parameter to the concept instance for the matrix operation.

3.3.1 Some Concepts for Sparse Matrix Computations

We give here an informal description of some concepts. v, v_1, v_2 denote (real or integer) vectors, rv a real vector, a a (real or integer) array, iv an integer vector, m, m_1, m_2 matrices in some format and r a range object. i, i_1,...,i_5 denote integer valued concept instances.

Memory access concepts

- `VAR`$(a, i_1, ..., i_5)$ — access to the real variable a. For scalar real array accesses, up to five index expressions can be specified. If no index expressions are given, a must be a scalar real variable.

- `IVAR`$(a, i_1, ..., i_5)$ — access to the integer variable a. For scalar integer array accesses, up to five index expressions can be specified. If no index expressions are given, a must be a scalar integer variable.
- `V`$(a, l_1, u_1, s_1, ...)$ — real vector access to real array a, where l_j, u_j, s_j hold concept instances for the expressions for the lower bound, upper bound, and stride in dimension j. The number of expressions depends on the dimensionality of a; hence, j varies from 1 to the maximum number of dimensions, which is 5 in FORTRAN.
- `IV`$(a, l_1, u_1, s_1, ...)$ — similar, for integer array array a
- `VX`$(a$,`IV`$(...))$ — real indirect access of real array a, indexed by the integer vector in the second slot.
- `IVX`$(a$,`IV`$(...))$ — integer indirect access of integer array a, indexed by the integer vector in the second slot.
- `STRIP`$(a$,`IVX`$(f_a, iv))$ — compose a vector as concatenation of rows of a CSR sparse matrix with work array a and first-in-row array f_a. The rows are selected according to the row indices given in the integer vector iv. See Appendix B.2 for an example.

In the following presentation, we sometimes omit the formal notation `VAR(a,i)` for array accesses and use instead the sloppy notation `a(i)` to enhance readability.

Constant constructor concepts

- `RANGE`(i, l, u, s) — variable i ranging across an interval defined by lower bound l, upper bound u, and stride s.
- `CON`(c) — real constant c
- `ICON`(c) — integer constant c

  The following concepts are used for lifting scalar expressions to the vector domain:

- `VCON`(r, n) — vector constant of extent n, each element containing the real constant r
- `IVCON`(i, n) — integer vector constant of extent n, each element containing the integer constant i
- `VEXP`(e, n) — real vector of extent n, each element containing the same real-valued scalar expression e
- `IVEXP`(e, n) — integer vector of extent n, each element containing the same integer-valued scalar expression e
- `VTAB`$(e$,`RANGE`$(i, l, u, s))$ — creates a real-vector-valued expression by substituting all occurrences of i in expression e by the sequence of values specified in the range parameter. i must not occur in e as an index of an array access.

Concepts for scalar computations

There are concepts for binary expression operators, like `ADD`, `MUL`, `MAX`, `EQ` etc., for unary expression operators like `NEG` (negation), `ABS` (absolute value), `INV` (reciprocal), `SQR` (squaring) etc., The commutative and associative operators, `ADD`, `MUL`, `MAX` etc., `MIN`, `OR`, `AND` may also have more than two operands. `STAR` is a special version of a multi–operand `ADD` denoting difference stencils [D2,J2]. The increment operators `INCR` (for accumulating addition) and `SCAL` (for accumulating product) are used instead of `ADD` or `MUL` where the result variable is identical to one of the arguments. Assignments to scalars are denoted by the `ASSIGN` concept.

For technical reasons there are some auxiliary concepts like `EMPTY` (no operation).

Vector and matrix computations

- `VMAPVV`$(v, \oplus, v_1, v_2)$ — elementwise application of binary operator $\oplus$, results stored in v
- `VMAPV`$(v, \ominus, v_1)$ — elementwise application of unary operator $\ominus$, results stored in v
- `VMAPVS`$(v, \oplus, v_1, r)$ — elementwise application with a scalar operand r, results stored in v
- `VINCR`(v, v_1) — $v(i) = v(i) + v_1(i),\ i = 1, ..., |v_1|$
- `VASSIGN`(v, v_1) — assign real-vector-valued expression v_1 to v
- `SREDV`$(r, \otimes, v)$ — reduction $r = \bigotimes_{j=1}^{|v|} v(j)$
- `SREDLOCV`$(k, \odot, v)$ — compute some k with $v(k) = \bigodot_{j=1}^{|v|} v(j)$
- `VPREFV`$(v, \otimes, v_1)$ — prefix computation $v(i) = \bigotimes_{j=1}^{i} v_1(j)$, $i = 1, ..., |v_1|$
- `VSUFFV`$(v, \otimes, v_1)$ — suffix computation $v(i) = \bigotimes_{j=|v_1|}^{i} v_1(j)$, $i = |v_1|, ..., 1$

Searching and sorting on a vector

- `SRCH`(k, v_1, r) — compute $k =$ rank of r in v_1
- `VSORT`(v, v_1) — sort v_1 and store the result in v
- `VCOLL`(v, v_1, v_2) — extract all elements $v_2(i)$ where $v_1(i) \neq 0$, with $i = 1...|v_1|$, and store the result vector in v.
- `VSWAP`(v, v_1) — elementwise swap the vectors v and v_1

Elementwise matrix computations

In the following list, m_i for $i = 0, 1, 2, ...$ stands for matrix objects `XXX(...)` in some format XXX.

- `MMAPMM`$(m, \oplus, m_1, m_2)$ — elementwise application of binary operator $\oplus$. The result matrix is stored in m.
- `MMAPM`$(m, \ominus, m_1)$ — elementwise application of unary operator $\ominus$. The result matrix is stored in m.
- `MMAPMV`$(m, \oplus, m_1, v_1, d_2)$ — map $\oplus$ across dimension d_2 of matrix m_1. The result matrix is stored in m.
- `MMAPMS`$(m, \oplus, m_1, r)$ — elementwise apply $\oplus$ to r and all elements of m_1. The result matrix is stored in m.
- `MMAPVV`$(m, \oplus, v_1, d_1, v_2, d_2)$ — map $\oplus$ across $v_1 \times v_2$, spanning dimensions d_1, d_2 of m. The result matrix is stored in m.
- `MASSIGN`(m, m_1) — assign the matrix-valued expression m_1 to the matrix m. m and m_1 must have the same format.

- `MCNVTM`(m, m_1) — like `MCOPYM`, but the formats of m and m_1 differ.
- `MEXPV`(m, v, d) — blow up vector v to a matrix m along dimension d
- `MTRANSPM`(m, m_1) — transpose matrix m_1. The result matrix is stored in m.
- `MTAB`$(m,$ `RANGE`$(i, ...)$, `RANGE`$(j, ...)$ — like `VTAB`, for matrices.

Note that outer product (`MOUTERVV`) is a special case of `MMAPVV`.

Searching and sorting on a matrix

In the following list, rv_1 denotes a matrix row, that is, a vector access.

- `MCOLLM`(m, m_1, f, i, j) — filter out all elements $m_1(i, j)$ fulfilling a boolean condition $f(m_1, i, j)$, parameterized by formal row index i and/or formal column index j. The result matrix is stored in m.
- `MGETSUBM`$(m, m_1, s_1, t_1, s_2, t_2)$ — extract a rectangular submatrix of m_1 defined by the index intervals $(s_1 : t_1, s_2 : t_2)$. The result matrix is stored in m.
- `MSETSUBM`$(m_1, s_1, t_1, s_2, t_2, m_2)$ — replace submatrix $m_1(s_1 : t_1, s_2 : t_2)$ by m_2
- `GETELM`(m, i, j) — return the element $m(i, j)$ if it exists, and 0 otherwise.
- `MSETELM`(m, i, j, r) — set element $m(i, j)$ to r
- `VGETROW`(m, i) — return row i from matrix m,
- `MSETROW`(m, i, rv) — set row i in matrix m to rv
- `VGETCOL`(m, i) — return column i of matrix m
- `MSETCOL`(m, i, cv) — set column i in matrix m to cv
- `VGETDIA`(m, i) — return the vector of elements in the diagonal i of matrix m.
- `MSETDIA`(m, i, v) — set the elements in diagonal i of matrix m to the elements in v
- `MGETL`(m) — return the left lower triangular matrix of m (including the main diagonal)
- `MGETU`(m) — return the right upper triangular matrix of m (including the main diagonal)
- `MPRMROW`(m, iv) — permute the rows of matrix m according to permutation vector iv
- `MPRMCOL`(m, iv) — permute the columns of m according to permutation vector iv

Matrix–vector and matrix–matrix product, decompositions

- `VMATVECMV`(v, r, m, v_1, v_2) — return the matrix–vector product $v = m \cdot v_1 + v_2$.
- `VVECMATMV`(v, m_1, v_1, v_2) — return the vector–matrix product $v = m_1^T \cdot v_1 + v_2$.
- `MMATMULMM`(m, m_1, m_2, m_3) — return the matrix–matrix product $m = m_1 \cdot m_2 + m_3$
- `VUSOLVEMV`(v, m_1, v_2) — solve a system by backward substitution $v = m_1^{-1} \cdot v_2$, where m_1 is upper triangular.
- `VLSOLVEMV`(v, m_1, v_2) — solve a system by forward substitution $v = m_1^{-1} \cdot v_2$, where m_1 is lower triangular.
- `VROT`(v, v_1, m_2) — Givens rotation
- `MMLUD`(m, m_1, m_2, p, t) — LU decomposition of m_2, pivot strategy p, drop tolerance t. The triangular result matrices are stored in m and m_1

- VUPDROW$(v, \oplus, m_1, pr, i, c, space, t)$ — update row i of m_1 in LU decomposition for pivot row pr, start column c, drop tolerance t, dense result vector of size $space$. The result vector is stored in v.

In order to express a transposed matrix–matrix product, the MTRANSP concept has to be applied to the operand matrix to be accessed in transposed order [1]. For dense matrices this can be skipped by toggling the leading dimension index in the MDNS instance.

It is interesting to note that a matrix–vector multiplication for a matrix in CSR format

```
VMATVECMV(..., CSR(...), ...)
```

looks exactly like a transposed matrix–vector multiplication for CSC format

```
VVECMATMV(..., CSC(...), ...)
```

and vice versa. Furthermore, for matrix–vector product the order of nonzero elements within the same row resp. column is not important here, thus the concept variants for CSR and CUR resp. CSC and CUC matrices are equivalent. Thus, for each such pair of equivalent concept variants only one common implementation is required for the back-end.

I/O concepts

READ and WRITE are the concepts for reading and writing a scalar value to a file.

- VREAD(v, F) — read a vector v from file F
- VWRITE(v, F) — write a vector v to file F
- MREAD(m, F, f) — read m from file F in file storage format f
- MWRITE(m, F, f) — write m to file F in file storage format f

There are various file storage formats in use for sparse matrices, e.g. the Harwell–Boeing file format, the array format, or coordinate format [BPRD97].

3.3.2 Exception Slots

For some of the concepts listed above there exist additional slots containing actions specified by the programmer to cover cases when possible exceptions occur. For example, the INV concept (scalar reciprocal) offers a "catch" slot to enter a statement that handles the "division by zero" exception. As another example, for LU decomposition (LUD) on a sparse operand matrix an exception slot indicates what should be done if the allocated space is exceeded.

[1]The reason why we do not define three more concepts for the combinations of transposed operand matrices is that executing a transpose, if not avoidable, is one order of magnitude less costly than a matrix–matrix product, while the execution time of a transpose is in the same order as a transposed matrix–vector product.

3.4 Speculative Concept Recognition

Safe identification of a sparse matrix operation consists of (1) a test for the syntactical properties of this operation, which can be performed by concept recognition at compile time, and (2) a test for the dynamic properties which may partially have to be performed at run time. Regarding (parallel) code generation, this implies that two versions of code for the corresponding program fragment must be generated: one version branching to an optimized sparse matrix library routine if the test is positive, and a conservative version (maybe using the inspector–executor technique, or just sequential) that is executed otherwise.

3.4.1 Compile-Time Concept Matching

The static part of our concept matching method is based on a bottom-up rewriting approach using a deterministic finite bottom-up[2] tree automaton that works on the program's intermediate representation (IR) as an abstract syntax tree or control flow graph, augmented by concept instances and dataflow edges computed during the recognition. Normalizing transformations, such as loop distribution or rerolling of unrolled loops, are done whenever applicable.

The matching rules for the concept idioms to be recognized, called ***templates***, are specified as far as possible in terms of subconcept occurrences (see Fig. 3.2), following the natural hierarchical composition of computations in the given programming language, by applying loops and sequencing to subcomputations. Since at most one template may match an IR node, identification of concept occurrences is deterministic. For efficiency reasons the applicable templates are selected by a hashtable lookup: each rule to match an occurrence of a concept c is indexed by the most characteristic subconcept c' (called the ***trigger concept***) that occurs in the matching rule. The graph induced by these edges (c', c) is called the ***trigger graph***[3]. Hence, concept recognition becomes a path finding problem in the trigger graph. Matched IR nodes are annotated with concept instances. If working on an abstract syntax tree, a concept instance holds all information that would be required to reconstruct an equivalent of the subtree it annotates.

Vertical matching

Vertical matching proceeds along the hierarchical nesting structure (statements, expressions) of the program. It uses only control dependence information which may be explicitly given by a hierarchically structured IR, or derived from the abstract syntax tree.[4] Let us, for simplicity, assume in the following that the IR has the form of a tree that corresponds more or less to the abstract syntax tree.

Vertical matching is applied at an IR node v after the post-order traversals of all subtrees rooted at the children of v are finished. If not all children of v are annotated by a concept instance, then matching for v immediately terminates, as complete information on all children's

[2]To be precise, for the unification of objects *within* a matching rule we apply a top-down traversal of (nested) concept instances for already matched nodes.

[3]An example of a trigger graph will follow in Section 3.5.1

[4]For languages with strictly block-structured control flow, control dependence information can be directly obtained from the abstract syntax tree.

concepts is essential for each matching rule. If a matching rule fails, it aborts. Otherwise, v is matched and annotated with a concept instance.

A program fragment for which one of the matching rules of a concept P matches, is annotated by a concept instance, a summary node that looks, if printed, similar to a call to an externally defined function `P()`. The slots (i.e., the formal parameters) of the concept instance are bound to the corresponding program objects occuring in the code fragment. If we are working on a hierarchically structured intermediate program representation, this summary node holds all information that would be required to reconstruct an equivalent of the subtree it annotates. In short, a concept instance completely describes *what* is being computed in a subtree, but abstracts from *how* it is computed.

As an example, consider the following program fragment:

```
    DO i = 1, N
S1:    b(i) = 0.0
       DO j = first(i), first(i+1)-1
S2:       b(i) = b(i) + A(j) * x(col(j))
       ENDDO
    ENDDO
```

The syntax tree is traversed bottom-up from the left to the right. Statement `S1` is recognized as a scalar initialization, summarized as `ASSIGN(VAR(b(i)),CONST(0.0))`. Statement `S2` is matched as a scalar update computation, summarized as `INCR(b(i), MUL(A(j), x(col(j)))`. Now the loop around `S2` is considered. The index expressions of `A` and `col` are bound by the loop variable `j` which ranges from some loop-invariant value `first(i)` to some loop-invariant value `first(i+1)-1` with step size 1. Hence, the set of accesses to arrays `A` and `col` during the `j` loop can be summarized as vector accesses `V( A, RANGE(, first(i), first(i+1)-1),1)` and `IV(col,RANGE(,first(i),first(i+1)-1,1))`, respectively. The access to array `x` is an indexed vector access. The entire `j` loop is thus matched as

```
INCR( VAR(b,i),
      SREDV( ADD,
             VMAPVV( MUL,
                     V( A, RANGE(,first(i),first(i+1)-1,1)),
                     VX(x, IV(col,RANGE(,first(i),first(i+1)-1,1))))
```

For this particular combination of a scalar reduction `SREDV` with a two-operand elementwise vector operation `VMAPVV` exists a special flat concept, namely `SDOTVVX`, describing a dot product with one indexed operand vector. The unparsed program is now

```
    DO i = 1, n
S1':   ASSIGN( VAR(b,i), CONST(0.0));
S2':   SDOTVVX( VAR(b,i),
                V( A, RANGE(,first(i),first(i+1)-1,1)),
                VX( x,  IV( col, RANGE(,first(i),first(i+1)-1,1))),
                VAR(b,i) )
    ENDDO
```

Although all statements in the body of the `i` loop are matched, there is no direct way to match the `i` loop at this point. We must first address the dataflow relations between `S1′` and `S2′`, which is discussed in the following paragraph.

Horizontal matching

Horizontal matching tries to merge several matched IR nodes v_1, v_2, ... belonging to the body of the same parent node which is, mostly, a loop header. If there is a common concept that covers the functionality of, say, v_i and v_j, there is generally some data flow relation between v_i and v_j that can be used to guide the matching process. For each concept instance we consider the slot entries to be read or written, and compute data flow edges (also called ***cross edges***) that connect slots referring to the same value. Most frequently, these edges correspond to data flow dependences[5], and are thus denoted as FLOW cross edges[6]. These cross edges guide the pattern matching process and allow to skip unrelated code that may be interspersed between two statements belonging to the same thread of computation.

Continuing on the example above, we obtain that the same value of `b(i)` is written (generated) by the `ASSIGN` computation in `S1′` and consumed (used and killed) by the `SDOTVVX` computation in `S2′`. Note that it suffices to consider the current loop level: regarding cross matching, the values of outer loop variables can be considered as constant.

Horizontal matching, following the corresponding template (similar to the second template in Fig. 3.2), "merges" the two nodes and generates a "shared" concept instance:

```
     DO i = 1, N
S′′:   SDOTVVX( VAR(b,i),
                V( A, RANGE(,first(i),first(i+1)-1,1)),
                VX( x, IV( col, RANGE(,first(i),first(i+1)-1,1))),
                CONST(0.0) )
     ENDDO
```

3.4.2 Speculative Concept Matching

In order to continue with this example, we now would like to apply vertical matching to the `i` loop. The accesses to `a` and `col` are supposed to be CSR matrix accesses because the range of the loop variable `j` binding their index expressions is controlled by expressions bound by the `i` loop. Unfortunately, the values of the `first` elements are statically unknown. Thus it is impossible to definitively conclude that this is an occurrence of a CSR matrix vector product.

Nevertheless we continue, with assumptions based on syntactic observations only, concept matching in a *speculative* way. We obtain (see also Fig. 3.3)

[5]More specifically, they correspond to *value-based* flow dependences. A data flow dependence from a statement S_1 to a statement S_2 is called *value-based*, if it is not killed by statements interspersed between S_1 and S_2.

[6]Further types of cross edges that are, for instance, induced by value-based anti-dependences or immediate neighborhood of array accesses have been introduced in [D2,J2], but these do not play a major role in the cases considered in this chapter.

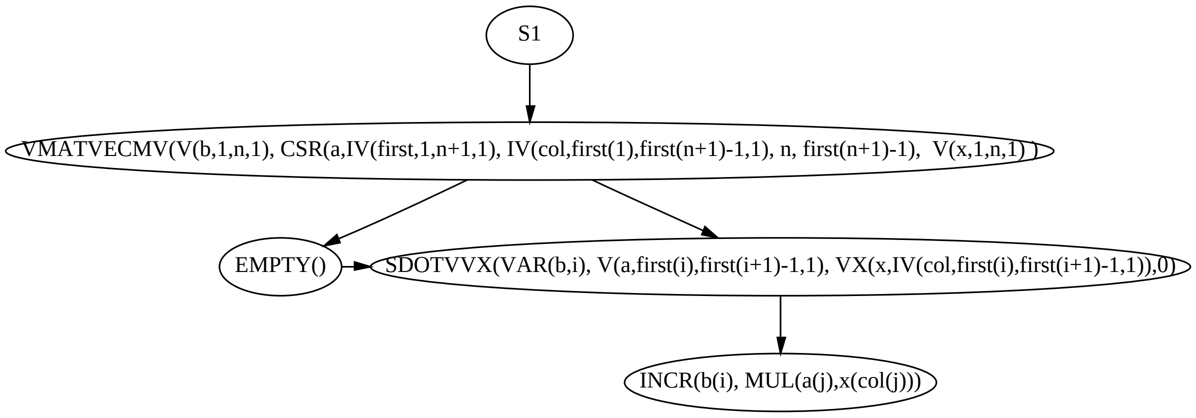

S1
VMATVECMV(V(b,1,n,1), CSR(a,IV(first,1,n+1,1), IV(col,first(1),first(n+1)-1,1), n, first(n+1)-1), V(x,1,n,1))
EMPTY()
SDOTVVX(VAR(b,i), V(a,first(i),first(i+1)-1,1), VX(x,IV(col,first(i),first(i+1)-1,1)),0)
INCR(b(i), MUL(a(j),x(col(j))))

FIGURE 3.3: The program graph (abstract syntax tree) of the CSR matrix–vector multiplication code after concept recognition, generated by DOT. As a side-effect of horizontal matching a pseudoconcept "EMPTY" is generated to hide a node from code generation but allow reconstruction of children concepts if desired.

```
   <assume first(1)=1>
   <assume monotonicity of V(first,1,n+1,1)>
   <assume injectivity of V(col,first(i),first(i+1)-1,1)
                       forall i in 1:n>
S: VMATVECMV( V(b,1,n,1),
              CSR( a, IV(first,1,n+1,1),
                   IV(col,first(1),first(n+1)-1,1),
                   n, first(n+1)-1),
              V(x,1,n,1),
              VCON(0.0,n) );
```

where the first three lines summarize the assumptions guiding our speculative concept recognition. If they cannot be statically eliminated, these three preconditions would, at code generation, result in three runtime tests being scheduled before or concurrent to the speculative parallel execution of `S` as a CSR matrix vector product. The range of the values in `col` needs not be bound-checked at runtime since we can safely assume that the original program runs correctly in sequential.

Now we have a closer look at these conditions for speculative recognition.

Definition 3.1 *(**monotonicity**) We call an integer vector* `iv` ***monotonic*** *over an index range* $[L : U]$ *at a program point* q *iff for any control flow path through* q, `iv(`i`)`$\leq$`iv(`$i+1$`)` *holds at entry to* q *for all* $i \in [L : U - 1]$.

Definition 3.2 *(**injectivity**) We call an integer vector* `iv` ***injective*** *over an index range* `L:U` *at a program point* q *iff for any control flow path through* q*, for all* $i, j \in$`L:U` *holds* $i \neq j \Longrightarrow$ `iv(`i`)` $\neq$ `iv(`j`)` *at entry to* q.

Monotonicity and injectivity of a vector are usually not statically known, but are important properties that we need to check at various occasions.

We must verify the speculative transformation and parallelization of a recognized computation on a set of program objects which are strongly suspected to implement a sparse matrix A. This consists typically of a check for injectivity of an index vector, plus maybe some other checks on the organizational variables. For instance, for nontransposed and transposed sparse matrix–vector multiplication in CSR or CUR row-compressed format, we have to check that

(1) `first(1)` equals 1,
(2) vector `IV(first,1,`n`+1,1)` is monotonic, and
(3) vectors `IV(col,first(`i`),first(`i+1`)-1,1)` are injective for all $i \in \{1, ..., n\}$.

These properties may be checked for separately.

3.4.3 Speculative Loop Distribution

Loop distribution is an important normalization applied in the concept recognizer. As an example, consider the following code fragment taken from the `SPARSE-BLAS` [Gri84] routine `DGTHRZ`:

```
   DO 10 i = 1, nz
       x(i)        = y(indx(i))
       y(indx(i)) = 0.0D0
10 CONTINUE
```

In order to definitely recognize (and also in order to parallelize) this fragment, we need to know the values of the elements of array `indx`. Unfortunately, this information is generally not statically available. But similar as for the speculative recognition of sparse matrix operations we speculatively assume that `indx` is injective in the range `1:nz`. As now there remain no loop-carried dependencies, we can apply loop distribution [ZC90] to the `i` loop:

```
<assume injectivity of INDX(1:NZ)>
DO i = 1, nz
    x(i) = y(indx(i))
ENDDO
DO i = 1, nz
    y(indx(i)) = 0.0D0
ENDDO
```

Applying concept matching to each loop separately makes the speculatively matched copy of the program segment look as follows:

```
<assumes injectivity of indx(1:nz)>
VASSIGN( V(x,1,nz,1), VX(y, IV(indx,1,nz,1)))
VASSIGN( VX(y,indx(1,nz,1)), VCON(0.0,n))
```

Speculative loop distribution saves the original program structure for the code generation phase. This allows to generate also the conservative code variant.

3.4.4 Preservation and Propagation of Format Properties

Even if at some program point we are statically in doubt about whether a set of program objects really implements a sparse matrix in a certain storage format, we may derive static information about some format properties of a speculatively recognized concept instance.

For any concept (or combination of a concept and specific parameter formats) the format property preconditions for its parameter matrices are generally known. If an instance I of a concept c generates a new (sparse) result matrix m, it may also be generally known whether m will have some format properties after execution of I (i.e., a *postcondition*). Such a property π of m may either hold in any case after execution of an instance of c, that is, $\pi(m)$ is installed by c. Or, π may depend on some of the actual format properties $\pi_1, \pi_2, ...$ of the operand matrices $m_1, m_2, ...$ In this case, $\pi(m)$ will hold after execution of I only if $\pi_1(m_1)$, $\pi_2(m_2)$ etc. were valid before execution of I. In other words, this describes a propagation of properties $\pi_1(m_1) \wedge \pi_2(m_2) \wedge ... \rightarrow \pi(m)$. Also, it is generally known which properties of operand matrices may be (possibly) deleted by executing an instance of a concept c.

The assumptions, preservations, propagations and deletions of format properties associated with each concept instance are summarized by the program comprehension engine in the form of pre- and postcondition annotations to the concept instances. Note that the preservations are the complementary set of the deletions; thus we renounce on listing them. If existing program objects may be overwritten, their old properties are conservatively assumed to be deleted. Note that the `install` and `propagate` annotations are postconditions that refer to the newly created values. The shorthand `all` stands for all properties considered.

For example, Figure 3.4 shows how annotations are (conceptually) inserted before the concept instance where a certain piece of program has been speculatively recognized as an occurrence of a CSC to CSR conversion concept.

If applied to an interactive program comprehension framework, these runtime tests correspond to prompting the user for answering yes/no questions about the properties.

Once the static concept recognition phase is finished, these properties, summarized as pre- and postconditions for each concept instance in the (partially) matched program, are optimized by a dataflow framework as described in the following section. This method allows to eliminate redundant conditions and to schedule the resulting runtime tests (or user interactions) appropriately.

3.4.5 Placing Runtime Tests

Program points are associated with each statement or concept instance, that is, with the nodes in the control flow graph after concept matching. In an implementation all properties π of interest for all arrays or array sections A of interest may be stored for any program point q in bitvectors

ASSUME$_{\pi,A}(q) = 1$ iff $\pi(A)$ is `assume`d to hold at entry to q,

DELETE$_{\pi,A}(q) = 1$, iff $\pi(A)$ may be `delete`d by execution of q, and

```
<assume FirstB(1)=1>
<assume monotonicity of IV(FirstB,1,M,1)>
<assume injectivity of IV(RowB,FirstB(i),FirstB(i+1),1)
                       forall i in 1:M>
<delete all of FirstA>
<delete all of ColA>
<install FirstA(1)=1>
<propagate (monotonicity of IV(FirstB,1,M,1))
 then (monotonicity of IV(FirstA,1,N,1))>
<propagate (monotonicity of IV(FirstB,1,M,1)
 and (injectivity of IV(RowB,FirstB(i),FirstB(i+1),1)
                     forall i in 1:M)
 then(injectivity of IV(ColA,FirstA(i),FirstA(i+1),1)
                     forall i in 1:N)>
MCNVTM( CSR( V(A,1,NZ,1), IV(FirstA,1,N+1,1),
             IV(ColA,1,N,1), N, NZ),
        CSC( V(B,1,NZ,1), IV(FirstB,1,M+1,1),
             IV(RowB,1,M,1), M, NZ) )
```

FIGURE 3.4: A recognized matrix conversion computation, where the preconditions and post-conditions on the format properties are explicitly displayed in the annotated intermediate representation.

$INSTALL_{\pi,A}(q) = 1$ iff $\pi(A)$ is `install`ed by execution of q.

Propagations are represented by sets

$PROPAGATE_{\pi,A}(q)$

containing all properties π_j of arrays A_j that must hold at entry to q in order to infer $\pi(A)$ at exit of q.

Moreover, we denote by

$TEST_{\pi,A}(q)$

whether a runtime test of property π of array section A has been scheduled immediately before q. When starting the placement of runtime tests, all $TEST_{\pi,A}(q)$ are zero.

For the placement of runtime tests we compute an additional property

$HOLD_{\pi,A}(q)$ which tells whether $\pi(A)$ holds at entry of q. We compute it by a standard data flow technique (see e.g. [ZC90]) iterating over the control flow graph $G = (V, E)$ of the program, using the following data flow equation:

$$
\begin{aligned}
HOLD_{\pi,A}(q) = TEST_{\pi,A}(q) & \qquad (3.1)\\
\vee \bigwedge_{(q',q)\in E} (HOLD_{\pi,A}(q') \wedge \neg DELETE_{\pi,A}(q') \vee INSTALL_{\pi,A}(q')) & \\
\vee \bigwedge_{(q',q)\in E} \quad \bigwedge_{(\pi',A')\in PROPAGATE_{\pi,A}(q')} HOLD_{\pi',A'}(q') &
\end{aligned}
$$

For the data flow computation of *HOLD*, we initialize all *HOLD* entries by 1. Since the *DELETE*, *INSTALL*, *PROPAGATE* and *TEST* entries are constants for each π, A, and q, the sequence of the values of $\mathit{HOLD}_{\pi,A}(q)$ during the iterative computation is monotonically decreasing and bounded by zero, thus the data flow computation converges.

Clearly, after all necessary runtime tests have been placed, $\mathit{HOLD}_{\pi,A}(q)$ must fulfill

$$\mathit{HOLD}_{\pi,A}(q) \geq \mathit{ASSUME}_{\pi,A}(q) \quad \text{for all } \pi, A, q$$

in order to ensure correctness of the speculative program comprehension. Thus we arrive, as a very general method, at the following simple nondeterministic algorithm for placing runtime tests:

Algorithm: *placing runtime tests*

```
(1) for all  $\pi$ and $A$ of interest do
(2)     for all $q$ do $\mathit{TEST}_{\pi,A}(q) = 0$;
(3)     forever do
(4)         initialize $\mathit{HOLD}_{\pi,A}(q') = 1$ for all program points $q'$
(5)         recompute $\mathit{HOLD}_{\pi,A}(q)$ for all program points $q$ according to equation (3.1)
(6)         if $\mathit{HOLD}_{\pi,A}(q) \geq \mathit{ASSUME}_{\pi,A}(q)$ for all $q$  then break;
(7)         set $\mathit{TEST}_{\pi,A}(q') = 1$ for some suitably chosen program point $q'$
        od
od
```

The goal is to place the runtime tests in step (7) in such a way that the total runtime overhead induced by them in the speculatively parallelized program is minimized. A very simple strategy is to place a test for $\pi(A)$ *immediately after* each statement q' killing property π of A, that is, where $\mathit{KILL}_{\pi,A}(q') = 1$, which is defined as follows:

$$\mathit{KILL}_{\pi,A}(q') := \mathit{DELETE}_{\pi,A}(q') \wedge \neg \mathit{INSTALL}_{\pi,A}(q') \\ \wedge \neg \bigwedge_{(\pi',A') \in \mathit{PROPAGATE}_{\pi,A}(q')} \mathit{HOLD}_{\pi',A'}(q')$$

Of course, this initial setting will typically introduce superfluous tests. For instance, for a sequence of consecutive killings of $\pi(A)$ it is sufficient to schedule a test for $\pi(A)$ only after the last killing program point, provided that $\pi(A)$ may not be assumed to hold at some point within this sequence. Also, a test for $\pi(A)$ after program point q' is superfluous if $\pi(A)$ is not assumed at any point q'' that may be executed after q'. These optimizations can be carried out in a second phase by another data flow framework. [7]

Ideally, there are, once a sparse matrix has been constructed or read from a file, no changes to its organizational variables any more during execution of the program, i.e., its sparsity pattern remains static. In that case SPARAMAT can completely eliminate all but one test on

[7] Alternatively, one may as well start with a test of $\pi(A)$ being inserted immediately before each program point q with $\mathit{ASSUME}_{\pi,A}(q) \wedge \neg \mathit{HOLD}_{\pi,A}(q)$, and then eliminating all tests in a sequence of uses of $\pi(A)$ but the first one (provided that no kill of $\pi(A)$ may occur in between), which is just symmetric to the strategy described above.

monotonicity and injectivity, which is located immediately after constructing the organizational data structures. An example program exhibiting such a structure is shown in the neural network simulation code given in Appendix B.1.

More formally spoken, if a property $\pi(A)$ is needed at many places in the program but the sparsity pattern associated with A remains static, only the first `<assume ... of A ...>` produces really a test, while at all other places the result of this can be reused.

Note that, due to speculative execution, a test can be executed *concurrently* with the subsequent concept implementation even if that is guarded by this test. In particular, possible communication delays of the test can be filled with computations from the subsequent concept implementation, and vice versa. The results of the speculative execution of the concept implementation are committed only if the test process accepts.

If applied to an interactive program comprehension framework, these runtime tests correspond to prompting the user for answering yes/no questions about the properties. Static optimization thus corresponds to reusing such user information.

Optimization of runtime tests in practice The data flow framework above with the nondeterministic placement algorithm can rather be used to verify a given placement of tests, but is hardly helpful to derive a good placement efficiently. In practice, one would instead use standard *code motion* techniques [KRS98]. The simplest and most-investigated variant of code motion is syntactic code motion, which is based on the structural equivalence of subexpressions.

Starting with a conservative placement of tests such as the one proposed in the previous paragraph, our optimization problem could be solved by applying *partial redundancy elimination*. A *partial redundancy* is a computation e (such as a subexpression or subcondition in an assumed condition) for which a path in the program flow graph exists such that e is computed more than once on that path. Partial redundancy elimination, introduced by Morel and Renvoise [MR79], performs a global common subexpression elimination and places instructions such that their execution frequency is minimized, which includes hoisting loop-invariant code out of loops and some other optimizations. Motion-based approaches to partial redundancy elimination have been proposed, for instance, by Dhamdhere [Dha88, Dha91] and by Knoop, Rüthing, and Steffen [KRS92, KRS94]. A good introduction to this topic can be found in the textbook by Muchnick [Muc97, Chap. 13.3].

Code motion for a given term e (the motion candidate) works by inserting statements of the form $t \leftarrow e$ at some points in the program, where t is a new temporary variable, and replacing occurrences of e by references to t where this preserves the program semantics, that is, e is never introduced in a control flow path on that it was absent before. This motion of e is only done if it actually decreases the execution time of the program, that is, it reduces the number of computations along at least one control flow path. The preservation of program semantics, that is, the safety of a placement of $t \leftarrow e$ at a program point, is determined by a dataflow framework similar to the one given in the previous paragraph. *Busy code motion* achieves this goal by placing e as early as possible in the program. However, this strategy tends to produce long live ranges of the new temporary variables, which constrain the register allocator (see Chapter 2). This problem is mitigated by the so-called *lazy code motion* [KRS92], which places computations as early as necessary, but as late as possible to keep the live ranges short.

As code motion techniques are now considered as state-of-the-art compiler technology, a further description and implementation of the optimization of the placement of runtime tests is not necessary here.

Note, however, that neither syntactic nor the more powerful semantic code motion techniques are able to derive an *optimal code placement*, as they are limited to a local scope of moving to the next control flow predecessor. A discussion of the limitations and differences of code motion and code placement is given by Knoop, Rüthing, and Steffen [KRS98].

Beyond optimizing the placement of the runtime tests, we should also optimize the tests themselves. The runtime tests on monotonicity and injectivity can be parallelized, as we shall see in the following paragraphs.

3.4.6 A Parallel Algorithm for the Monotonicity Tests

Monotonicity of a vector $x(1:n)$ is very easy to check in parallel. Each of the p processors considers a contiguous slice of x of size at most $\lceil n/p \rceil$. The slices are tested locally; if a processor detects nonmonotonicity in its local part, it signals a FAIL and aborts the test on all processors. Otherwise, the values at the boundary to the next upper slice of x are checked concurrently. The test passes if no processor detects nonmonotonicity for any pair of neighboring elements.

On a distributed memory architecture, the latter, global phase of this algorithm requires each processor (but the first one) to send the first element of its slice of x to the processor owning the next lower slice; thus each processor (but the last one) receives an element and compares it with the element at the upper boundary of its slice.

3.4.7 A Parallel Algorithm for the Injectivity Tests

A parallel algorithm may reduce the overhead of the injectivity test. For a shared memory parallel target machine we apply an algorithm similar to bucket sort[8] to test injectivity for an integer array `a` of n elements, as it is likely that the elements of `a` are within a limited range (say, $1:m$).[9] We hold a shared temporary array `counter` of m counters, one for each possible value, which are (in parallel) initialized by zero. Each processor k, $k = 1, ..., p$ increments[10] the corresponding counters for the elements $a((k-1)n/p + 1 : kn/p)$. If a processor detects a counter value to exceed 1, it posts a FAIL signal, the test returns FALSE. Otherwise, the test accepts. The test requires m additional shared memory cells. Concurrent write access to the same counter (which may sequentialize access to this location on most shared memory systems) occurs only if the test fails. Thus, the runtime is $O((m+n)/p)$.

On a distributed memory system, we use an existing algorithm for parallel sorting of an integer array of size n on a processor network that may be appropriately embedded into the present hardware topology. As result, processor i holds the ith slice of the sorted array, of size

[8]A similar test was suggested by Rauchwerger and Padua [RP94].

[9]m is to be chosen as a conservative overestimation of the extent of the compressed matrix dimension which is usually not statically known.

[10]This should be done by an atomic fetch&increment operation such as `mpadd(&(counter[a[j]]),1)` on the SB-PRAM, cf. Chapter 4.

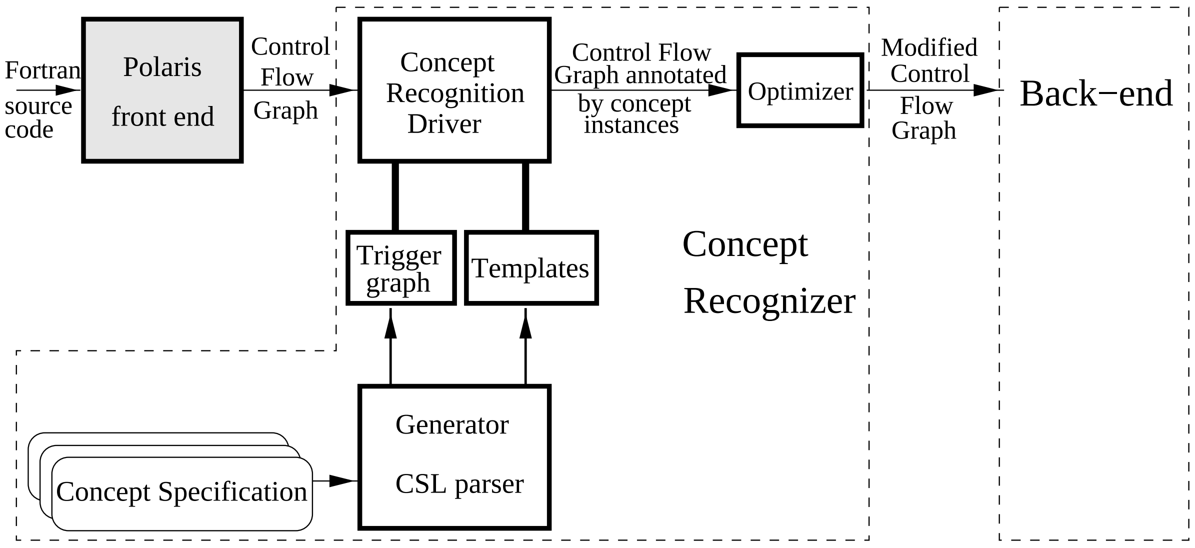


FIGURE 3.5: SPARAMAT implementation.

n/p. Furthermore, each processor $i > 0$ sends the first element of its slice to its predecessor $i-1$ who appends it as $(n/p+1)$st element to its local slice. Each processor now checks its extended slice for duplicate entries. If the extended slices are injective, then so is the original array. The runtime is dominated by the parallel sorting algorithm.

3.5 SPARAMAT Implementation

The SPARAMAT implementation uses the Polaris Fortran compiler [Blu94, FWH$^+$94] as front-end and technical platform. Its intermediate program representation (IR) is a hierarchically structured control flow graph, where loop headers and if headers are still explicitly given; hence, control dependence information, which is required for the matching process, can be easily inferred from the IR, at least as long as arbitrary jumps do not occur in the input program.

SPARAMAT is conceptually broken up into two major systems, the concept recognizer and the code generator (see Figure 3.5 and Figure 3.6). When configuring the SPARAMAT system, the generator accepts specification files describing concepts and templates, and creates C++ files containing the trigger graph and the template matching functions. These are then linked with the SPARAMAT concept matching core routines to form the driver.

The analysis of the program begins when the control flow graph of the specified program is passed from the Polaris front-end to the driver. The driver, upon completion of analysis, passes to the optimizer the control flow graph annotated with concept instances. The runtime tests are optimized in a separate pass. This optimizer passes the modified control flow graph to the back-end.

If the intended application purpose is automatic parallelization, the back-end uses the attached concept instances and remaining conditions to insert runtime checks and replace matched code with calls to parallel implementations of the concepts (see Figure 3.6).

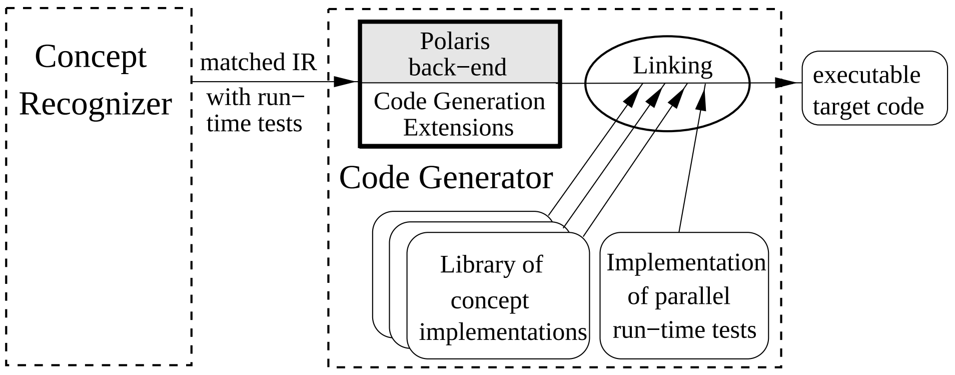

Concept Recognizer
matched IR with run−time tests
Polaris back−end
Code Generation Extensions
Linking
executable target code
Code Generator
Library of concept implementations
Implementation of parallel run−time tests

FIGURE 3.6: SPARAMAT code generator implementation for an automatic parallelization system.

If applied in an interactive framework, SPARAMAT could instead compute and present a certainty factor for specific matrix formats and then ask the user, displaying the code, if the detected format is correct. This requires a user interface to display the code highlighting matched concepts, such as in the PAP Recognizer [J6].

Due to the generative approach, the implementation can be easily extended for further formats, concepts, and templates by any user of the SPARAMAT system.

3.5.1 Static Concept Matching

The driver algorithm of the pattern matcher traverses the treelike IR recursively, applies at each IR node the templates that may match according to the inspection of the trigger graph, and, if one of these matches, annotates the node with an instance of the recognized concept. Normalizing transformations, such as loop distribution or rerolling of unrolled loops, are done whenever applicable.

Vertical matching

Vertical matching uses only control dependence information and equality checks on variable occurrences. Data flow information is not required for vertical matching. The possible candidate matching rules, derived from the trigger graph, are tested one after another; only one of them can match the IR node under consideration. Hence, the recognition process is deterministic. Note that it is a concept or template design error if two different templates match the same subtree.[11]

[11]In some cases, there are flat, “syntactic sugar” concepts for special combinations of generic concepts, for instance SDOTVV (dot product), which may be expressed using SREDV and VMAPVV. For these special cases it is important that their templates are matched prior to the templates for the general case. In the current implementation, this ordering of templates is still left to the SPARAMAT user.

Horizontal matching

When horizontal matching succeeds, two IR nodes are virtually merged to a single node. Technically, the IR structure remains unchanged in order to allow later for a potential undo of this recognition step. Instead, the relevant summary information for the new concept covering both nodes is annotated with one of the two nodes while the other node is hidden from further pattern matching steps by annotating it with the `EMPTY` concept, so the code generator will ignore it, and vertical matching can later continue at the shared parent node. The old, now overwritten concept instances are internally saved.

Trigger concepts and trigger graph

Narrowing the scope of searching for the set of candidate templates is essential for horizontal as well as for vertical matching, because a naive application of all existing template functions to a particular node is impractical and unnecessary.

For vertical matching, SPARAMAT uses the concept matched for the child node[12] as trigger concept. For horizontal matching, the trigger concept is likewise defined as (usually) the concept annotating the target node of a cross edge.

SPARAMAT represents the trigger graph internally as a table indexed by the trigger concept that yields a list of all promising template functions. The trigger graph is generated automatically when the generator processes the concept specifications. Figure 3.7 shows the trigger subgraph for the concept `SDOTVV` (ordinary dot product of two vectors). Note that the "leaf" concepts `IVAR` and `RVAR` (integer and real variable accesses) never have child nodes in the syntax tree, therefore an artifical concept `NONE` is inserted as an artificial source node of the trigger graph in order to serve as an implicit trigger concept for leaf concepts. Each edge is labeled with the name of the corresponding template function. A dotted edge denotes a template for horizontal matching.

Note that the trigger graph does not represent all the concepts that are required for a concept to match. Rather, it shows the concepts that have a more or less characteristic relationship with the concept of an already matched child or sibling node.

3.5.2 Delayed Format Resolution

In some situations the matrix format might not be clear until a discriminating piece of code is encountered. Until then, it may be necessary to store a set of possible formats in the concept instance and postpone the final identification of the format to a later point in the matching process. In order to support this, we use a special, variant concept that summarizes the same matrix object in a set of different possible formats within the same slot of the concept instance.

3.5.3 Descriptors

A key operation in finding cross edges for horizontal matching is computing the intersection of two (bulk) memory accesses. There are two types of intersection that are of interest here.

[12]If the child node is annotated with a concept instance that has further concept instances as parameters, only the topmost concept in this hierarchy is used as trigger concept.

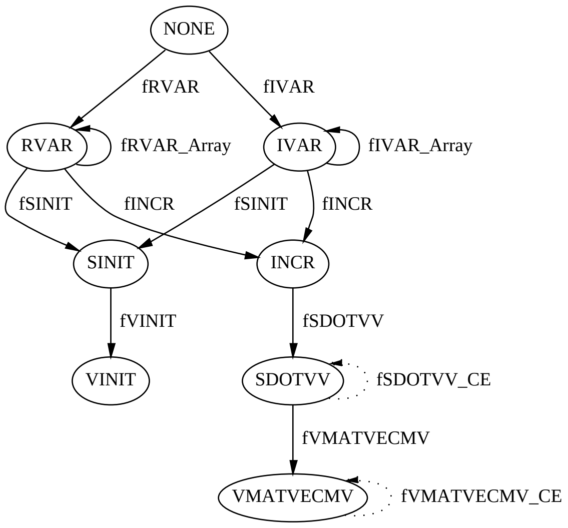


FIGURE 3.7: Trigger graph as far as used for matching the matrix–vector multiplication concept.

For instance, a write access w to an array a in statement S_1 that updates all elements read in a subsequent read access r to a in statement S_2 may result[13] in a FLOW cross edge (S_1, S_2). Hence, a safe underestimation (*must-intersect*) is required for determining the source of a cross edge in horizontal matching (see Section 3.6.5).

In contrast, a write access w' in a statement S' located between S_1 and S_2 may kill the cross edge if it updates some of the elements read by r, that is, the set of elements accessed in w' has a nonempty intersection with these accessed in r. Hence, for the nodes spanned by data flow cross edges in horizontal matching (i.e., fill nodes, see Section 3.6.5), a safe overestimate of the intersection (*may-intersect*) is necessary when checking that the cross edge is not killed by w'. If an intersection cannot be statically excluded, it must be conservatively assumed to exist, hence the horizontal matching will fail, unless speculation is explicitly desired.

Assuming that all aliasing relations are explicitly given, this is, as far as scalar variables are concerned, just common data flow analysis.[14] The problem becomes hard for (bulk) array accesses, such as vector and matrix accesses, where aliasing information may be partially unknown at compile time.

Often it is sufficient to know that two bulk accesses to the same array are disjoint, that is, they are addressed at runtime by disjoint index subsets. This is quite straightforward to test for nonindirect, rectangularly shaped bulk array accesses [CK87, HK91]. Two array accesses may intersect if they have matching identifiers and the index expression ranges intersect in all dimensions. Note that index expressions can be constants variables, arithmetic expressions involving variables or even array accesses.

An *access descriptor* is a data structure that describes the extent of an access to an (array)

[13]Assuming that S_1 and S_2 belong to the same control dependence block in the program.

[14]Note that, after procedure inlining, all aliases (except for array references) are explicit, as there are no pointers in Fortran77.

variable, that is, the set of memory locations addressed. The variable could be a scalar or a one- or higher-dimensional array—for the purpose of determining intersection, all variables are normalized to a unique and general descriptor format, which reduces the number of cases to be distinguished. A descriptor consists of two components: (1) the identifier of the variable, and (2) a list of range objects that describe the access limits and stride for each dimension. If the variable is a scalar, then only the name field of the descriptor is set. A *range object* consists of four components: (a) the lower bound, (b) the upper bound, and (c) the stride of the access, and (d) a pointer to another descriptor object. If the accessed range is a constant or a nonvarying variable, say `c`, then the lower bound and upper bound of the range object is set to `c` and the stride to 0. Otherwise the lower and upper limit and the stride are set to the corresponding (maybe symbolic) expressions. For a vector or matrix access resulting from an array reference in a loop, these limits can usually be derived by substituting[15] the loop bounds in the corresponding index expression. If the access is an indirect array access, the descriptor pointer is set to the computed descriptor for that array, and the ranges are set to conservative estimates.

Access descriptors have been introduced in the context of parallelizing compilers, usually to summarize the behaviour of subroutines in interprocedural program analysis. Common operations on descriptors are intersection, union, and difference. The structures that have been proposed to realize descriptors include regular sections[16], octogonal extensions of regular sections by adding diagonal boundaries [Bal90], and arbitrary convex polytopes defined by linear inequalities [AI91]. A more general representation of array accesses are last-write trees [Fea91, MAL93], but these are more difficult to use and involve expensive operations. In [D2] we proposed an extension of regular sections to trapezoidal sections with a compact representation and the possibility to exclude certain values from an index interval. Note that, by now, no descriptor has been proposed that allows the representation of indirect array accesses. See also [D2] for more pointers to related work on array access descriptors.

For SPARAMAT, we use thus an extension of a standard array access descriptor, namely regular sections for simplicity, by a symbolic representation of indirect array accesses. For testing must-intersection of two indirect array accesses, the test algorithm just recurses to the corresponding descriptors for the indexing array accesses. If these index vectors have a nonempty must-intersection, so do the indexed vectors as well.

3.6 The Concept Specification Language CSL

Creating concept recognition rules directly as C++ routines by hand is a tedious, time consuming and error prone practice best left to automation. A generative approach solves this problem, allowing the rapid and concise specification of concepts and templates in a high-level language. For SPARAMAT, we have hence defined a concept specification language, called CSL.

[15]This method works for index expressions that are monotonic in the loop variable, such as expressions linear in the loop variable.

[16]A *regular section* is a rectangular section that is defined as cross product of intervals on the axes of the index space, [CK87].

The specification of a concept in CSL is fairly straightforward. Each component of a specification, as outlined below, translates directly to a construct of the language. Like most other languages, CSL is whitespace insensitive and block structured.

As a running example we use matrix–vector multiplication for a MSR matrix (cf. Section 3.2). The corresponding specification is given in Figure 3.8.

3.6.1 Header Specification

A concept specification consists of a specification of the concept signature and a set of template specifications. It starts with the keyword `concept` followed by the concept name and a block that contains a list with the names (optional) and types of the slots, followed by the template specifications.

For dependency analysis purposes, slots are assigned ***usage modes***: read (`in`), write (`out`), read and write (`inout`) and ignore (`none`).

After a semicolon, the slot type is specified. The ***slot type*** is used to identify a specific concept or a group of concepts that return values of the same type and/or shape. Only concepts in the proper concept group can appear in the correspondingly typed slot. If a concept has at least one `out` slot, then the slot type of the first of these is considered the *result type* of the concept.[17]

Here is a (nonexhaustive) list of some built-in slot types (the user may add more if necessary by defining groups of concepts):

- `range`: A `RANGE` concept instance holding a loop variable and bounds.
- `rvar`: A `VAR` instance.
- `ivar`: An `IVAR` instance.
- `real`: A `CON` or `VAR` instance.
- `int`: An `ICON` or `IVAR` instance.
- `vector`: A vector-valued concept instance.
- `ivector`: An integer-vector-valued concept instance.
- `xvector`: A `VX` or `IVX` concept instance of an indexed vector access.
- `ixvector`: An `IVX` concept instance.
- `matrix`: A concept instance for a matrix access.
- `operator`: The concept name of a matched operator (e.g. `MUL`).
- `symtabentry`: A name, i.e. an IR symbol table entry.

Hence, when concept instances occur in pattern matching or when creating new concept instances, these occurrences can be statically type-checked. The CSL parser issues an error message if it detects a type clash in a CSL concept specification.

[17]We never encountered the case where an instance of a concept with more than one result slot, such as `SWAP` or `MMLUD`, appeared as a parameter of another instance. Instead, these tend to occur always as root instances for statements.

3.6.2 Concept Instance Lists

Concept instance lists are a CSL data structure to represent FORTRAN code constructs that contain a variable number of concepts—specifically, array indices. First attempts of writing CSL code to match scalar variable arrays without concept lists turned out to be problematic. For instance, the `VAR` (or `IVAR`) concepts may have a variable number of arguments, one for every possible array dimension. If the number of arguments per concept were fixed, different concept names, depending on the number of arguments (FORTRAN restricts the number of array indices to five), would be required. This leads to a combinatorial explosion of concepts. For this reason, the number of argument slots is now fixed for every concept, while every slot may represent a list of parameters. Concept instance lists are denoted in CSL by a sequence of comma-separated concept instances, enclosed in braces. In order to enhance readability by omitting braces, the CSL parser converts occurrences of concept instances (in CSL expressions) that have a variable number of arguments, such as `VAR`, automatically to the corresponding structure using a concept instance list in the proper place, wherever this is nonambiguous. Consequently, we can abbreviate in CSL most occurrences of one-element lists by just writing the element itself.

The concept instance list data structure also supports the annotation of a single IR node with more than one concept instance, which is helpful at some transformations such as `IF` distribution.

3.6.3 Selector Expressions

When referring to a certain part of a structural pattern specification in CSL, it is possible to address subpatterns that are already bound to a variable. In addition the ability to select specific elements of a concept list proved useful.

The selector syntax in CSL is very simple. In order to select a concept instance argument, a dot (’`.`’) followed by the argument index is appended to the variable, where the arguments of a concept instance are numbered starting at zero.

For instance, for the following structural pattern specification

```
pattern
  sibling ($init) ASSIGN( VAR($s, $varlist), $const)
  fill    ($fill)
  node    ($node) SDOTVV( VAR($b, $blist),
                          V($a, $alist),
                          VX($x, $xlist),
                          VAR($s, $slist))
```

the CSL expression `$node.3` selects the fourth argument, which is `VAR($s, $slist)`.

For selecting an element from a concept list, the concept index (again counting from zero) is appended in brackets (e.g., `$alist[0]`) to the CSL variable name.

```
concept VMATVECMV
{
  // b = A*v + init
  param (out)  $p_b: vector;
  param (none) $p_rr: range;
  param (in)   $p_A: matrix;
  param (in)   $p_v: vector;
  param (in)   $p_init: vector;

  templateVertical    // CSR format
  {
    pattern
       node  DO_STMT $lv=$lb:$ub:$st
      child    SDOTVVX( VAR( $s ),
                        RANGE( $rlv, $rlb, $rub, $rst )
                        V( $a, $lb, $ub, $st ),
                        VX( $v, IV( $arr1, $lb1, $ub1, $st1 ) ),
                        VAR( $s ))
    where
    {
        IsEqual($rlb->array(), $rub->array())
      && IsLowerFIRArray($rlb, $lv)
      && IsUpperFIRArray($rub, $lv)
      && IsArrayIndexedOnlyBy($s->GetExpr(), $lv)
      && IsArrayIndexedOnlyBy($a->GetExpr(), $rlv)
      && IsArrayRef(ArrayIndex(0, $v->Expr()))
      && ArrayIndex(0, $v->GetExpr())->array() == $rlb->array()
      && IsArrayIndexedOnlyBy(ArrayIndex(0, $v->GetExpr()), $rlv)
    }
    pre
    {
      ForAll($lv, $lb, $ub,
             Injectivity($rlb->array(), $rlb, $r->GetEnd()));
      Monotonicity($rlb->array(), $lb, $ub);
    }
    instance VMATVECMV(
               newVector($s,$lv,$lb,$ub,$st),
               newRANGE($lv,$lb,$ub,$st),
               newCSR(newVector($a,$rlb->array()($lb),
                                $rlb->array()($ub+1)-1,1),
                      newIV($rlb->array(),$lb,$ub+1,1),
                      newIV($arr1,$lb,$ub+1,1),
                      $ub-$lb+1,
                      $rlb->array()($ub+1)-$rlb->array()($lb)),
               newVector($v,$lv,1,$ub,1),
               newVector($s,$lv,$lb,$ub,$st) )
    post
    {
      ForAll($lv, $lb, $ub,
             Injectivity($rlb->array(), $rlb, $rub));
      Monotonicity($rlb->array(), $lb, $ub);
    }
  }
}
```

FIGURE 3.8: Excerpt of the CSL specification of the `VMATVECMV` concept for matrix–vector product.

3.6.4 Specification of Templates for Vertical Matching

For each nonbasic concept there should be at least one template that matches it. The templates are classified into templates for vertical matching and templates for horizontal matching.

The conditions for a vertical template to match an IR node can be classified into five main components:

- **Structural matching conditions**: A structural pattern describes conditions on control dependence, concept instances annotated at child nodes, and equality of certain variables, which must be met for the concept to match the current node. Whether such a structural pattern matches can be decided at compile time by inspecting the IR.

- **Additional conditions**: Conditions on some expressions on unmatched nodes and on slots of concept instances that can not easily be integrated in the structural matching conditions. An example is the frequent situation where several closely related cases that involve slight structural variation should be handled together in a single template.

  These conditions need auxiliary C++ functions that check certain properties of program objects and concept instances. These functions are provided by the SPARAMAT system; the user may add further functions if necessary.

- **Runtime preconditions**: Runtime conditions that must be met when executing speculatively matched and replaced code. This component is not required for nonstable concepts that do not imply an option of code replacement, and not for templates that involve no speculative matching decision.

- **Concept instance creation**: The code necessary to build the concept instance and fill its slots.

- **Runtime postconditions**: Runtime conditions that will be valid after code execution, assuming that the preconditions held.

Accordingly, a template specification for vertical matching is structured into five ***clauses***.

A vertical template block is identified by the `verticalTemplate` keyword. The specification of the static matching criteria is subdivided into two clauses. The first clause, starting with the `pattern` keyword, describes the main structural constraints on the child's concept instance and on the current IR node.

For instance, the pattern described in the example template in Figure 3.8 is a `DO` loop with one child node annotated by a `SDOTVVX` instance. The CSL variable `$lv` binds the loop variable and the variables `$lb`, `$ub` and `$st` bind the lower bound, upper bound and the stride expression, respectively. These bindings are used to test properties of code or specific slots in the subsequent `where` clause.

Later occurrences of already bound variables in the `pattern` clause result in implicit tests for structural equality that are automatically inserted by the generator. In our running example, such an implicit test is generated to verify that the argument found in the initializer slot of the dot product is equal to the result variable of the dot product. This variable `$s`, expanded to a vector access spanned by the `DO` loop, is later used as argument for the offset slot `$init` of the `VMATVECMV` instance to preserve the semantics of the program.

The concept instance of the child node can contain nested pattern instances. This powerful feature makes it easy to specify which further concept instances are expected to occur in the slots of the child's concept instances, and to assign names to the slots of those inner concept instances.

The next clause, starting with the `where` keyword, tests static properties that are not expressed in the `pattern` clause. The C++ code in the `where` clause is a boolean expression that consists of calls to auxiliary C++ library functions. The `where` clause is only evaluated if the `pattern` criteria match.

In virtually all code that manipulates sparse matrices in a format that stores the indices of the rows in a separate array (see the CSR, MSR, CSC and JAD format descriptions), a frequently occurring common pattern is the pair of expressions `fir(i)` and `fir(i+1)-1`, where `fir` is the array containing the starting indices of each row, and `i` is a loop variable ranging over the rows. The auxiliary C++ functions `IsLowerFIRArray` and `IsUpper-FIRArray` test if the given loop limit expressions match these patterns. The auxiliary function `array` extracts the array name from an array reference. The auxiliary function `IsEqual` checks for structural equality of its arguments. The auxiliary function `IsArrayIndexe-dOnlyBy` returns true if the first argument is an array where only one index expression contains the variable given in the second argument. The auxiliary function `IsArrayRef` returns true if a given expression is an array reference. Finally, the auxiliary function `ArrayIndex` returns an index expression, selected by the first argument, of the array named in the second argument. This collection of additional conditions completes the static test for an occurrence of the `VMATVECMV` concept.

The following clause, starting by the keyword `pre`, specifies the runtime conditions that must be met to guarantee that the code does indeed implement a matrix–vector multiplication in the CSR format.

The next clause, starting with the keyword `instance`, defines how an instance of the concept is created after the static part of the template matches, and fills the slots. The predefined C++ functions `newRANGE`, `newV`, `newIV`, `newCSR`, and so on, create the corresponding concept instances as parameters to the new `VMATVECMV` instance. A special case are the functions `newVector` and `newIVector` that compute the section of accessed vector elements automatically by substituting the loop bounds for the occurrences of the loop variable in the index expressions of the array reference passed as the first argument.

The last clause, starting with the `post` keyword, defines the runtime conditions that will hold after the code implementing the concept has been executed. In our example the `post` conditions are just the same as the `pre` conditions.

3.6.5 Specification of Templates for Horizontal Matching

Templates for horizontal matching consist likewise of the following parts:

- Structural matching conditions
- Additional constraints
- Runtime preconditions

```
S1:    b(j) = 0.0
S2:    c = a(1)
S3:    do 10 i = 1, 10
S4:      b(j) = b(j) + a(i) * x(col(i))
S5: 10 continue
```

FIGURE 3.9: Example FORTRAN code implementing a dot product, with an interspersed statement (`S2`).

- Instance creation
- Runtime postconditions

However, as multiple nodes in the same IR hierarchy block are involved in horizontal matching, CSL allows to specify individual clauses for each of these parts for each involved node, as can be seen in the example specification in Figure 3.10.

A specification of a horizontal template consists thus of several ***clauses***. The `pattern` clause defining structural matching conditions consists of several ***pattern units***. These are indicated by the keywords `sibling`, `fill` and `node`. Optional `where` clauses, specified separately for each node, are used to express further conditions on the IR nodes that match the sibling, the fill or the node pattern unit. The `define` clause allows the CSL programmer to create auxiliary variables and assign them. In the `conditions` clauses the CSL programmer specifies constraints on the data dependences that must be met (see detailed description below). Finally, the `instance` clauses detail how the intermediate representation nodes matching the sibling unit and the node unit are modified if the template matches.

The `node` keyword specifies the start of the cross edge and the `sibling` keyword indicates the target of the cross edge. The programmer must label the pattern units with identifiers to relate the constraints, expressed in the subsequent clauses, to the respective pattern units. The direction of the cross edge is determined by the position of the `node` keyword in relation to the `sibling` keyword. In Figure 3.10 the cross edge starts at a node annotated with the `SDOTVVX` concept instance and terminates at a node with an `ASSIGN` concept instance. The fill pattern unit refers to code spanned by the cross edge.

It is important that the CSL programmer is aware of the order of binding of pattern unit identifiers, the variables created therein, and the variables assigned in the define clauses for the respective pattern unit. The concept expected to match the node corresponding to the `node` unit serves as the trigger concept to invoke the template. Thus, the bindings for the `node` pattern unit are done first. The template searches then for the cross edge target, which is the first intermediate representation node that matches the conditions of the `sibling` unit, provided that all nodes between source and target fulfill the data constraints for the `fill` unit. In Figure 3.9, statement `S2` is first tested to see if it matches an `ASSIGN`. Then, it is tested whether it writes to a variable that matches the fourth argument from the `SDOTVVX` concept instance (this is expressed via the data dependency constraints described below). Since it does not match, `S2` is considered as "fill" and hence tested if it interferes with the given assumptions of a cross edge. In the present example, there is no interference. Then `S1` is considered as "sibling". Indeed, it matches all the constraints on the cross edge target.

```
templateHorizontal
{
  pattern {
    sibling ($init) ASSIGN( VAR($var, $varlist), $const)
    fill    ($fill)
    node    ($node) SDOTVVX( RANGE( $lvar, $lb, $ub, $st),
                             VAR( $bout, $bout_list),
                             V( $a, $alist),
                             VX( $x, $xlist),
                             VAR( $bin, $bin_list) )
  }
  where ($init) {
    IsCONST($const) || IsICONST($const)
  }
  define ($node) {
    @(list)$bin_notwrite  = ExtractVarsFromList($bin_list);
    @(list)$alist_notwrite = ExtractVarsFromList($alist);
    @(list)$xlist_notwrite = ExtractVarsFromList($xlist);
  }
  conditions ($init) {
    read = {}
    write = { $node.4 }  // implies isEqual($var,$bin)
  }                      // and isEqual($varlist,$bin_list)
  conditions ($fill) {
    notread = { $node.4 }
    notwrite = { $node.2, $node.3, $bin_notwrite,
                 $alist_notwrite, $xlist_notwrite }
  }
  instance ($init) EMPTY()
  instance ($node) SDOTVVX($node.0, $node.1, $node.2, $node.3, $init.1)
}
```

FIGURE 3.10: CSL template specification for matching a SDOTVVX with initialization.

Data constraints for the cross edge target, and the nodes spanned by the cross edge, are differentiated by referring to the pattern unit identifiers (`$sib` and `$fill` respectively). Note that the identifier for the source node of the cross edge is only used for accessing arguments of the concept instance at that node.

The `define` clause is a feature that was added to simplify references to subconcepts on the source or the target of the cross edge. Variables in the define block are created and bound to the results of user-defined functions or other variables. This greatly enhances both efficiency and readability of concept specifications—functions need not be called more than once and the storing variables can be clearly named. The variables could be defined in the `where` clause as well, like in many vertical templates, but creation of auxiliary variables required a special clause. For a vertical template, there is only one time where the `where` clause can be evaluated, namely after the conditions are met for matching the parent and its children. In contrast, for horizontal matching, there are three possible times where a `where` clause could be evaluated for a horizontal template: after the source of the cross edge matched, after the target matched and after the fill conditions matched. In order not to constrain the programmer, a horizontal template can contain up to three `where` clauses and requires each `where` keyword to be followed by one of the pattern unit identifiers (`$sib`, `$fill`, or `$node` in this example) to clarify when the contents of the clause should be evaluated.

The constraints on data dependencies between the source, target and spanned nodes of the cross edge are expressed in the `conditions` clauses. In the same way as for the `where` and `define` clauses, these clauses are labeled with a pattern unit identifier to specify the part of the cross edge the clause applies to.

The description of the expected data dependences for the sibling is divided into two sets: `read` and `write`. The `read` set contains the variables that must be read in the sibling's concept instance. For most cases the `read` set will be empty. The `write` set contains the variables that must be written to by the sibling. In our example, the variable in the last argument of the `SDOTVVX` instance must be written to.

Negative data dependency constraints on the fill nodes are expressed by two sets in the appropriately labeled `conditions` clause, the `notread` set and the `notwrite` set.

The `notread` set identifies what variables must not be read by any statement in the fill. Consider the different situation where, in Figure 3.9, statement `S2` is changed to `r = b(j)` and assume for a moment that no `notread` set were specified. Then the cross edge specification in Figure 3.10 matches, and the information from the cross edge target is merged into a new concept instance annotating the cross edge source—although this transformation is wrong: the variable `r` receives an undefined value, since `b(j)` is no longer initialized to zero. For that reason, `b(j)` has to be placed in the `notread` set.

Similar to the `notread` set, the `notwrite` set contains variables that must not be written to in the fill. Consider Figure 3.9 where statement `S2` is changed to `j = 3`. The `SDOTVVX` initialization cross edge is no longer valid because the change to `j` makes the value of `b(j)` on statement `S1` possibly different from that in `S3`.

In order to obtain the variables to be placed into `notread` and `notwrite` sets, we have auxiliary functions such as `ExtractVarsFromList`. This user-defined function extracts all the variables from the supplied list of expressions, and returns them in a list. For instance the variable `$bin_notwrite` is set in a `define` clause to the integer variable concept instance `IVAR(j, {})`.

The `instance` clauses at the end of the horizontal template specify the results. The node that should be annotated by a newly created concept instance is referenced with the corresponding pattern unit identifier. In Figure 3.10, the `instance` clause for the cross edge target replaces the old concept instance with an `EMPTY` concept instance, indicating that this node contains no pertinent information and should be ignored by later matching steps. The information from the sibling is incorporated into the new concept instance created for the source of the cross edge.

3.6.6 Further Features of CSL

Handling intrinsic functions

Intrinsic functions exist in the FORTRAN runtime library and hence cannot be inlined. We added these to CSL as special operator concepts. Simple CSL templates were written to match their occurrences in the FORTRAN code.

Pre-Matching of GOTO jumps

FORTRAN has no statement for structured jumps out of a loop such as the `break` and `continue` statements in C. Instead, using a `GOTO` jump to a label located just after the loop is a common idiom for simulating a `break` in FORTRAN code

As SPARAMAT is based on a tree traversal of the intermediate program representation and thus only works on well-structured programs, it cannot handle arbitrary `GOTO` jumps immediately. Instead, we perform a prepass on the IR, locate occurrences of `GOTO` statements that simulate a `break`, and annotate them with `BREAK` concepts, which can thus be used by the CSL programmer in the matching rules. Jumps simulating `continue` can be handled accordingly.

`IF` distribution

Conditional statements in the input program usually present a difficult issue for the CSL programmer. The statements in the body of a condition can appear in many different orderings (data dependences allowing) and different FORTRAN programmers may group the statements in the body of the conditional statement differently.

Instead of trying to merge these separate statements together, SPARAMAT converts all statements in the body of the condition into a *guarded form*. The condition is copied for each statement, creating one-sided `IF` statements with only one statement in its body,

Horizontal matching without cross edges

`IF` distribution does, though, not eliminate the problem that the ordering of the condition statements affects matching.

Note that this is an instance of a more general problem that occurs in horizontal matching where the statements are completely unrelated by data dependencies, such that no cross edges can be used to guide horizontal matching. This problem can be solved in several ways; either within the generator, by automatically generating all possible variations of the template to take all possible permutations into account, or by making the matcher itself tolerate various orderings automatically. A third, and perhaps simplest, variant consists in defining more versatile concepts and cyclic templates, as the author did in PARAMAT [D2,J2] for matching difference stencil operators.

Debugging support

Debugging of a template is inevitable. All hand-made changes to the generated code will be lost upon the next invocation of the generator, therefore there is a directive in the CSL language that instructs the generator to insert debugging information in the generated code. This debugging directive of CSL can be applied globally to all templates or constrained to particular concepts, or templates.

Multiple CSL files

As the number of concepts increased, it became cumbersome to keep all concept specifications in a single CSL file. In addition, compilation time increased because many unmodified concepts were recompiled.

In order to alleviate these problems, we introduced separate compilation of multiple CSL files. This improved the extensibility, enhanced the overall organization of the implemented concepts, and sped up the compilation time, as only the modified files are recompiled, at the expense of increasing compilation complexity. This complexity comes from the interdependencies among the CSL files that are caused by using instances of externally defined concepts in templates.

A pre-pass over all CSL files maintains a database tracking automatically the signatures of all concepts. This database allows to reconstruct the global dependency structure, which can be used to determine whether other concepts need to be also recompiled because of dependencies.

First results

Currently there is an initial set of 60 concepts implemented in the SPARAMAT system, covering most of the simpler concepts and some of the advanced concepts, such as CSR matrix–matrix multiplication.

Some statistical evaluation of the currently existing CSL files, regarding the relation of the size of the generated C++ code to the length of the original CSL code, the speed of matching, or the number of IR nodes matched for selected example programs are reported in a forthcoming technical report of the University of Trier. We summarize here the most important observations.

The generated C++ files are longer than the original CSL files by a factor that varies between 3 and 25, depending on the complexity of the concept specifications. The ratio is approximately the same for the number of lines as well as for the number of characters. This shows that hand-written pattern matching code is impractical for matching nontrivial templates.

For the more complicated example codes considered, such as CSR matrix–matrix multiplication codes and biconjugate gradient solver codes, the recognition rate is close to 100%, where the recognition rate is determined as the number of matched IR nodes divided by the total number of IR nodes that are expected to be recognized by the system.[18]

[18]Unfortunately, a large-scale evaluation of the current SPARAMAT implementation was not yet possible, because, due to technical problems with the underlying Polaris system, several fundamental transformations such as loop distribution have not been implemented and performed manually. The implementation of SPARAMAT, which was just becoming usable enough to deliver first results, had to be terminated when the programmer who worked exclusively on the implementation, Craig Smith, left the University of Trier in September 2000, thus the preliminary results with the current status of the prototype must suffice. Due to manpower restrictions, a continuation of the implementation in Trier is not possible; interested readers can get the entire implementation from the author.

3.7 Related work

Several automatic concept comprehension techniques have been developed over the last years. These approaches vary considerably in their application domain, purpose, method, and status of implementation.

General concept recognition techniques for scientific codes have been contributed by Snyder [Sny82], Pinter and Pinter [PP91], Paul and Prakash [PP94], diMartino [MI96] and Keßler [C4,C5,J2]. Some work focuses on recognition of induction variables and reductions [AH90, PE95, GSW95] and on linear recurrences [Cal91, RF93]. General techniques designed mainly for nonnumerical codes have been proposed by Wills et al. [RW90] and Ning et al. [HN90, KNS92].

Concept recognition has been applied in some commercial systems, e.g. EAVE [Bos88] for automatic vectorization, or CMAX [SW93] and a project at Convex [Met95] for automatic parallelization. Furthermore there are several academic applications, by the author [D2,J2] and diMartino [MI96], and proposals for application [BS87, BHRS94] of concept recognition for automatic parallelization. Today, most commercial compilers for high-performance computers are able to perform at least simple reduction recognition automatically.

Most specification languages for generated pattern matching systems that have been developed so far, such as OPTRAN [Wil79, LMW88], PUMA [Gro92], TRAFOLA [HS93, Fer90], and TXL [CC92, CC93], as well as systems for retargetable code generation based on tree parsing, such as [GG78], Twig [AG85, AGT89], [PG88] or IBURG [FHP92a, FH95], are restricted in their scope of applicability, namely to expression tree rewriting. The reason for this is probably that these systems have mainly been developed for structural optimizations of expression trees, such as exploiting distributivity of operators or operator strength reduction, and for retargetable code generation for basic blocks represented by expression trees or DAGs (cf. Chapter 2). In contrast, our approach is applicable to pattern matching beyond the expression level, especially to loops and statement sequences.

The PAP Recognizer [MI96] uses Prolog rules to describe templates and relies on the backtracking inference engine of Prolog to perform the pattern matching. Hence, the PAP Recognizer can be regarded rather as an interpreter for a pattern matching language, while SPARAMAT actually compiles the concept specifications to C++ code.

A more detailed survey of these approaches and projects can be found in [D2,J2,J6].

Our former PARAMAT project (1992–94) [D2,J2,J6] kept its focus on dense matrix computations only, because of their static analyzability. The same decision was also made by other researchers [PP91, MI96, Met95, BHRS94] and companies [SW93] investigating general concept recognition with the goal of automatic parallelization. According to our knowledge, there is currently no other framework that is actually able to recognize sparse matrix computations in the sense given in this chapter.

3.8 Future Work

3.8.1 More Concepts, Templates, Example Programs

Adding further concepts and templates specifications to the existing set of CSL files would be desirable to allow for a larger-scale evaluation of the recognition rate. This is now just a matter of time and manpower, as the generative approach simplifies the specification considerably.

3.8.2 Interactive Program Comprehension

An interactive environment for the SPARAMAT system would yield many benefits. The SPARAMAT user could, in an incremental process, teach the system to match further concepts and thereby update its permanent concept database. A key component of such an interactive system is a suitable user interface that supports entering new concept specifications. This could be realized as a pop-up window containing a blank concept specification form where SPARAMAT could automatically pre-enter some template specification text that it derives from the latest matching failure in a previous run on an example code.

3.8.3 Application to Automatic Parallelization

The optimization of the runtime tests and a back-end for the automatic generation of parallel code for the speculatively matched program parts has not been implemented, as this seems to be a more or less straightforward task, at least for shared-memory parallel target architectures.

However, when faced with distributed memory target architectures, we will have to address automatic array distribution and redistribution. It seems to be a reasonable idea to integrate runtime distribution schemes for sparse matrices [UZSS96] into this framework; for instance, the runtime analysis of the sparsity pattern, which is required for load-balancing partitioning of the sparse matrix, may be integrated into the runtime tests.

Beyond automatic parallelization, our automatic program comprehension techniques for sparse matrix codes can also be used in a nonparallel environment, such as for program flow analysis, for program maintenance, debugging support, and for more freedom of choice for a suitable data structure for sparse matrices.

3.8.4 Applicability to Other Problem Domains

SPARAMAT was intended for and applied to the problem domain of linear algebra operations on sparse matrices. Nevertheless, the principles and techniques of SPARAMAT are not specific to this particular domain. We expect that other areas could benefit from the SPARAMAT approach. Investigating how similar program comprehension techniques could be applied to fields like image processing or operations on dynamic data structures could provide important insights.

3.9 Summary

We have described a framework for applying program comprehension techniques to sparse matrix computations and its implementation. We see that it is possible to perform speculative program comprehension even where static analysis does not provide sufficient information; in these cases the static tests on the syntactic properties (pattern matching) and consistency of the organizational variables are complemented by user prompting or runtime tests whose placement in the code can be optimized by a static data flow framework.

The SPARAMAT implementation, consisting of the driver, the CSL parser with the generator, and a set of auxiliary library routines and tools, is operational. Virtually all basic concepts and many of the concepts listed in Section 3.3 along with the most important templates have been implemented in CSL. The generative approach allows for a fast implementation of further concepts and templates also by other users.

Acknowledgements

The author thanks Prof. Dr. Emilio L. Zapata for an early discussion about the applicability of program comprehension techniques to sparse matrix computations. Prof. Dr. Helmut Seidl contributed some suggestions on specific recognition techniques. Useful comments after presentation of early versions of this work, such as some pointers to related literature, came from the audience at the workshops AP'97 at UPT Barcelona, EuroPar'97 in Passau, and CPC'98 in Vadstena.

Part of the material presented in this section has been developed within the SPARAMAT project, which was funded by the Deutsche Forschungsgemeinschaft (DFG) from October 1997 to September 2000. During this period, Craig Smith was responsible for the implementation of the SPARAMAT system on top of the open-source Polaris compiler.

The SPARAMAT implementation is available to interested readers upon request.

Chapter 4

Design of Parallel Programming Languages

MIMD programming languages constitute a more flexible alternative to SIMD languages. Imperative MIMD languages can be classified into mainly two different categories, depending on the execution style, that is, the way how parallel activities are created in a program.

The *fork-join style* of parallel execution corresponds to a tree of processes. Program execution starts with a sequential process, and any process can spawn arbitrary numbers of child processes at any time. While the fork-join model directly supports nested parallelism, the necessary scheduling of processes requires substantial support by a runtime system and incurs overhead.

In contrast, the languages Fork, ForkLight, and NestStep presented in this chapter all follow the *SPMD style* of parallel execution. There is a constant set of threads that are to be distributed explicitly over the available tasks, e.g. parallel loop iterations. Given fixed sized machines, SPMD seems better suited to exploit the processor resources economically. Nevertheless, SPMD languages can be designed to provide a similar support of nested parallelism as is known from the fork-join style languages. This chapter focuses on SPMD-style MIMD languages with support for nested parallelism.

Furthermore, parallel programming languages may be classified according to the abstract machine model they are based on. This concerns, for instance, the memory organization, the memory consistency scheme, or the degree of “natural” synchronization between the processors. An accompanying cost model can give the programmer or the compiler hints on the complexity of certain operations that may be useful to optimize the program. In this chapter, we investigate programming languages for three different machine models.

The chapter begins with a presentation of the general processor group concept in Section 4.1. This mechanism allows for the realization of nested parallelism and more flexibility in SPMD languages. Then, we present the designs of the three languages Fork, ForkLight, and NestStep in detail.

Fork, previously also called Fork95, is a C-based, parallel programming language for the PRAM model. The PRAM model provides a sequentially consistent shared memory. The instruction-level synchronous execution of the code by the PRAM processors is made transparent to the programmer at the language operator level. The scope of synchronous execution and of sharing of variables is controlled by the processor group concept. Even in program regions where processors are allowed to proceed asynchronously, the PRAM-typical sequential memory consistency is maintained. A compiler for the experimental massively parallel computer SB-PRAM is available. Fork is mainly used for teaching parallel programming classes and for research on the practicality of PRAM algorithms from the literature.

The description of Fork, given in Section 4.2, is followed by a plea for structured parallel programming in Fork; Section 4.3 shows how the support for nested parallelism in Fork can be exploited to implement a set of skeleton-style generic functions that allow for a structured way of parallel programming.

Section 4.4 describes ForkLight [C16], a language for the Asynchronous PRAM model [Gib89, CZ89] that abstracts from synchronous execution within basic blocks by replacing the synchronous mode by the so-called *control-synchronous mode* of execution. Where group-wide barriers or bilateral synchronization is necessary within a basic block to protect instructions with cross-processor data dependences from race conditions, the programmer must provide these explicitly. Nevertheless, ForkLight assumes a sequentially consistent shared memory, such as provided in the Tera MTA [ACC$^+$90]. The syntax of ForkLight is very similar to Fork. ForkLight is implemented by a precompiler (ForkLight to C) and a runtime system based on portable shared memory interfaces like P4 [BL92, BL94] or OpenMP [Ope97].

Section 4.5 presents the design of NestStep [C18], a language for the bulk-synchronous parallel (BSP) model [Val90], which additionally abstracts from sequential memory consistency but retains a shared address space by a software–emulated distributed shared memory. NestStep programs are structured in supersteps. Supersteps can be nested, which enables group splitting and accordingly narrows the scope of barriers and shared memory consistency to subgroups. NestStep is implemented by a precompiler (NestStep to Java resp. to C) and a runtime system that is based on a message passing interface like Java socket communication [Far98] or MPI [SOH$^+$96].

Both ForkLight and NestStep are MIMD languages, follow the SPMD style, and support nested parallelism by a hierarchical group concept that controls the scope of sharing and barriers, as in Fork. Hence, programming in these languages is quite straightforward if there is some familiarity with Fork.

We close this chapter with a review of related parallel programming languages in Section 4.6. Implementation issues for the three languages will be considered later in Chapter 5.

We would like to emphasize that the sections of this chapter are *not* intended as tutorials for the corresponding programming languages. Instead, an in-depth introduction to programming in Fork can be found in the chapters 5 and 7 of *Practical PRAM Programming* [B1]. Accordingly, we omit the description and implementation of larger example problems for sake of brevity.

4.1 The SPMD Processor Group Concept

In order to support nested parallelism in a SPMD programming language, statements are executed by *groups* of processors, rather than by individual processors. Initially, there is just one group containing all p available processors. Groups may be statically, dynamically and

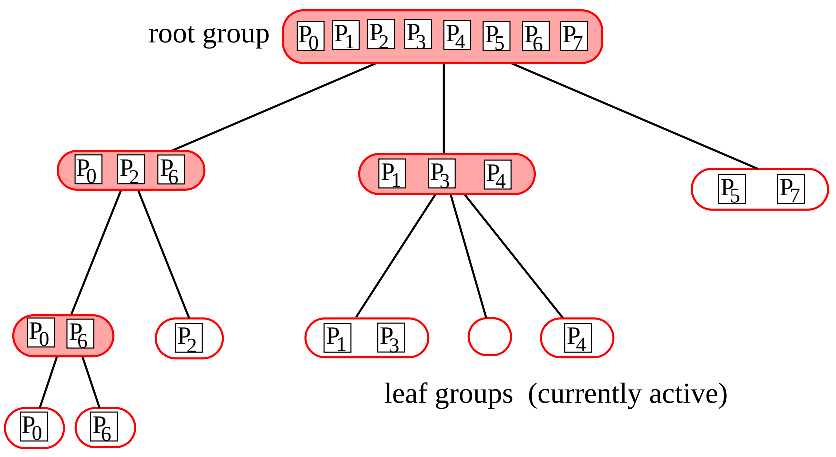


FIGURE 4.1: The group hierarchy in Fork forms a logical tree. Nonactive groups are shaded.

even recursively subdivided into subgroups. A subgroup-creating language construct is called *group-splitting* if it may create $k > 1$ subgroups. Where a group G with p processors is subdivided into $k \geq 1$ subgroups $G_1, \ldots, G_k$, the group G is temporarily deactivated. Each of its processors either joins one of the subgroups to perform a computation with that subgroup being its active group, or skips the subgroups' computation completely and waits for the other processors at the program point where G is reactivated. When all subgroups have finished their computation, they cease to exist, and then G is reactivated, typically after an implicit, G-wide barrier synchronization, and continues with its computation. Hence, at any point of program execution, all presently existing groups form a treelike[1] *group hierarchy*. Only the *leaf groups* of the group hierarchy are active (see Figure 4.1). A processor belongs to at most one active group at any time, and to all (inactive) predecessors of that group in the group hierarchy tree.

In Fork, ForkLight, and NestStep, the group concept is used for controlling different invariants regarding the degree of synchronicity, sharing of variables, and memory consistency. Nevertheless, there are several common organizational properties of the groups in these languages.

A processor can inspect the number of processors belonging to its current group, either by calling a standard library routine or by reading the predefined constant shared integer variable `#`.

Each group has a (shared) integer group ID, which is primarily intended to distinguish the subgroups of a group. The group ID of the root group is 0. Group-splitting constructs set the subgroup IDs consecutively from 0 to the number of subgroups minus one. At the other subgroup-creating constructs, the subgroup inherits the group ID of its parent group. A processor can inspect the group ID of its current group by the predefined constant shared integer variable `@`.

Within a group, processors are ranked automatically from 0 to the group size minus one.

[1]If `join` statements (see Section 4.2.9) are executed, the group hierarchy forms a *forest* rather than a tree. As this difference is not important for the following discussions, we omit it for simplicity of presentation.

This *group rank* can be inspected by the predefined constant private integer variable $$. In some cases the programmer needs a user-defined numbering of processors that does not change, even if some processors leave the group. For these cases, there is the *group-relative processor ID*, which is accessible via the predefined private integer variable $. At program start, $ is initialized to the processor's rank in the root group. At all subgroup-creating constructs, the $ value of the subgroups is just inherited from the parent group; the programmer is responsible to renumber them explicitly if desired.

The group ID @ and the group-relative processor ID $ are automatically saved when deactivating the current group, and restored when reactivating it.

There may be regions in the code where the group concept is temporarily disabled. Such regions are marked explicitly in the program code. If processors belonging to a currently active group G enter such a region r, G is deactivated while the processors remains inside r. Hence, the properties controlled by the group concept are not available within r. At exit of r, G is reactivated.

The *activity region* of a group G is the set of statements that may be executed by the processors of G when G is their currently active group. This includes all statements from the creation of G to the termination of G, excluding the activity regions of subgroups of G and, if existing, interspersed regions where the group concept is temporarily disabled.

The *live region* of a group G denotes all statements where G may exist, either active or inactive. Hence, it can be defined recursively as the union of the activity region of G, interspersed regions in the activity region of G where the group concept is disabled, and the live regions of all subgroups of G.

The activity region of a group can be structured as a sequence of *activity blocks* that are separated by implicit or explicit group-wide barriers. The positions of implicit barriers are defined by the respective programming language, and explicit barriers can be inserted by the programmer at virtually any point in the program.

4.2 Fork Language Design

4.2.1 The PRAM model

The Parallel Random Access Machine (PRAM) [FW78] is an abstract machine model for the execution of parallel programs. It mainly consists of a set of processors, which are Random Access Machines (RAMs) as known from the analysis of sequential algorithms, and of a large shared memory. The processors receive a common clock signal and thus operate synchronously on the machine instruction level. Each processor has its own program counter and may follow an individual trace of computation in the program memory. All processors are connected to the shared memory, which may be accessed by any processor in any machine cycle. Each cell may be either read or written by an arbitrary number of processors in each machine cycle. Each memory access takes the same time, ideally one machine cycle, for any processor and any memory location. The shared memory is sequentially consistent; there is no implicit caching of values. The processors may have optionally also have local memory cells, such as processor registers. Figure 4.2 shows a block diagram of a PRAM.

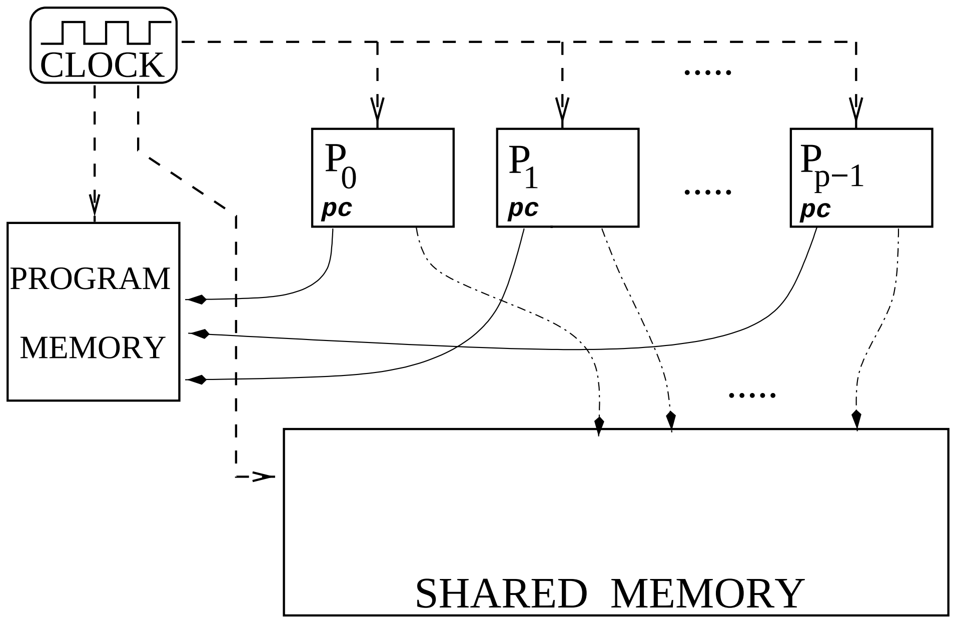


FIGURE 4.2: The PRAM model.

Based on the resolution of write conflicts that occur when several processors try to access the same memory location, we distinguish between three basic variants of a PRAM:

- the EREW PRAM (Exclusive Read, Exclusive Write): in each cycle, a memory cell may be either read by at most one processor, or written by at most one processor.
- the CREW PRAM (Concurrent Read, Exclusive Write): in each cycle, a memory cell may be either read by several processors, or written by at most one processor.
- the CRCW PRAM (Concurrent Read, Concurrent Write): in each cycle, a memory cell may be either read by several processors, or written by several processors.

The CRCW (or short: CW) PRAM is obviously the strongest of these three forms. Clearly, simultaneous write accesses to the same memory location in the same cycle require another arbitration mechanism. Consequently, the CRCW PRAM is, depending on the resolution of concurrent write accesses, subdivided into several subvariants. The most important ones are the following, listed in ascending order of computational power:

- the COMMON CRCW PRAM: if several processors write to the same memory cell at the same cycle, they must write the same value.
- the ARBITRARY CRCW PRAM: if several processors try to write (different) values into the same memory cell at the same cycle, then one of the values is selected arbitrarily and written.

- the PRIORITY CRCW PRAM: if several processors try to write (different) values into the same memory cell at the same cycle, then the value of the processor with least processor ID is written.
- the COMBINE CRCW PRAM: if several processors try to write (different) values into the same memory cell at the same cycle, a combination of all values to be written and of the former content of that memory cell will be written. The combination may be a global sum, a global maximum or minimum, or a logical or bitwise boolean operator (OR, AND).
- the PREFIX CRCW PRAM: like the Combine CRCW PRAM; additionally, the prefix sums, prefix maximum, etc., in the order of processor ranks, can be obtained as a byproduct of a concurrent memory access.

The PRAM model has been very popular in the theory of parallel algorithms in the last 20 years because it abstracts from many architectural details that are typically encountered in real machines, such as distributed memory modules, the cost of memory access, nonuniform or even unpredictable memory access times, individual clock signals for the processors, weaker memory consistency schemes, etc. The PRAM has thus often been criticized as being highly unrealistic, and other programming models that account for the cost of memory access or that do not rely on synchronous execution have been proposed. For the same reason, a physical realization of the PRAM was not available for a long time. A research project in the 1990s at the University of Saarbrücken, Germany, led by Prof. Wolfgang Paul, realized a massively parallel PREFIX CRCW PRAM in hardware: the SB-PRAM ([AKP91, ADK$^+$93]; see also [B1, Chap. 4]). Although being no longer competitive with current supercomputer designs, the SB-PRAM opened a path for research on the practical aspects of PRAM programming, which resulted in the development of system software, tools, programming languages and compilers, libraries, and applications. The SB-PRAM architecture, system software, and the PRAM-specific parts of the programming environment and libraries are described in *Practical PRAM Programming* [B1]. That book also contains an in-depth treatment of PRAM theory [B1, Chap. 2], which is beyond the scope of this thesis.

4.2.2 Fork Language Design Principles

Fork is a high-level programming language for PRAMs. Its programming model, summarized in Figure 4.3, is identical to the Arbitrary CRCW PRAM programming model, with the minor exception that the private memory of each processor is embedded as a private address subsection into the shared memory. This modification enables treating private and shared addresses similarly at the language level, which keeps pointer declarations simple.

Fork is based on ANSI C [ANS90]. The language constructs added allow to control parallel program execution, such as by organizing processors into groups, managing shared and private address subspaces, and handling various levels of synchronous parallel program execution. Because of carefully chosen defaults, existing (sequential) C codes can be easily integrated into Fork programs. Furthermore, powerful multiprefix operations are available

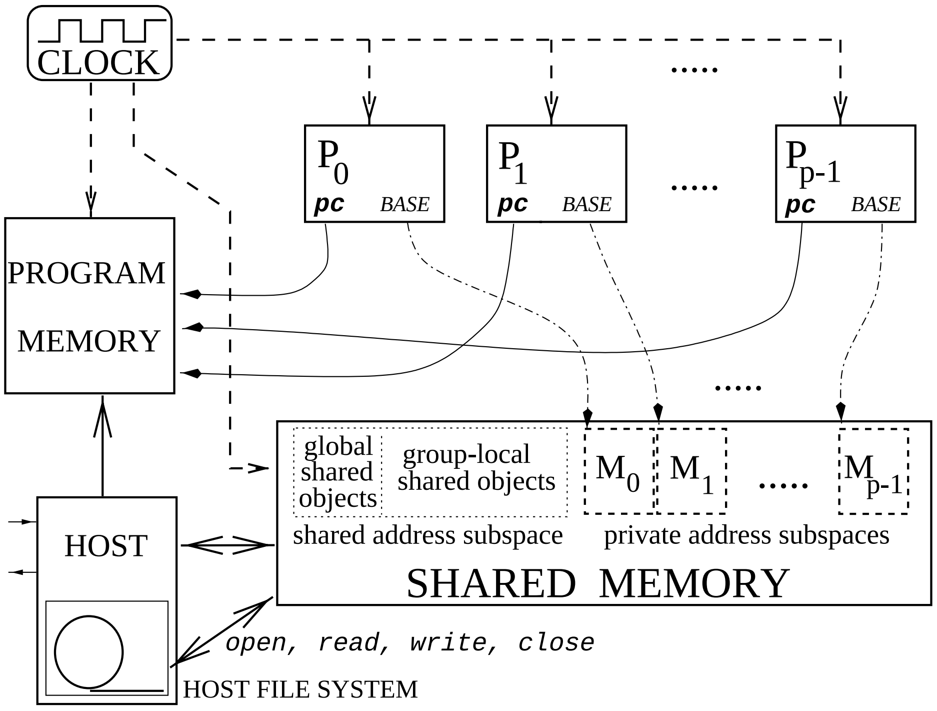


FIGURE 4.3: Block diagram of Fork's programming model. Fork assumes that private address subspaces are embedded into the shared memory by handling private addresses relative to a BASE pointer. Beyond preselecting the private subspace size, this mechanism is implicitly handled by the implementation and not accessible to the programmer at the language level. The other features are the same as for the theoretical PRAM model; all processors receive the same clock signal, thus the machine is synchronous on the machine instruction level. The memory access time is uniform for each processor and each memory location. Each processor works on the same node program. Access to the PRAM is possible only via a user terminal running on a host computer that is invisible for the programmer. The job of the host is to load code to the program memory, start the processors, and perform external I/O. The host computer's file system is accessible to each PRAM processor by specific I/O routines. The differences in addressing (BASE-relative vs. absolute addresses) are handled automatically by the compiler and are not visible to the programmer.

as language operators. In short, Fork enables to exploit the specific features of PRAMs and makes them transparent to the programmer at the language level.

In contrast to most other MIMD programming languages, the philosophy of Fork is *not* to start with a sequential thread of computation and successively spawn new ones (which is known as the *fork-join style* of parallel program execution) but to take the explicitly given set of available processors—which is active from the beginning and remains constant during parallel program execution—and distribute them explicitly across the work to be done. This parallel execution model is known under the term *single program, multiple data* (SPMD). Hence, the `main()` function of a Fork program is executed by all available processors of the PRAM partition on which the program has been started. This number of available processors can be inspected at runtime by the global shared constant integer variable `__STARTED_PROCS__`. Moreover, each processor holds a global private constant integer variable `__PROC_NR__` that contains its processor ID at run time. These processor IDs are consecutively numbered from 0 to `__STARTED_PROCS__` minus 1.

Each processor works on exactly one process all the time. The number of processors available in a Fork program is inherited from the underlying PRAM hardware resources; an emulation of additional (logical) PRAM processors by multithreading in software is not supported for good reason. As we will see later in Section 5.2.1, it turns out that the implementational overhead to maintain *synchronously* executing logical processors in software is a too high price to pay for a potential gain in comfort of programming. On the other hand, execution of an arbitrary number of asynchronous tasks can be easily implemented in Fork, as we have exemplified in [B1, Chap. 7].

Fork programs are statically partitioned into block-structured regions of code that are to be executed in either synchronous or asynchronous mode. For instance, functions are classified as either synchronous, straight, or asynchronous.

In *synchronous mode* processors operate in groups. Each processor belongs to exactly one active group. By its *strict synchronicity invariant,* Fork guarantees that all processors of the same active group execute the same instruction at the same time. In other words, their program counters are equal at any time. Hence, the exact synchronicity available at the PRAM machine instruction level is made available at the source language level: From the Fork programmer's point of view, all processors of the same active group execute simultaneously the same statement or expression operator.

In general, the strict synchronicity invariant eliminates "race conditions"; that is, it enables deterministic behavior of concurrent write and read accesses to shared memory locations at the language's operator level without additional protection mechanisms such as locks and semaphores.

Where control flow may diverge as a result of private branch conditions, the current group is automatically split correspondingly into subgroups; exact synchronicity is then maintained only within each subgroup. The subgroups are active until execution of all branches is finished and control flow is reunified; then the parent group is reactivated and the strict synchronicity invariant for it is reinstalled automatically. Group splitting can be nested statically and dynamically; hence, at any time of program execution, the group hierarchy forms a logical tree.

In *asynchronous mode*, this strict synchronicity invariant is not maintained. Hence, the processors may follow individual control flow paths. Asynchronous execution is unavoidable in some cases; for instance, all I/O routines are asynchronous. In some cases, asynchronous execution allows for a better utilization of processors, as we will see in Section 4.2.8, for instance.

The remainder of this section on the Fork language design is organized as follows: Section 4.2.3 introduces shared and private variables, and Section 4.2.4 discusses the special expression operators in Fork. Section 4.2.5 introduces the language constructs that allow to switch properly from synchronous to asynchronous mode and vice versa. Section 4.2.6 discusses the synchronous mode of execution in detail. It explains the group concept, describes how the strict synchronicity invariant is relaxed automatically by splitting a group into subgroups, and shows how this feature can be used to implement synchronous parallel divide-and-conquer algorithms. Like C, Fork offers pointers and heaps, which are discussed in Section 4.2.7. The default mode of parallel program execution is the asynchronous mode. Section 4.2.8 introduces the language constructs and standard library routines provided by Fork for asynchronous programming, and explains how they may be used to protect critical sections. Section 4.2.9 introduces the `join` statement that allows to switch from asynchronous to synchronous mode more flexibly. We give some recommendations concerning programming style and common pitfalls in Section 4.2.10. Section 4.2.11 introduces the `trv` tool for graphical visualization of Fork program traces, which can be used for performance analysis.

More example programs written in Fork will follow in Section 4.3. The compilation aspects of Fork are discussed in Chapter 5.

4.2.3 Shared and Private Variables

As mentioned above, the entire shared memory of the PRAM (see Figure 4.3) is partitioned—according to the programmer's wishes—into private address subspaces (one for each processor) and a shared address subspace, which may be again dynamically subdivided among the different processor groups. Accordingly, program variables are classified as either *private* or *shared*. The term *shared* always relates to the processor group that defined that variable, namely, the group that the processor belonged to when it executed that variable's declaration. For each variable, its *sharity*—an attribute indicating whether the variable is shared or private—is to be declared as a qualifier of the variable's storage class.

Shared variables declared globally (i.e., outside of functions) are shared by all processors executing the program. Shared variables that occur as formal parameters of a function or that are declared locally at the top level of a function exist once for each group executing this function. Finally, block local variables declared to be shared exist once for each group executing that block.

For a variable declared to be private, one copy resides in the private address subspace of each processor that executes the declaration.

Declaration

To declare a global shared integer variable `v`, we write

```
sh int v;
```

Here `sh` is a storage class qualifier indicating that the *sharity* of `v` is "shared." The sharity may also be specified after the type declaration, but it is convenient programming style to specify it first.

Fork accepts only new-style declarations of functions; that is, the complete parameter types are to be specified between the parentheses. Shared formal parameters can only occur in synchronous functions (see Section 4.2.5). For instance, a shared formal parameter of a synchronous function may be declared as follows:

```
sync void foo( sh float x );
```

The sharities of the formal parameters must match at redeclarations; they are part of the function's type.

Shared local variables may be declared only in synchronous program regions, such as at top level of a synchronous function.

Private variables are declared as follows:

```
pr int w;
int q;
```

where the sharity declarator `pr` is redundant since "private" is the default sharity. For simplicity, we will mostly omit `pr` in declarations of private variables, except for special emphasis.

System Variables

We already mentioned that the *total number of started processors* is accessible through the constant shared integer variable

```
__STARTED_PROCS__
```

and the *processor ID* of each processor is accessible through the constant private integer variable

```
__PROC_NR__
```

With these, it becomes straightforward to write a simple dataparallel loop:

```
#include <fork.h>
#define N 30
sh int sq[N];
sh int p = __STARTED_PROCS__;

void main( void ) {
  pr int i;
  for (i=__PROC_NR__; i<N; i+=p)
     sq[i] = i * i;
  barrier;
}
```

Since the processor IDs are numbered consecutively from 0 to $p-1$, the private variable `i` on processor j loops over the values j, $j+p$, $j+2p$, and so on. Thus, this loop computes in the shared array `sq` all squares of the numbers in the range from 0 to N minus 1.

There are special private variables `$` and `$$` and a special shared constant `@` for each group; these will be introduced in Section 4.2.6.

Concurrent-Write Conflict Resolution

Fork supports the Arbitrary CRCW PRAM model. This means that it is not part of the language to define which value is stored if several processors write the same (shared) memory location in the same cycle. Instead, Fork inherits the write conflict resolution method from the target machine. In the case of the SB-PRAM, which implements a priority concurrent-write conflict resolution, the processor with highest hardware processor ID[2] will win and write its value. Hence, concurrent write is deterministic in our implementation of Fork. However, meaningful Fork programs should not be dependent on this target-machine-specific conflict resolution scheme. For practically relevant applications of concurrent write, the multiprefix instructions (see Section 4.2.4) are often better suited to obtain a portable program behavior.

4.2.4 Expressions

In addition to the expression operators known from C, Fork provides several new expression operators: the four binary multiprefix operators `mpadd()`, `mpmax()`, `mpand()` and `mpor()`, and the `ilog2()` operator.

We also introduce the terms *shared expression* and *private expression* in this section.

Atomic Multiprefix Instructions

The SB-PRAM supports powerful built-in multiprefix instructions called `mpadd`, `mpmax`, `mpand` and `mpor`, which allow the computation of multiprefix integer addition, maximization, bitwise AND and bitwise OR for up to 2048 processors within two CPU cycles. As they are very useful in practice, these multiprefix instructions are available in Fork.[3] The multiprefix instructions are accessible as Fork language operators rather than standard library functions, in order to allow for potential optimizations and to avoid the function call overhead.

For instance, assume that the statement

```
k = mpadd( ps, expression );
```

is executed simultaneously by a set P of processors (e.g., a group). `ps` must be a (potentially private) pointer to a shared integer variable, and `expression` must be an integer expression. In this way, different processors may address different `sharedvar`s, thus allowing for multiple `mpadd` computations working simultaneously. Let $Q_s \subseteq P$ denote the subset of

[2]The hardware processor IDs of the SB-PRAM are not numbered consecutively. Also, because of a hardware design bug, the hardware processor IDs in the current SB-PRAM prototype are *different* from the processor ranks in the multiprefix operations. Since the latter ones are more interesting for the programmer, we have chosen this multiprefix rank as the processor ID `__PROC_NR__` in our implementation of Fork. Thus `__PROC_NR__` does generally *not* reflect the concurrent write priority! Instead, the hardware ID of a processor can be inspected by a call to the library function `getnr()`.

[3]Note, however, that the implementation of multiprefix operations like `mpadd` that can be used deterministically in synchronous execution mode may lead to extremely inefficient code when compiled to other (non-priority-based) PRAM variants, because the (conceptually) fixed linear internal access order among the processors (which is equivalent to the processor IDs in Fork) must be simulated in software where hardware support is missing. Fortunately, the programmer relies quite rarely on this fixed order. One example is the integer parallel prefix routine `prefix_add` of the PAD library [B1, Chap. 8].

the processors in P whose pointer expressions `ps` evaluate to the same address s. Assume the processors $q_{s,i} \in Q_s$ are subsequently indexed in the order of increasing processor ID `__PROC_NR__`.First, each processor $q_{s,i}$ in each Q_s computes `expression` locally, resulting in a private integer value $e_{s,i}$. For each shared memory location s addressed, let v_s denote the contents of s immediately before executing the `mpadd` instruction. Then, the `mpadd` instruction simultaneously computes for each processor $q_{s,i} \in Q_s$ the (private) integer value

$$v_s + e_{s,0} + e_{s,1} + \cdots + e_{s,i-1}$$

which is, in the example above, assigned to the private integer variable `k`. Immediately after execution of the `mpadd` instruction, each memory location s addressed contains the global sum

$$v_s + \sum_{j \in Q_s} e_{s,j}$$

of all participating expressions. Thus, `mpadd` can as well be "misused" to compute a global sum by ignoring the value of `k`.

However, if we are interested only in this side effect, there are better alternatives, namely, the library routines `syncadd`, `syncmax`, `syncand`, and `syncor`, the equivalents of the SB-PRAM instructions with the same name. They take the same parameters as the corresponding multiprefix operators but do not return a value. If these should be scheduled only at a specific value of the modulo bit[4] there are customized variants `syncadd_m0`, `syncmax_m0`, `syncand_m0`, and `syncor_m0`, which are scheduled only to cycles where the modulo bit is zero, and `syncadd_m1`, `syncmax_m1`, `syncand_m1`, and `syncor_m1`, which are scheduled only to cycles where the modulo bit is one.

For instance, the number p of currently available processors can be found out by synchronously executing

```
sh int p = 0;
syncadd( &p, 1 );
```

This may be extended in order to obtain a consecutive renumbering $0,1,\ldots,p-1$ stored in a user-defined processor ID `me`:

```
sh int p = 0;
pr int me = mpadd( &p, 1 );
```

Similarly to `mpadd`, the atomic `mpmax()` operator computes the prefix maximum of integer expressions, `mpand()` the prefix bitwise AND, and `mpor()` the prefix bitwise OR.

[4]The modulo bit is a globally visible flag in the SB-PRAM that toggles after each clock cycle. It contains the least significant bit of the global cycle counter. It is important for the assembler programmer (and the compiler) of the SB-PRAM to avoid simultaneous access of a memory location with different access types (like load and store), which is forbidden in the SB-PRAM.

The `ilog2` Operator

The unary `ilog2()` expression operator [5] is applicable to positive integer operands. It computes the floor of the base 2 logarithm of its argument. For example, `ilog2(23)` evaluates to 4.

Return Values of Functions

The return value of a non-`void` function is always private; each processor passes its own return value back to the function call.

If a private value, such as the return value of a function, should be made shared among the processors of the calling group, this can be simply arranged by assigning it immediately to a shared variable. For instance, in

```
sq[N-1] = square( N-1 );
```

the private result value returned by function `square()` is assigned to the shared array element `sq[N-1]` that is accessible to all processors, following the default concurrent-write policy.

Shared and Private Expressions

We call an *expression* to be *shared* if it is guaranteed to evaluate to the same value on each processor of a group if evaluation starts synchronously, and *private* otherwise.

In Fork, an expression must be conservatively assumed to be private if it contains a reference to a private variable, `$` or `__PROC_NR__`, a multiprefix operator, or a function call. All other expressions are shared.

Note that any other processor that does not belong to the group under consideration may at the same time modify any shared variable s occurring in a shared expression e. Nevertheless, this does not compromise the abovementioned condition of evaluating e to the same value on all processors for that group. Because of the synchronous expression evaluation within the group, either all processors of the group use the old value of s, or all of them use the modified value of s.

4.2.5 Synchronous and Asynchronous Regions

Fork offers two different program execution modes that are more or less statically associated with source code regions: synchronous execution mode and asynchronous execution mode.

In *synchronous execution mode*, processors remain synchronized on the statement level and maintain the *strict synchronicity invariant*, which says that in synchronous mode, all processors belonging to the same (active) group operate strictly synchronous, that is, their program counters are equal at each machine cycle. The group concept will be discussed in more detail in Section 4.2.6.

[5]On the SB-PRAM, `ilog2()` is implemented by the one-cycle `rm` instruction.

```
sync int *sort( sh int *a, sh int n )
{
  extern straight int compute_rank( int *, int);
  if ( n>0 ) {
     pr int myrank = compute_rank( a, n );
     a[myrank] = a[__PROC_NR__];
     return a;
  }
  else
     farm {   /* enter asynchronous region */
        printf("Error: n=%d\n", n);
        return NULL;
     }
}
extern async int *read_array( int * );
extern async int *print_array( int *, int );
sh int *A, n;
async void main( void )
{
  A = read_array( &n );
  start {   /* enter synchronous region */
     A = sort( A, n );
     seq  if (n<100) print_array( A, n );
  }
}
```

FIGURE 4.4: A Fork program is statically partitioned into synchronous (dark dashed boxes), asynchronous (light solid boxes) and straight regions.

In asynchronous execution mode, the strict synchronicity invariant is not enforced. The group structure is read-only; new shared variables and objects on the group heap cannot be allocated. There are no implicit synchronization points. Synchronization of the current group can, though, be explicitly enforced using the `barrier` statement.

The functions and structured blocks in a Fork program, called program *regions* for short, are statically classified as either synchronous, straight, or asynchronous; see Figure 4.4. This static property of the code is called its *synchronicity*. Synchronous regions are to be executed only in synchronous mode, asynchronous regions only in asynchronous mode, and straight regions in synchronous or asynchronous mode (see Table 4.1). In order to guarantee this, straight program regions must not contain any statements that may cause a divergence of control flow, such as `if` statements or loops whose branch condition depends on a private value.

Advantages of the Synchronous Mode

The synchronous mode overcomes the need for protecting shared program objects by locks (see Section 4.2.8) because they are accessed in a deterministic way: The programmer can rely on a fixed execution time for each operation, which is the same for all processors at any time during program execution. Hence, no special care has to be taken to avoid race conditions

	Program region synchronicity		
Execution mode	Synchronous	Straight	Asynchronous
Synchronous	√	√	—
Asynchronous	—	√	√

TABLE 4.1: The relationship between program execution mode and program region synchronicity.

at concurrent accesses to shared program objects. The synchronous mode saves the time and space overhead for locking and unlocking but incurs some overhead to maintain the strict synchronicity invariant during synchronous program execution.

As an example, consider the following synchronous code:

```
sh int a = 3, x = 4;
pr int y;
...
  if (__PROC_NR__ % 2)  a = x;
  else                  y = a;
...
```

Here, the processors with odd processor ID execute the `then` branch of the `if` statement, the other ones execute the `else` branch. While it is, for the processors executing the `else` branch, not determined whether `y` will finally have the old (3) or the new (4) value of `a`, it is nevertheless guaranteed by the synchronous execution mode that all these processors assign to their private variable `y` the *same* value of `a`. A lock to guarantee this is not required.

Advantages of the Asynchronous Mode

In asynchronous or straight program regions, there are no implicit synchronization points. Maintaining the strict synchronicity invariant in synchronous regions requires a certain overhead that is incurred also for the cases where exact synchronicity is not required for consistency because of the absence of data dependencies. Hence, marking such regions as asynchronous can lead to substantial savings. For instance, during the implementation of the (mostly synchronous) routines of the PAD library [B1, Chap. 8], we found considerate usage of internally asynchronous execution to pay off in significant performance improvements. Also, asynchronous computation is necessary for dynamic load balancing, such as in dynamic loop scheduling (see Section 4.2.8).

Declaring the Synchronicity of a Function

Functions are classified as either synchronous (declared with type qualifier `sync`), straight (declared with type qualifier `straight`), or asynchronous (`async`,) this is the default); for example

```
sync void foo ( void ) { ... }
straight int add ( int a, int b ) { return a+b; }
extern sync int sort ( sh int, sh int [],
                       sh sync int (*cmp)(int,int) );
sh sync int (*spf)( sh int );
async void (*pf)( void );
```

The order of qualifiers for sharity and synchronicity in a declaration is irrelevant. For a pointer to a function, its synchronicity and parameter sharities must be declared and match at assignments. Note that the sharity of a pointer to a function refers to the pointer variable itself, not to the function being called.

`main()` is asynchronous by default. Initially, all processors on which the program has been started by the user execute the startup code in parallel. After that, these processors start execution of the program in asynchronous mode by calling `main()`.

The `farm` Statement

A synchronous function is executed in synchronous mode, except from blocks starting with a `farm`[6] or `seq` statement.

```
farm  <stmt>
```

causes the executing processors to enter the asynchronous mode during the execution of the asynchronous region `<stmt>`, and to reinstall synchronous mode after having finished their execution of `<stmt>` by an implicit group local exact barrier synchronization. During the execution of `<stmt>`, the group is inactive (see Figure 4.5); thus shared local variables or group heap objects cannot be allocated within `<stmt>`.

`farm` statements make sense only in synchronous and straight regions. Using `farm` within an asynchronous region is superfluous, and its implicit barrier may even introduce a deadlock. Hence, the compiler emits a warning if it encounters a `farm` statement in an asynchronous region.

`return` statements are not permitted in the body of a `farm` or `seq` statement.

The `seq` Statement

The `seq` statement

```
seq  <statement>
```

works in a way similar to that of the `farm` statement. Its body `<statement>` is an asynchronous region and is thus executed in asynchronous mode. But, in contrast to `farm`,

[6]The keyword `farm` is due to historical reasons [M4]; it was originally motivated by the intended use for programming independent tasks (*task farming*). From today's point of view, a better choice would be something like `relax` or `sloppy` (*relaxed synchronicity*). Anyway, what's in a name? A programmer who prefers another keyword to replace `farm` may simply use the C preprocessor, for instance.

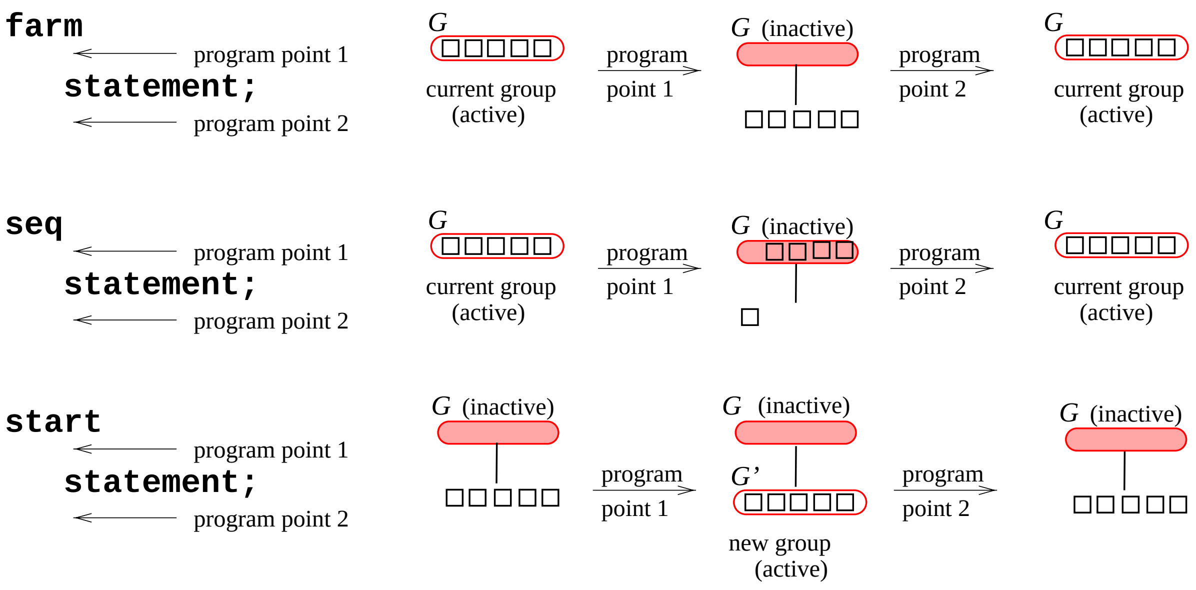


FIGURE 4.5: Visualization of the semantics of `farm`, `seq`, and `start`. Groups are represented by ovals; and processors, by rectangular boxes. Inactive groups are shaded.

`<statement>` is executed by only *one* processor of a group executing the `seq` statement (see Figure 4.5). This processor is selected arbitrarily within the group.

When this processor finishes execution of the `<statement>`, the synchronous mode of execution is reinstalled for the group.

Although, from the programmer's view, only one processor is active within the `seq` body, a clever compiler may try to keep more processors busy by exploiting instruction-level parallelism in `<statement>`.

`seq` statements make sense only in synchronous and straight regions. The compiler emits a warning if it encounters a `seq` statement in an asynchronous region.

The `start` Statement

Asynchronous functions are executed in asynchronous mode, except from blocks starting with the `start` statement[7]

```
start <stmt>
```

The `start` statement, applicable in straight and asynchronous regions, switches to synchronous mode for its body `<stmt>`. It causes *all* available processors to perform an exact barrier synchronization with respect to their last group. At the top level of program execution this is the root group consisting of all started processors. These processors form a new group

[7]Again, the keyword `start` is not the best choice from today's point of view; something like `strict` may be more suitable.

	Synchronicity of the called function[a]		
Caller region	Synchronous	Straight	Asynchronous
Synchronous	$\surd$	$\surd$	Via `farm`/`seq`
Straight	Via `start`	$\surd$	Via `farm`/`seq`
Asynchronous	Via `start`/`join`	$\surd$	$\surd$

TABLE 4.2: Rules for calling functions regarding their synchronicity. The checkmark $\surd$ means that a call is directly possible.

and execute `<stmt>` in synchronous mode, with unique processor IDs `$` renumbered subsequently from 0 to `__STARTED_PROCS__`-1. `<stmt>` is a synchronous region. Shared variables and group heap objects allocated by the former group are no longer visible. After having executed `<stmt>`, the processors restore their previous mode of execution, the previous group structure is visible again.

A generalization of `start`, the `join` statement, provides a very flexible way of entering a synchronous program region from an asynchronous region. It will be described in detail in Section 4.2.9.

Calling Synchronous, Straight, and Asynchronous Functions

There are several rules concerning function calls necessary to maintain this nearly one-to-one correspondence between the (static) synchronicity of a program region and the (dynamic) synchronicity of the mode of its execution, as defined in Table 4.1.

From an asynchronous region, only asynchronous and straight functions can be called. Synchronous functions can be called only if the call is explicitly made a synchronous region by using a `start` or `join` statement.

In the other way, calling an asynchronous function from a synchronous or straight region results in an explicit (via a `farm` or `seq` statement) or implicit (then the programmer receives a warning) entering of the asynchronous mode.

From straight regions, only straight functions can be called. If a synchronous function should be called, the programmer must use a `start` or `join`. Straight calls to an asynchronous function again result in an implicit cast of the call synchronicity to an asynchronous region—the programmer may also wrap the call by a `farm` or `seq` to make this transition explicit.

These calling rules are summarized in Table 4.2.

4.2.6 Synchronous Execution and the Group Concept

In synchronous regions, the statements are executed by *groups* of processors, rather than by individual processors. Initially, there is just one group containing all `__STARTED_PROCS__` available processors. Groups can be recursively subdivided, such that there is a treelike [8] group hierarchy at any time, as discussed in Section 4.1.

At each point of program execution in synchronous mode, Fork maintains the strict synchronicity invariant, which says that all processors of a group operate strictly synchronously:

> ***Strict synchronicity***
>
> *All processors belonging to the same active processor group are operating strictly synchronously, that is, they follow the same path of control flow and execute the same instruction at the same time.*

Also, all processors within the same group have access to a common shared address subspace. Thus newly allocated shared variables exist once for each group allocating them.

A processor can inspect the number of processors belonging to its current group, either by calling the standard library routine

```
straight unsigned int groupsize( void );
```

or by reading the predefined constant shared integer variable #.

`groupsize()` may also be called in asynchronous mode, but then it returns only the number of those processors of the current group that are still working inside their asynchronous region; in other words, that have not yet reached the implicit barrier at the end of the asynchronous region they are working on.

Subgroup Creation and Group Activity Regions

At the entry into a synchronous region by `start` or `join`, the processors form one single processor group. However, it may be possible that control flow diverges at conditional branches with private condition expression. To guarantee synchronous execution within each active subgroup, the current leaf group must then be temporarily deactivated and split into one or several subgroups. The subgroups become active and the strict synchronicity invariant is maintained only within each of the subgroups. Where control flow reunifies again, the subgroups terminate, and the parent group is reactivated.

The *subgroup-creating constructs* are summarized in Table 4.3. A subgroup-creating construct is called *group-splitting* if the number of subgroups that it creates may be greater than 1, namely the `fork` statement and the two-sided `if` statement with a private condition.

The *activity region* of a group G is the set of statements that may be executed by the processors of G when G is their currently active group: a leaf of the group hierarchy tree. This includes all statements from the creation of G to the termination of G, excluding the activity regions of subgroups of G and the bodies of `farm` statements (if there are any).

The *live region* of a group G denotes all statements where G may exist, either active or inactive. Hence, it can be defined recursively as the union of the activity region of G, all `farm` bodies in the activity region of G, and the live regions of all subgroups of G.

[8] If `join` statements (see Section 4.2.9) are executed, the group hierarchy forms a *forest* rather than a tree. As this difference is not important for the following discussions, we omit it for simplicity of presentation.

Construct	Region	# sub-groups	Redefines @	Sets $	Ranks $$	See Section
`fork(`*k*`;@=...;)` *stmt*`;`	Synchronous	*k*	√	—	√	4.2.6
`if(`*prcond*`)` *stmt*`;`	Synchronous	1	√ to 0	—	√	4.2.6
`if(`*prcond*`)` *stmt*`; else` *stmt*`;`	Synchronous	2	√ to 0/1	—	√	4.2.6
loop with private exit condition	Synchronous	1	—	—	√	4.2.6
call of a synchronous function	Synchronous	1	—	—	√	4.2.6
`start` *stmt*`;`	Asynchronous or straight	1	√ to 0	√	√	4.2.5
`join(...)` *stmt*`;`	Asynchronous	1	√ to 0	√	√	4.2.9

TABLE 4.3: Summary of the subgroup-creating constructs in Fork.

Group ID `@`, Group-Local Processor ID `$`, and Group Rank `$$`

Each group has a (shared) integer group ID, which is primarily intended to distinguish the subgroups of a group. At `start`, `join`, and at the beginning of program execution, the group ID of the created group is set to 0. Group-splitting constructs set the subgroup IDs consecutively from 0 to the number of subgroups minus one. At the other subgroup-creating constructs, the subgroup inherits the group ID of its parent group. A processor can inspect the group ID of its current group by the predefined constant shared integer variable `@`.

Within a group, processors are ranked automatically from 0 to the group size minus one. This *group rank* can be inspected by the predefined constant private integer variable `$$`. At each (implicit or explicit) groupwide barrier synchronization, `$$` is kept up-to-date. Hence, `$$` may change at a barrier if a processor leaves the group earlier.

In some cases the programmer prefers to rely on a groupwide processor numbering that does not change, even if some processors leave the group. For these cases, there is the *group-relative processor ID*, which is accessible via the predefined private integer variable `$`. At program start, `$` is initialized to the processor ID `__PROC_NR__`. `start` and `join` renumber `$` from 0 to the group size minus one. At all other subgroup-creating constructs, the `$` value of the subgroups is just inherited from the parent group; the programmer is responsible to renumber them explicitly if desired.

The group ID `@` and the group-relative processor ID `$` are automatically saved when deactivating the current group, and restored when reactivating it.

Synchronous `if` Statements with Private Condition

Shared `if` or loop conditions do not affect the strict synchronicity invariant, as the branch taken is the same for all processors executing it.

Now consider an `if` statement with a private condition (or "private if statement" for short), such as

```
if ( $$ % 2 )
   statement1;       // then-part (condition was true)
```

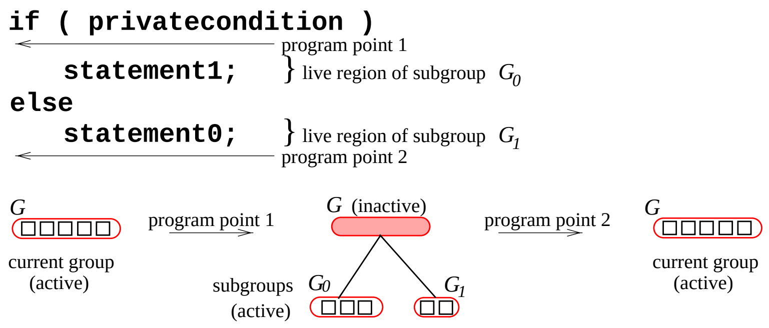


FIGURE 4.6: A synchronous two-sided `if` statement with a private condition causes the current group to be split into two subgroups. Processors are represented by small rectangular boxes. At program point 1, the subgroups are created and entered. At program point 2, the subgroups are left by their processors and terminate.

```
else
   statement0;       // else-part (condition was false)
```

Maintaining the strict synchronicity invariant means that the current processor group G must be split into two subgroups G_1 and G_0 (see Figure 4.6); the processors for which the condition evaluates to a nonzero value form the first subgroup G_0 and execute the `then` part while the remaining processors form subgroup G_1 and execute the `else` part. The parent group G is deactivated; its available shared address subspace is subdivided among the new subgroups.

Each subgroup can now declare and allocate shared objects relating only to that subgroup itself. Also, the subgroups inherit their group-relative processor IDs `$` from the parent group but may redefine them locally. The group IDs `@` are automatically set to 0 for G_0 and 1 for G_1.

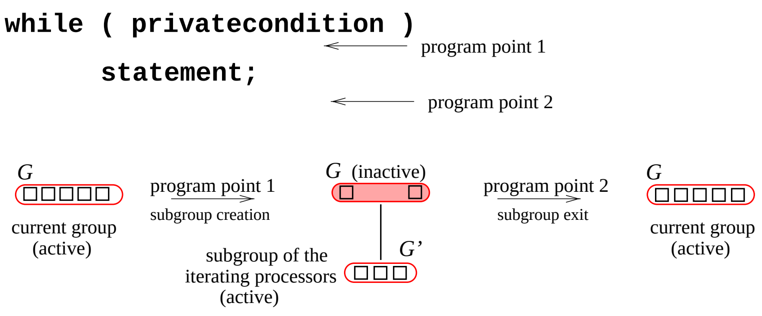
When both subgroups G_0 and G_1 have finished the execution of their respective branch, they are released and the parent group G is reactivated by an explicit groupwide barrier synchronization of all its processors.

Nested synchronous `if` statement with private conditions result in nested subgroup creation, see *Practical PRAM Programming* [B1, Chap. 5] for an example. Nesting may also be applied recursively; a larger example for the application of the two-sided `if` statement for recursive dynamic group splitting can also be found in that book [B1, Chap. 5], namely, a parallel implementation of the Quickhull algorithm [PS85].

Synchronous Loops with Private Exit Condition

As in C, the following loop constructs are available in Fork:

while (e) S
do S while (e);
for (e_1; e_2; e_3) S



```
while ( privatecondition )
      statement;
```

FIGURE 4.7: In synchronous mode, a loop (here: `while`) with a private exit condition causes the current group G to form a subgroup G' for the iterating processors.

where the `for` loop is just an abbreviation for

e_1`; while (`e_2`) {` S`;` e_3`; }`

If the loop exit condition (e or e_2, respectively) is shared, all processors will perform the same number of iterations and thus remain synchronous.

On the other hand, if the loop exit condition is private, the number of iterations executed will generally vary for the processors of the group G executing the loop. Hence, a subgroup G' of G must be created for the iterating processors (see Figure 4.7), as barriers within the loop body should address only those processors that are still iterating in the loop.

Once the loop iteration condition has been evaluated to zero for a processor in the iterating group G', it leaves G' and waits at the end of the loop to resynchronize with all processors of the parent group G. At loops it is not necessary to split the shared memory subspace of G, since processors that leave the loop body are just waiting for the last processors of G to complete loop execution.

Note that the evaluation of the loop exit condition and, in the case of the `for` statement, the increment e_3, are done by the iterating subgroup, not by the parent group.

As an illustration, consider the following synchronous loop

```
pr int i;
for (i=$$; i<n; i+=#)  a[i] += c * b[i];
```

If the loop is executed by p processors, the private variable `i` of processor k, $0 \leq k \leq p-1$, iterates over the values k, $k+p$, $k+2p$, and so on. The whole index range from 0 to `n` -1 is covered because the involved processors are consecutively ranked from 0 to $p-1$. Note that the access to the group size # in the increment expression does not harm here, as it is here equal to the parent group size p for all but the last iteration. Nevertheless, if some processors leave the group by `break`, this may no longer be the case. In that case, p should be assigned to a temporary shared variable before executing the loop.

In contrast to asynchronous loops, the use of `continue` is not permitted in synchronous loops.

Calls to Synchronous Functions

A synchronous call to another synchronous function may cause a divergence of control flow if processors in a subgroup created during execution of the called function leave it by a `return` statement while other subgroups continue to work on that function. Hence, a single subgroup for the calling processors must be created, and there is an implicit barrier synchronization point immediately after the program control returns to the calling function. If the called function is statically known and such a situation cannot occur, the subgroup construction and synchronization can be omitted.

The `fork` Statement

Splitting into subgroups can, in contrast to the implicit subdivision at private branch conditions, also be done explicitly, by the `fork` statement[9] that gave the entire language its name. The semantics of

`fork (`e_1`; @=`e_2`; $=`e_3`)  <stmt>`

is as follows. First, the shared expression e_1 is evaluated to the number (say, g) of subgroups to be created. Then the current leaf group is split into that many subgroups, whose group ID's `@` are indexed $0, ..., g-1$ (see Figure 4.8). Evaluating the private expression e_2, every processor determines the index of the newly created subgroup it wants to become a member of. Processors that evaluate e_2 to a value outside the range $0, ..., g-1$ skip `<stmt>` and wait at the implicit barrier synchronization point at the end of `<stmt>`. The assignment syntax for the second parameter of `fork` enlights the fact that the group IDs are redefined for the subgroups. Finally, by evaluating the private expression e_3, a processor can renumber its current group-relative processor ID within the new leaf group. The assignment to `$`, though, can also be omitted if renumbering `$` is not required. Note that empty subgroups (with no processors) are possible; an empty subgroup's work is immediately finished, though.

Continuing, the parent group is now deactivated. Its shared memory subspace is partitioned into g slices, each of which is assigned to one subgroup. Thus, each new subgroup has now its own shared memory subspace.

Then, each subgroup continues by executing `<stmt>`; the processors within each subgroup work synchronously, but different subgroups can choose different control flow paths.

After a subgroup has completed the execution of the body `<stmt>`, it is abandoned. All processors of the parent group are synchronized by an exact barrier; the parent group is then reactivated as the current leaf group. The shared memory subspaces of the former subgroups are re-merged and can now be overwritten by the parent group.

Finally, the statement following the `fork` statement is executed again synchronously by all processors of the parent group.

[9]Also, the keyword `fork` was chosen for historical reasons [HSS92]. From today's perspective, something like `split` would be better suited because this avoids confusion with the UNIX `fork` mechanism that denotes spawning a child process from a single process, while in Fork the number of processes remains constant and only the organization of them into groups is rearranged. Again, if the programmer prefers a different keyword, the C preprocessor can be used.

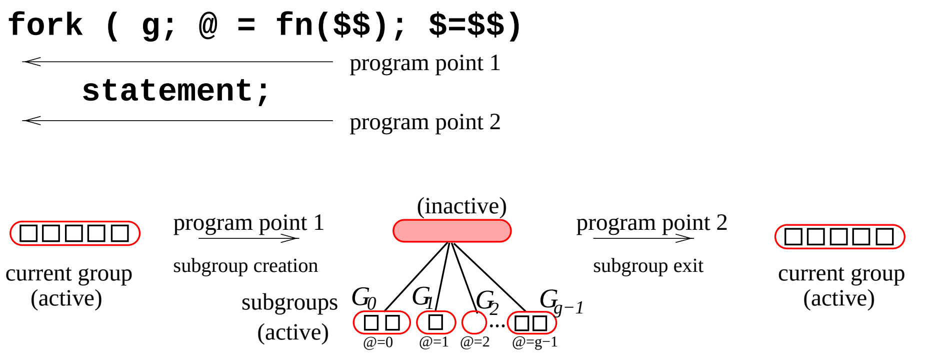


FIGURE 4.8: The `fork` statement splits group G into g subgroups G_0, G_1, ..., G_{g-1}.

Another example for a parallel divide-and-conquer algorithm, here implemented using the recursive application of the `fork` statement, is the parallel drawing of fractals[10] known as *Koch curves* [Man82]. Koch curves consist of a polygonal chain of straight-line segments. A Koch curve of degree d is obtained recursively by replacing a straight-line segment in a curve of degree $d-1$ by an appropriately scaled copy of a special curve, the so-called generator pattern. The procedure is initialized by an initiator figure as a Koch curve of degree zero (see Figure 4.9).

Drawing a Koch curve of degree `DEGREE` in parallel is straightforward using the recursive `Fork` function `Koch` given in Figure 4.10, which refines a straight line segment connecting two points `(startx,starty)` and `(stopx,stopy)`. The shared parameter `level` indicates the current level of recursive refinement.

The routine `line(`x_1, y_1, x_2, y_2, c`)` plots in parallel a line connecting two points (x_1, y_1) and (x_2, y_2) with color c. This and the other required graphics routines are declared in `graphic.h` and summarized in [B1, Appendix D]. The constant `factor` may be used to distort the curve; its default value is `0.33`.

[10]The reader is referred to Section 10.18 of the textbook by Hearn and Baker [HB94] for an introduction to fractals in computer graphics.

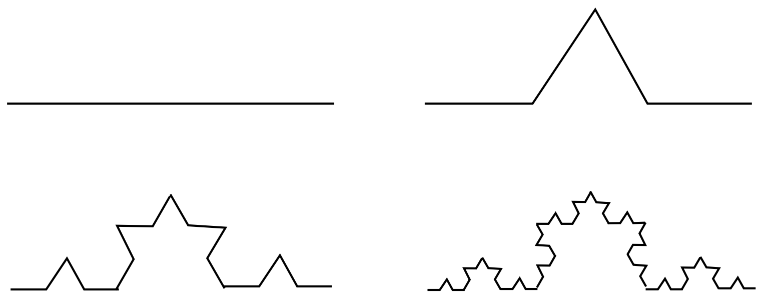


FIGURE 4.9: Koch curves of degrees 0, 1, 2, and 3 for a straight-line segment initiator (left upper diagram). The generator pattern is here identical to the degree 1 curve (right upper diagram).

```
sync void Koch ( sh int startx, sh int starty,
                 sh int stopx,  sh int stopy,  sh int level )
{
 if (level >= DEGREE) {   /*reach the limit of recursion*/
    line( startx, starty, stopx, stopy, level+2, 1 );
    return;
 } /* else: another recursion step */
 if (#<4)
    seq  seq_Koch( startx, starty, stopx, stopy, level );
 else {
    sh int x[5], y[5], dx, dy;
    seq {         /* linear interpolation: */
       dx = stopx - startx;      dy = stopy - starty;
       x[0] = startx;            y[0] = starty;
       x[1] = startx + (dx/3);   y[1] = starty + (dy/3);
       x[2] = startx + dx/2 - (int)(factor * (float)dy);
       y[2] = starty + dy/2 + (int)(factor * (float)dx);
       x[3] = startx + (2*dx/3); y[3] = starty + (2*dy/3);
       x[4] = stopx;             y[4] = stopy;
    }
    fork ( 4; @=$$%4; )
       Koch( x[@], y[@], x[@+1], y[@+1], level + 1 );
 }
}
```

FIGURE 4.10: The recursive core routine for drawing Koch curves.

Final Remarks

A dual point of view to the hierarchical group concept and the synchronous execution mode in Fork is the concept of *parallel processes*. In contrast to common sequential processes known from sequential programming systems, a parallel process is executed concurrently by the processors of a group. Determinism is guaranteed by exact synchronicity. Group splitting is analogous to spawning more processes—where now the child processes are parallel processes with a correspondingly lower degree of available parallelism since the set of processors is also subdivided across the subgroups.

4.2.7 Pointers and Heaps

Pointers

The usage of pointers in Fork is as flexible as in C, since all private address subspaces have been embedded into the global shared memory. In particular, the programmer does not have to distinguish between pointers to shared and pointers to private objects, in contrast to some other parallel programming languages like C^* [RS87], where the sharity of the pointee of a pointer also must be declared and statically checked at assignments. Shared pointer variables may point to private objects and vice versa; the programmer is responsible for such assignments making sense. Thus

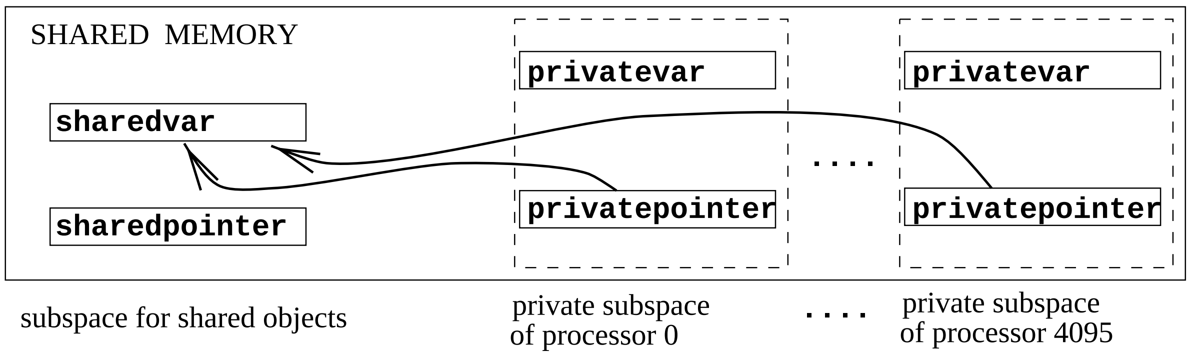
```
sh int *sharedpointer;
```

declares a shared pointer that may point either to a private or to a shared location.

For instance, the following program fragment

```
pr int privatevar, *privatepointer;
sh int sharedvar;
privatepointer = &sharedvar;
```

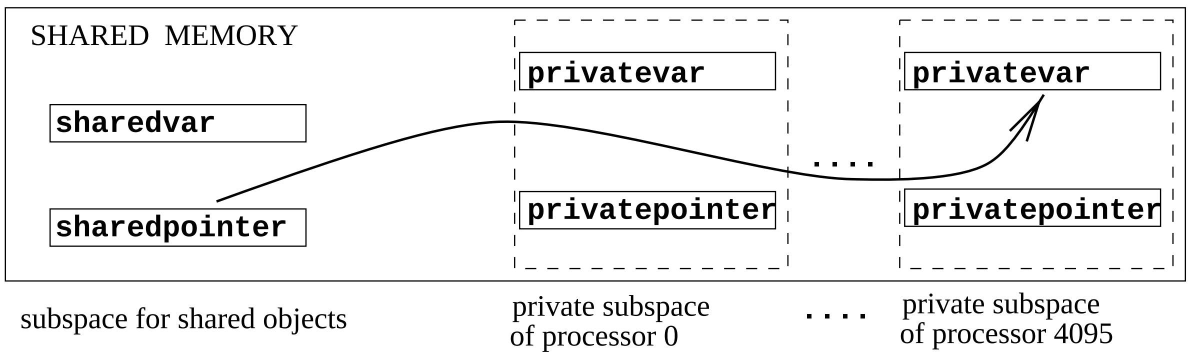
causes the private pointer variable `privatepointer` of each processor participating in this assignment to point to the shared variable `sharedvar`.


SHARED MEMORY
sharedvar
sharedpointer
privatevar
privatepointer
....
privatevar
privatepointer
subspace for shared objects
private subspace of processor 0
....
private subspace of processor 4095

Accordingly, if all processors execute simultaneously the assignment

```
sharedpointer = &privatevar; // concurrent write access,
                             // result is deterministic
```

the shared pointer variable `sharedpointer` is made point to the private variable `privatevar` of the processor that finally got its value stored; on the SB-PRAM this is the processor with the highest write priority participating in this assignment.


SHARED MEMORY
sharedvar
sharedpointer
privatevar
privatepointer
....
privatevar
privatepointer
subspace for shared objects
private subspace of processor 0
....
private subspace of processor 4095

Hence, a shared pointer variable pointing to a private location nevertheless represents a shared *value*; all processors belonging to the group that declared that shared pointer see the *same* private object `privatevar` through that pointer. In this way, private objects can be made globally accessible. Note that the compiler automatically inserts code that converts the potentially BASE-relative address `&privatevar` to an absolute address, such that it defines a unique memory location.

Heaps

Fork supplies three kinds of heaps: one permanent, private heap for each processor; one automatic shared heap for each group; and a global, permanent shared heap.

Space on the *private heap* of a processor can be allocated and freed by the asynchronous functions `malloc()` and `free()` known from C.

Space for k memory words on the *permanent shared heap* is allocated by calling the `shmalloc()` function:

```
void *ptr = shmalloc( k );
```

and a pointer `ptr` to the allocated block is returned. If there is not enough permanent shared heap memory left over, an error message is printed, and a NULL pointer is returned. Note that each processor calling `shmalloc(k)` allocates a separate block of length `k`.

Shared memory blocks allocated by `shmalloc()` can be freed and reused. Such a block pointed to by `ptr` is freed by calling the asynchronous function `shfree()`:

```
shfree( ptr );
```

If reuse of a permanent shared memory block is *not* intended, a faster way of allocating it uses the `alloc()` routine:

```
void *ptr = alloc( k );
```

Be aware that `shfree()` must not be applied to blocks allocated by `alloc()`; the program may crash in this case.

The *group heap*, which is accessible only in synchronous regions, is intended to quickly provide temporary shared storage blocks that are local to a group. Consequently, the life range of objects allocated on the group heap by the synchronous `shalloc()` function

```
sh void *sptr = shalloc( k );
```

is limited to the life range of the group by which that `shalloc()` was executed. Thus, such objects are automatically removed if the group allocating them is released, in the same way as the local variables of a function allocated on the runtime stack cease to exist when the function terminates. In this way, the group heap is really something in between a parallel stack and a parallel heap.

For better space economy in the group heap, there is the synchronous function `shall-free()` that is applicable in synchronous functions. A call

```
shallfree();
```

frees *all* objects `shalloc`ed so far in the current (synchronous) function.

Note also the following important difference between `shmalloc` and `shalloc`. If p processors call `shalloc()` concurrently, each processor obtains a copy of the address of *one* allocated block. In contrast, if p processors call `shmalloc()` concurrently, each processor obtains an individual pointer to an individually allocated block, that is, p blocks are allocated altogether.

Pointers to Functions

Pointers to functions are also supported. The synchronicity and the sharities of the parameters must be declared when declaring the pointer, and at assignments, these are statically checked. Type casting is possible as far as allowed in C, but the declared synchronicities and parameter sharities must be identical.

Special attention must be paid when using private pointers to functions in synchronous mode:

```
sync int fp_example ( ..., pr void (*fp)(void), ... )
{ ...
  fp();   // ! this may call several different functions!
  ...     // ! call to *fp is casted to asynchronous mode
}
```

Since each processor may then call a different function (and, in general, it is statically not known which one), a synchronous call to a function using a private pointer would correspond to a huge switch over all functions possibly being called, and maintaining the strict synchronicity invariant would require to create a separate subgroup for each possible call target—a tremendous waste of shared memory space! For this reason, synchronous calls to synchronous functions via private pointers are forbidden. If the callee is asynchronous, the compiler automatically switches to asynchronous mode for the call and emits a warning. Private function pointers may thus only point to `async` functions.

In contrast, shared function pointers may also point to synchronous functions, as control flow is not affected:

```
sync int fp_example ( ..., sh sync void (*fp)(void), ... )
{ ...
  fp(); // all processors of the group call the same function
  ...
}
```

4.2.8 Exploring the Asynchronous Mode

The `barrier` Statement

The statement

```
barrier;
```

forces a processor to wait until all processors of its current group have also reached a `barrier` statement. Generally these need not be identical in the program text, as processors may follow individual control flow paths in asynchronous mode, but in any case each processor of the group should execute the same number of `barrier`s before exiting the asynchronous region (or the entire program), to avoid deadlocks and unforeseen program behavior.

Critical Sections, Locks, and Semaphores

A ***critical section*** is a piece of code that may be executed by at most one processor at any point of time. If a processor wants to execute the critical section while another processor is working inside, the processor must wait until the critical section is free.

Critical sections generally occur when shared resources such as files, output devices like screen or printer, or shared memory objects like shared variables and shared heap objects are accessed concurrently. If these concurrent accesses are not protected by some arbitrating mechanism, nondeterminism (also called *race conditions*) may occur, which means that the result of a program may depend on the relative speed of the processors executing it.

In a general sense, a ***semaphore*** is a shared program object accessed according to a safe protocol in a deterministic way. More restricted definitions assume that semaphores be simple unsigned integer (i.e., memory-word-sized) variables that are only accessed by atomic operations. Semaphores are useful for many purposes. In this section we consider three cases where semaphores are used: (1) making the execution of critical sections deterministic by restricting access to a critical section to a fixed set of processors only, (2) for user–coded barrier synchronization, and (3) for dynamic loop scheduling. More examples of using semaphores will follow in the subsequent chapters of this book.

A ***lock*** is a binary semaphore variable used to guarantee deterministic execution of a critical section: If all processors follow a well-defined locking/unlocking protocol based on this variable, only one processor is allowed to be inside a critical section at any point of time. Thus, access to the critical section is sequentialized. This is also called *mutual exclusion*, and for this reason locks are also known as *mutex* variables.

Simple locks In the simplest case, a lock can be realized as a shared (integer) variable with two possible states: 0 (`UNLOCKED`) and 1 (`LOCKED`). Initially it is `UNLOCKED`. A critical section S is protected by forcing all processors to wait at the entry of S until the lock is `UNLOCKED`. As soon as a processor manages to enter S, it sets the lock to `LOCKED`, with the goal of prohibiting entry for all other processors attempting to enter S. When the processor leaves S, it resets the lock to `UNLOCKED`.

However, we must avoid the situation that multiple processors evaluate the condition `(lock != UNLOCKED)` simultaneously, such that all of them would read a zero value and enter the critical section. To avoid this problem with simultaneous or nearly simultaneous read and write accesses to shared memory locations, the combination of reading the (old) value (`UNLOCKED`) and writing the new one (`LOCKED`) must be ***atomic*** for each processor, that is, at simultaneous accesses by several processors, only one of them reads the old value while the other ones read the new one (or, if different values are written, each processor also reads a different value). This is a classical problem of parallel processing. Atomic operations [GLR83] such as *test&set*, *fetch&add*, or *compare&swap*, are supported by hardware on most

```
#include <fork.h>
#include <io.h>

#define UNLOCKED 0
#define LOCKED 1

sh int lock = UNLOCKED;

main()
{
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n",
          __STARTED_PROCS__);
 barrier;
 while (mpmax(&lock,LOCKED) != UNLOCKED) ;
 printf("Hello world from P%d\n", __PROC_NR__); //crit. sect.
 lock = UNLOCKED;
 barrier;
}
```

FIGURE 4.11: Mutual exclusion for the critical section is protected by a lock variable `lock`. All accesses to `lock` are atomic. If executed concurrently, `mpmax()` returns 0 (`UNLOCKED`) only for one processor, if `lock` was `UNLOCKED` before, and `LOCKED` (1) in all other cases. When using `mpmax`, it is sufficient if `LOCKED` $>$ `UNLOCKED`.

systems.[11] In Fork, the `mpadd()` operator introduced in Section 4.2.4 can be used immediately for atomic *fetch&add*, and the `mpmax` operator for atomic *test&set*. In the program of Figure 4.11 we use `mpmax()` for locking.

And we get the desired output, where the `Hello world` messages are not intermixed [B1, Chap. 5] but printed consecutively in separate lines:

```
PRAM P0 = (p0, v0)> g
Program executed by 4 processors

Hello world from P0
Hello world from P1
Hello world from P2
Hello world from P3
EXIT: vp=#0, pc=$00000214
EXIT: vp=#1, pc=$00000214
EXIT: vp=#2, pc=$00000214
EXIT: vp=#3, pc=$00000214
Stop nach 24964 Runden, 713.257 kIps
0214 18137FFF  POPNG   R6, ffffffff, R1
PRAM P0 = (p0, v0)>
```

[11] Software realizations are possible but generally less efficient in practice (see Zhang et al. [ZYC96] for a survey of software locking mechanisms).

Comparing this with the version without locking, we see that the number of machine cycles required by the program has now considerably increased as the execution of the critical section is sequentialized.

The mechanism described here is made available in Fork as a data type called ***simple lock***

```
typedef int simple_lock, *SimpleLock;
```

which is predefined in `fork.h`. The following functions resp. macros for simple locks are defined in the Fork standard library:

```
SimpleLock new_SimpleLock( void );
```

creates a new `SimpleLock` instance on the permanent shared heap and initializes it to a zero value. This should be done before it is used for the first time.

```
simple_lock_init( SimpleLock sl );
```

initializes an already allocated `SimpleLock` instance `sl` to a zero value. This should be done before it is used for the first time. A simple lock `sl` is acquired by

```
simple_lockup( SimpleLock sl );
```

which is equivalent to

```
while (mpmax(sl,LOCKED) != UNLOCKED) ;
```

and it is released by

```
simple_unlock( SimpleLock sl );
```

which resets the integer variable pointed to by `sl` to `UNLOCKED`.

Figure 4.12 illustrates the usage of these functions at a rewritten version of the `Hello world` program.

When using mutual exclusion locks, the programmer must be particularly careful. For instance, if the `unlock` operation is skipped by some processor (e.g., because its program control jumps across it, or the programmer has just forgotten to insert the `unlock`, or `unlock`s the wrong lock variable), the subsequent processors would wait forever in their `lockup` call. Such a situation is called a ***deadlock***.

Another common source of deadlock is unsuitable nesting of critical sections. In principle, critical sections may be nested, for example, if a processor needs exclusive access to two or more shared resources. However, problems arise if such a nested critical section is not contiguous in the program, or, conversely spoken, if several pieces of the program are protected by the same lock variable.

Chapter 5 of *Practical PRAM Programming* explains how deadlocks can occur and how they can be avoided.

```
#include <fork.h>
#include <io.h>

#define UNLOCKED 0
#define LOCKED 1

sh simple_lock lock;

main()
{
 simple_lock_init( &lock );
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n",
   __STARTED_PROCS__);
 barrier;
 simple_lockup( &lock );
   printf("Hello world from P%d\n", __PROC_NR__);
 simple_unlock( &lock );
 barrier;
}
```

FIGURE 4.12: A version of the `Hello world` program that uses the simple lock data structure of Fork.

Fair locks A simple lock sequentializes the accesses to a critical section. However it is not guaranteed that processors will get access in the original order of their arrival at the `lockup` operation. A processor may even be forced to wait quite a long time while other processors coming later are served earlier. Thus, the `SimpleLock` is not *fair*. This may become critical if there exists an upper limit for the completion time of the processor's computation, as in real-time applications. Exceeding this limit is called *starvation*.

A ***fair lock*** sequentializes accesses while granting access to the critical section in the order of arrival of the processors at the lock acquire operation. For the Fork programmer there is the data type

```
typedef struct {...} fair_lock, *FairLock;
```

predefined in `<fork.h>`. A new `FairLock` instance is created and initialized by the constructor function

```
FairLock new_FairLock( void );
```

while an already allocated fair lock can be reinitialized by

```
void fair_lock_init( FairLock );
```

Again, these should be called before the first use of the `FairLock` object. The lock acquire operation

```
void fair_lockup( FairLock );
```

causes the calling processor to wait until it is its turn to enter the critical section. By the unlock operation

```
void fair_unlock( FairLock );
```

the critical section is left.

The space required to store a `FairLock` object is two memory words, twice the space of a simple lock. The run time overhead incurred by the `fair_lockup` and the `fair_unlock` operation is practically the same as that for `simple_lockup` resp. `simple_unlock`.

Reader–writer locks In some cases the processors can be classified into two types, according to their intended actions in the critical section: the *readers* that "only" want to read a shared resource but leave it unchanged, and the *writers* that want to write to it.

The *reader–writer lock* is useful when multiple readers should be allowed to work in the critical section at the same time, and mutual exclusion is required only to sequentialize different write accesses with respect to each other, and write and read accesses. Different read accesses need not be sequentialized with respect to each other.

The current status of a reader–writer lock can thus be described as a pair $(w, r) \in \{0, 1\} \times \{0, ..., p\}$, where w is a counter indicating the number of writers currently working inside the critical section and r is a counter indicating the number of readers currently working inside the critical section. Readers may enter only if $w = 0$, while writers may enter only if $w = 0 \wedge r = 0$. Note that access to w and r must be atomic.

The reader–writer lock data type

```
typedef struct {...} rw_lock, *RWLock;
```

is implemented in the Fork standard library and declared in `fork.h`.

A new `RWLock` instance can be created by the constructor function

```
RWLock new_RWLock( void );
```

while an existing `RWLock` instance `l` can be reinitialized by

```
void rw_lock_init( RWLock l );
```

The lock acquire operation

```
void rw_lockup( RWLock l, int mode );
```

with `mode` $\in$ { `RW_READ`, `RW_WRITE` }, causes the executing processor to wait until the critical section is free for it, depending on whether its intended action is a read access (`mode=RW_READ`) or write access (`mode=RW_WRITE`).

```
void rw_unlock( RWLock l, int mode, int wait );
```

releases a reader–writer lock `l` held by the executing processor in mode `mode` $\in$ { `RW_READ`, `RW_WRITE` }.

The implementation of `RWLock` for `Fork` (see Section 5.1.11) is a *priority reader–writer lock* that prioritizes the writers over the readers; fairness is guaranteed only among the writers. If there are many write accesses, the readers may suffer from starvation. In order to avoid this by "ranking down" the writers' advantage, the corresponding `rw_unlock` function offers a tuning parameter `wait` that specifies a delay time for the writers at exit of the critical section before allowing a new writer to enter. Hence, increasing `wait` improves the chance for readers to enter the critical section.

A `RWLock` instance requires 3 words of memory. The locking and unlocking routines incur significantly more time overhead than these for simple and fair locks, depending on the `mode` parameter.

Reader–writer–deletor locks In some situations it may happen that a lock is part of a non-persistent data structure, for instance, in data structures with delete operations (see, e.g., the parallel skip list described in *Practical PRAM Programming* [B1,Chap. 7.4]. In that case, the lock itself will be subject to `shfree` operations and potentially reused by subsequent `shmalloc` operations. Hence, all processors spinning on that lock must be immediately notified that their `lockup` operation *fails* as a result of lock deletion, thus they can resort to some recover action, such as restarting the last transaction from the beginning.

We define an extension of the reader–writer lock for this case, the reader–writer–deletor lock (`RWDLock`). Processors are classified into readers, writers, and deletors, where the deletors are a special case of writers who do not (only) perform a modification of the protected data structure but also delete a part of it that includes the `RWDLock` instance.

The data type

```
typedef struct {...} rwd_lock, *RWDLock;
```

is declared in `fork.h`. A new `RWDLock` instance is created by calling the constructor function

```
RWDLock new_RWDLock( void );
```

The lock acquire operation

```
int rwd_lockup( RWDLock l, int mode );
```

with `mode` $\in$ { `RW_READ`, `RW_WRITE`, `RW_DELETE` }, causes the executing processor to wait until the critical section is free for it, depending on whether its intended action is a read access (`mode` is `RW_READ`), write access (`mode` is `RW_WRITE`), or delete access (`mode` is `RW_DELETE`). If the `lockup` operation succeeds, a nonzero value is returned. In the case of failure, the function returns 0.

```
void rwd_unlock( RWDLock l, int mode, int wait );
```

releases the reader–writer–deletor lock `l` that the executing processor held in mode `mode` $\in$ { `RW_READ`, `RW_WRITE`, `RW_DELETE` }. The parameter `wait` is used as in Section 4.2.8.

An `RWDLock` instance uses 4 words of memory. The overhead of locking and unlocking is comparable to that of `RWLock`s.

User-defined barrier synchronization A barrier synchronization point causes the processors executing it to wait until the last processor has reached it. The `barrier` statement of Fork provides an exact barrier synchronization, which additionally guarantees that all processors continue with program execution at exactly the same clock cycle. An inexact barrier (which thus makes sense only in asynchronous regions) does not require this additional condition.

Such an inexact barrier can be realized by an integer semaphore. The initializiation is again to be executed before any processor uses the semaphore:

```
sh unsigned int my_barrier = 0;
```

A processor reaching the barrier synchronization point atomically increments the semaphore and waits until the semaphore's value reaches the total number of processors that should execute this barrier, such as `__STARTED_PROCS__`:

```
syncadd( &my_barrier, 1 );     // atomic increment
while (my_barrier != __STARTED_PROCS__)  ;   // wait
```

From this simple implementation it also becomes obvious that a processor that jumped across this piece of code would cause a deadlock, as the other processors would wait for it forever in the `while` loop.

If the semaphore `my_barrier` should be reused for subsequent barrier synchronizations, then, after continuing, `my_barrier` must be reset to zero. However, this requires waiting for a implementation-dependent (short) time after exiting the `while` loop, because otherwise a processor leaving the while loop[12] early may reset the semaphore before a "late" processor has checked the exit condition, which would then read the new semaphore value zero—and wait forever. Also the other processors would be waiting forever at the next synchronization point; this is a deadlock.

In order to avoid this insecure solution, we implement a *reinitializing barrier* [Röh96] where a processor determines whether it is the last one to arrive at the barrier. If so, then it is responsible for resetting the semaphore:

```
if ( mpadd( &my_barrier, 1) != __STARTED_PROCS__ - 1 )
   while (my_barrier != __STARTED_PROCS__)  ;   // wait
else {
   // executed by the processor arriving last:
   while (my_barrier != __STARTED_PROCS__)  ;   // wait
   my_barrier = 0;
}
```

The `while` loop in the `then` branch guarantees that no processor re-uses `my_barrier` before its reinitialization has been completed. The `while` loop in the `else` branch seems to be redundant, as `my_barrier` has then already reached the value `__STARTED_PROCS__`,

[12]Even an empty while loop takes a few machine cycles per iteration for checking the exit condition. As long as we do not program explicitly in assembler, we cannot rely on specific execution times.

but it is necessary to guarantee that the processor arriving last waits for at least one iteration, in order to give the other processors enough time to check the loop exit condition and exit their `while` loop before the semaphore is reset to zero.[13]

Example: dynamic loop scheduling

In asynchronous mode, integer semaphores can be used together with the `mpadd()` operator to program self-scheduling parallel loops. Loop scheduling is discussed in detail in [B1, Chap. 7.1]; for now it is sufficient to know that a so-called self-scheduling loop is a loop with independent iterations that are dynamically assigned to processors. Dynamic scheduling generally achieves better load balancing than static scheduling if the loop iterations may differ considerably in their execution times.

A shared loop iteration counter

```
sh int iteration = 0;
```

initialized to zero serves as a semaphore to indicate the next loop iteration to be processed. A processor that becomes idle applies a `mpadd()` to this iteration counter to fetch its next iteration index:

```
int i;                  /* dynamic scheduling: */
...
for ( i=mpadd(&iteration,1); i<N; i=mpadd(&iteration,1) )
    execute_iteration ( i );
```

Hence, if for a processor its execution of `mpadd` returns an iteration index greater than `N`, it can leave the loop. The mechanism guarantees that each iteration in $\{0, ..., N-1\}$ is executed exactly once. The only condition is that $N + p < 2^{32}$ where p denotes the number of processors executing this loop.

Macros for parallel loop constructs in Fork

For common cases of one–, two– and threedimensional loops, there are several macros predefined in `fork.h`. Most of them are available in a statically scheduled as well as in a dynamically scheduled variant.

Static scheduling composes sets of iterations to p parallel tasks for p processors, either blockwise, where a task consists of contiguous iterations, or cyclic, where task j consists of iterations j, $j+p$, $j+2p$, and so on, as applied the `forall` macro:

```
#define forall(i,lb,ub,p) for(i=$$+(lb);i<(ub);i+=(p))
```

Dynamic scheduling uses an integer semaphore as in Section 4.2.8.

The parallel loop constructs are described and compared in *Practical PRAM Programming* [B1, Chap. 7.1]; we give here a short survey in Table 4.4.

[13]For the execution of Fork programs the operating system guarantees that processors are not interrupted asynchronously, except for reasons that lead to immediate abortion of the execution of the program. Röhrig [Röh96] describes a barrier mechanism handling the case where processors may be interrupted.

Parallel loop construct	Dimen-sions	Step size	Mapping to processors	Distribution of iterations
`forall`	1	$+1$	static	cyclic
`Forall`	1	> 1	static	cyclic
`forallB`	1	$+1$	static	blockwise
`FORALL`	1	≥ 1	dynamic	(cyclic)
`fork+forall`	2	$+1$	static	cyclic/cyclic
`forall2`	2	$+1$	static	cyclic
`Forall2`	2	> 1	static	cyclic
`FORALL2`	2	≥ 1	dynamic	(cyclic)
`forall3`	3	$+1$	static	cyclic

TABLE 4.4: Summary of macros for parallel loop constructs introduced in *Practical PRAM Programming* [B1, Chap. 7.1].

4.2.9 The `join` Statement

The `join` statement is a generalization of the `start` statement. While `start` expects all `__STARTED_PROCS__` processors to synchronize and execute its body in synchronous mode, `join` allows switching from asynchronous to synchronous mode with much more flexibility and permits (nearly) arbitrary nesting of synchronous and asynchronous regions.

A useful analogy to understand the semantics of the `join` statement is a *bus stop*. Imagine a city with several excursion *bus lines*, each of which has one unique bus stop. One excursion bus circulates on each bus line. At the door of each bus there is a *ticket* automaton that sells tickets when the bus is waiting. Tickets are numbered consecutively from 0 upward. All passengers inside a bus form a group and behave synchronously. They can be distinguished by their *rank* of entering, namely, their ticket number. Each bus has a *bus driver*, namely, the passenger that entered first and thus obtained rank zero.

Let us now consider what happens at the bus stop. Passengers come by asynchronously and look for the bus to join the excursion. If the bus is *gone*, they have the choice to either retry and *continue* waiting for the next bus tour of this line (if there is one), perhaps by individually doing some *other useful work* meanwhile, or to resign, *break* waiting and continue with the next statement.

If the bus is not gone, it is waiting and its door is not locked. Hence the passenger can get a ticket at the ticket automaton and enter. The first passenger entering the bus receives ticket number 0; this person becomes the bus driver and does some *initialization* work for the bus excursion. The passenger waits as long as a certain *start condition* does not hold; in the meantime, other passengers may enter the bus. Then the driver signals that the bus is about to start and switches off the ticket automaton—thus no one can enter this bus any more. At this point, the passengers inside still have the possibility to decide whether to *stay inside* the bus for the excursion, or whether they should immediately spring off the starting bus, for example, if they feel that the bus is too crowded for them. After that, the door is definitely locked. The passengers inside form a *group* and behave synchronously during the *bus tour*. They can allocate shared objects in the bus that are then accessible to all bus passengers during the tour.

When the tour is finished and the bus returns, all passengers leave the bus at the bus stop and proceed, again asynchronously, with their next work. The bus itself is cleaned and ready to receive new passengers by reopening the ticket automaton.

The behavior described in this bus analogy is supplied in Fork by the language construct

```
join ( SMsize; startCond; stayInsideCond )
       busTourStatement;
  else missedStatement;
```

where

- `SMsize` is an expression evaluating to an integer that determines the size of the shared stack and group heap (in memory words) for the group executing the bus tour. It should be at least 100 for simple "bus tours."
- `startCond` is an integer expression evaluated by the "driver" (i.e., the first processor arriving at the `join` when the bus is waiting). The driver waits until this condition becomes nonzero.
- `stayInsideCond` is an integer expression evaluated by all processors in the "excursion bus." Those processors for which it evaluates to zero branch to the `else` part. The others barrier-synchronize, form a group, and execute `busTour` in synchronous mode.
- `startCond` and `stayInsideCond` can access the number of processors that meanwhile arrived at this `join` statement by the macro `__NUM_PR__` and their own rank in this set of processors (their "ticket code") by the macro `__RANK__`.
- `busTourStatement` is a synchronous statement.
- `missedStatement` is an asynchronous statement. It is executed by the processors missing the "excursion bus" (i.e., arriving at the `join` when a group is currently operating in the `join` body) or springing off after testing the `stayInsideCond` condition. The `else` part is optional.
- Inside the `else` branch, processors may return to the entry point of this `join` statement by `continue`, and leave the `join` statement by `break`. This syntax corresponding to loops in C is motivated by the fact that the `else` clause indeed codes a loop that is executed until the processor can participate in the following bus excursion or leaves the loop by `break`.
- The `else` clause is optional; a missing `else` clause is equivalent to `else break;`

The passengers mentioned in the bus tour analogy are the processors. Each `join` instruction installs a unique bus line with a bus stop at the place where the `join` appears in the program. The start condition `startCond` is evaluated only by the bus driver processor. It spins until a nonzero value is obtained. This condition may be used to model a time interval, such as by incrementing a counter, or to wait for an event that must occur, such as for a minimum bus group size to be reached before the bus tour can start. The `stayInsideCond` condition is an integer expression that may evaluate to different values for different processors. The start condition as well as the stay-inside condition may read the current value of the ticket counter, __NUM_PR__, and the ticket number of the evaluating processor, __RANK__. The bus driver allocates a chunk of SMsize words of shared memory on the permanent shared heap (see Section 4.2.7), where it installs a new shared stack and heap for the bus. This chunk of memory is released when the bus tour terminates. The bus tour itself corresponds to the body busTourStatement of the join instruction and is a synchronous region. When the bus tour starts, the group local processor IDs $ of the processors remaining in the bus group are automatically renumbered consecutively, starting at 0.

The optional else part may specify an asynchronous statement missedStatement that is executed by those processors that missed the bus and by those that spring off. A continue statement occurring inside missedStatement causes these processors to go back to the bus stop and try again to get the bus, very similar to continue in loops. Correspondingly, processors encountering a break statement inside missedStatement immediately leave the join statement and proceed with the next work.

If no else part is defined, then all processors missing the bus or springing off the bus leave the join statement and proceed with the next work. Hence, a missing else part is equivalent to else break;.

As an example, consider the following code:

```
join( 100; __NUM_PR__>=1; __RANK__<2)
   // at least one, at most 2 passengers
   farm pprintf("JOIN-BODY! $=%d\n",$);
else {
   pprintf("ELSE-PART!\n");
   continue;  /* go back to join head */
              /* break would leave it */
}
```

Note that a bus line can have only one entry point (i.e., the program point where the join keyword appears) within the program. To ensure this, the programmer can encapsulate the join in a function and call that function from several sites.

Bus tours of different bus lines can be nested. Recursion (directly or indirectly) to the *same* bus line will generally introduce a deadlock because a processor would wait forever for a bus whose return it is blocking itself just by its waiting.

The passengers inside a bus will generally have their origin in different former leaf groups. The old group structure, as well as all shared local variables and objects allocated by that old group structure, are *not* visible during the bus tour. Global shared objects are always visible.

As soon as join statements are executed in a Fork program, the group hierarchy forms a *forest* rather than a tree, since each bus tour caused by a join statement installs a new root group.

Example: Simulating the start statement by join

Using join, the programmer may specify an exact number of passengers expected, or only a maximum or minimum number for them—or some other criterion esteemed relevant; any desirable constellation can easily be programmed using `startCond` and `stayInsideCond` appropriately.

As an example, consider the semantics of the `start` statement. `start` is just a special case of `join`; it collects *all* `__STARTED_PROCS__` available processors, synchronizes and renumbers them, and executes a `statement` in synchronous mode. Thus, we may rewrite

```
start
   statement;
```

as

```
join( 100000; (__NUM_PR__ == __STARTED_PROCS__); 1 )
   statement;
```

where the bus is equipped with 100,000 words of shared memory taken from the permanent shared heap. As the start condition depends on the current value of `__NUM_PR__` and no processor will spring off, the bus tour `statement` starts if and only if all `__STARTED_PROCS__` processors have entered the bus. An `else` clause is not necessary here, as it would not be reached by any processor.

Synchronous Parallel Critical Sections

The `join` statement enables the implementation of a generalization of the (sequential) critical section introduced in Section 4.2.8, namely, *synchronous parallel critical sections*.

Remember that a (sequential) critical section requires sequentialization to guarantee its deterministic execution by multiple processors. On the other hand, the PRAM mode of computation provides deterministic parallel execution. Hence, a synchronous statement executed by at most *one* group (and by no other processor) at any point of time will produce deterministic results. And just this condition is guaranteed by the `join` statement, where the parallel critical section corresponds to the `busTourStatement`—no other processor can join a bus tour as long as the door is locked. They have to wait (at least) until the processors inside the bus have finished their excursion, that is, have left the parallel critical section.

The question arises in which cases synchronous parallel critical sections may be the better choice, compared to traditional sequentializing critical sections. The prerequisite for exploiting the potential benefits of parallel critical sections—parallel speedup and/or the possibility to apply more sophisticated and expensive algorithms—is the availability of a suitable parallel algorithm for the problem to be solved in the critical section.

As an example let us consider a very simple shared memory allocation scheme where a large slice of available shared heap memory is divided into N equal-sized blocks. Pointers to free blocks are kept in a queue `avail`. Since N is a known constant, this queue is implemented by a shared array, together with two integers `low` and `high`. Index `low % N` points to the first occupied cell in `avail` whereas index `high % N` points to the first free cell.

```
sh void *avail[N];  /*array of N pointers*/
sh int high, low;
```

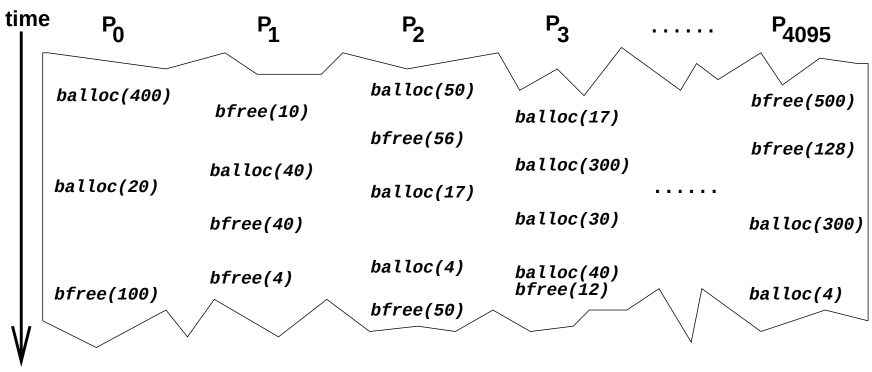


FIGURE 4.13: Nearly simultaneous, asynchronous queries to the block heap memory allocator may be collected and processed in parallel using the `join` statement.

Each processor works on its local task and sometimes issues a query `balloc()` to allocate or `bfree` to release a block of memory, see Figure 4.13 for illustration and Figure 4.15 for the test program. In order to implement operations `void bfree(void *p)` and `void *balloc()`, we introduce an auxiliary function `void *pcs(void *ptr, int mode)` with an extra argument `mode` to distinguish between the two usages.

Now, function `pcs()` is implemented using the `join` construct (see Figure 4.15). Essentially, it consists in applying an `mpadd` operation to variable `high` (in case `mode == FREE`) resp. to variable `low`. Nevertheless, one must handle the case that the queue is empty, as no block can be delivered from an empty queue.

This routine should be contrasted to a conventional, asynchronous implementation that protects access to the shared variables `low` and `high` by locks (see Section 4.2.8). Note that the simplicity of our problem somewhat favors the asynchronous implementation. The runtime figures for both implementations, run with a varying number of processors, are given in Figure 4.16. The breakeven point is here approximately at 128 processors. For smaller numbers of processors, the overhead spent in `join` and for maintaining exact synchronicity during the bus tour outweighs most of the parallel speedup, and the density of queries over

```
/* modes of pcs: */
#define ALLOC 1
#define FREE 0

void bfree(void *p) {
   pcs(p, FREE);
}

void *balloc() {
   return pcs(NULL, ALLOC);
}
```

FIGURE 4.14: A simple test program for the implementation of a parallel critical section.

```
void *pcs( void *ptr, int mode )
{
 int t=0, my_index;
 void *result = NULL;

 join( 1000; (t++ > 2); 1 ) {
   if (mode == FREE) {
      my_index = mpadd( &high, 1 );
      avail[my_index % N] = ptr;
      /*insert ptr into queue of free blocks*/
   }
   if (mode == ALLOC)  {
      my_index = mpadd( &low, 1 );
      if (my_index  >= high) { // can’t get a block
        result = NULL;
        mpadd( &low, -1 ); // correct value of low
      }
      else result = avail[my_index % N];
   }
 }
 else continue;
 return result;
}
```

FIGURE 4.15: An implementation of the parallel critical section in the block allocation example that uses the `join` statement.

time is not dense enough to outperform the asynchronous implementation. For large numbers of processors, the runtime for the asynchronous variant grows linearly in the number of processors, as the sequentialization due to the sequential critical section dominates the execution time.

The `join` variant is likely to perform even better if the queries are not equally distributed over time. If a burst of many nearly simultaneous queries appears within a small time interval, these can be handled by a single bus tour.

The `pcs` example program could be extended for variable block sizes and a more sophisticated block allocation method (*best fit*). The list of free blocks is kept in a search tree, ordered by increasing size, to enable fast access to the smallest block whose size is still greater than or equal to the one demanded for by `balloc()`. In this way, a synchronous algorithm can be developed that allows to process several `balloc`/ `bfree` queries (see Figure 4.13) synchronously in parallel.

Other situations for the application of `join` have been described in *Practical PRAM Programming* [B1], such as synchronous bulk updating of a parallel dictionary based on 2-3 trees [B1, Chap. 8], or parallel rehashing of a hashtable data structure [B1,Chap. 7].

p	asynchronous		using `join`	
1	5390 cc	(21 ms)	6608 cc	(25 ms)
2	5390 cc	(21 ms)	7076 cc	(27 ms)
4	5420 cc	(21 ms)	8764 cc	(34 ms)
8	5666 cc	(22 ms)	9522 cc	(37 ms)
16	5698 cc	(22 ms)	10034 cc	(39 ms)
32	7368 cc	(28 ms)	11538 cc	(45 ms)
64	7712 cc	(30 ms)	11678 cc	(45 ms)
128	11216 cc	(43 ms)	11462 cc	(44 ms)
256	20332 cc	(79 ms)	11432 cc	(44 ms)
512	38406 cc	(150 ms)	11556 cc	(45 ms)
1024	75410 cc	(294 ms)	11636 cc	(45 ms)
2048	149300 cc	(583 ms)	11736 cc	(45 ms)
4096	300500 cc	(1173 ms)	13380 cc	(52 ms)

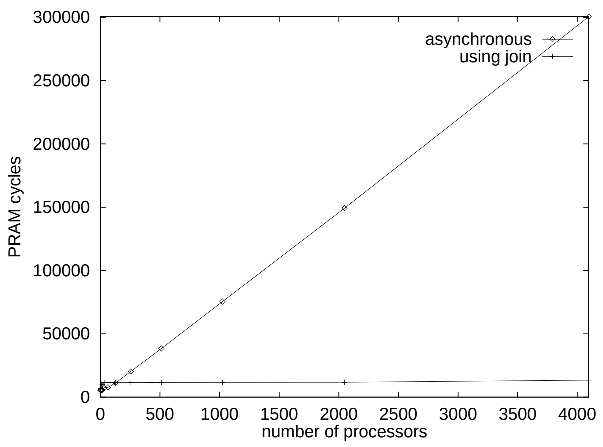


FIGURE 4.16: Timings on the SB-PRAM for the parallel shared block allocator. Each processor submits three `balloc()` resp. `bfree()` queries that are distributed randomly over time. We compare an asynchronous implementation using simple locks (second/third column) with a synchronous one using `join` (fourth/fifth column). The measurements are taken on the SB-PRAM simulator.

Alternatives and possible extensions for `join`

A major discussion in the design phase of the `join` statement was whether it should be a static or a dynamic program construct. In both cases, a bus line is bound to a certain program text location, namely where the `join` keyword appears. Finally, we decided in favor of the former alternative, that is, each bus line has exactly one bus circulating on it, mainly because this substantially simplified the allocation of the semaphore variables and other organizational data structures necessary for the implementation of `join`, and because this scenario is easier to understand for the programmer, in particular for cases where the `join` appears in a loop or in a recursive function.

The other alternative would mean that there be multiple buses circulating on the same bus line. The total number of buses needs not be limited. A new bus can start whenever "enough" (w.r.t. the wait condition) passengers have arrived at the `join`, regardless of the number of buses that are already on tour. Hence, the organizational data structures for each bus have to be allocated dynamically, and the programmer must take care of cases where shared data structures may be simultaneously accessed by different bus groups of the same `join` statement that are asynchronous with respect to each other.

In an object-oriented view, the concept of a bus line with a certain "bus tour" operation could be regarded as a class, with a synchronous parallel `run` method corresponding to the bus tour. The circulating bus with its organizational data structures is then just an instance of that class, that is, an object. The organizational data structures of a bus are, in the former variant, associated with the class, and in the latter, with the object. This object-oriented view has the advantage that one may declare, for instance, an array of bus lines. In order to avoid intricate constructions, bus line objects should be declared and allocated statically and globally. Such a bus line array may indeed be helpful in practice. For instance, we investigated an improved asynchronous shared heap memory management (i.e., implementations of the functions `shmalloc` and `shfree`) with k-way parallel search lists for free memory blocks of each size category; processors may be distributed over the sublists either statically or dynamically by randomization. If eventually multiple subsequent `shmalloc` calls by different processors fail, for instance caused by unequal distribution of free blocks across the k heads, the lists can be reorganized by a synchronous parallel operation on all lists of the same category. If there are m different size categories, and thus m different bus lines are necessary, the same `join` statement must be textually replicated m times[14], possibly in the body of a `switch` statement with the category index as selector. This textual replication is not desirable from a software engineering point of view. With an array of bus line objects, the replication could be avoided, and the category index could be used as an index into the array of bus lines.

4.2.10 Programming Style and Caveats

In general, Fork offers a lot of comfort to the programmer. Nevertheless, there are some problems and common pitfalls the programmer should take care of. These are partly inherent to the language itself and motivated by the design goals of performance and generality, and are partly due to weaknesses of the current implementation that are likely to disappear in a future version of the compiler. We also discuss some aspects of good programming style that may help to increase the clarity of a Fork program and to avoid potential errors.

Absolute versus Relative Group Identifiers

The group ID @ is intended to distinguish the different subgroups of a group with respect to each other. Generally, Fork constructs that create a single subgroup cause the subgroup to inherit its group ID @ from its parent group. Only the group-splitting constructs, the `fork` statement and the private `if` statement, explicitly rename the group IDs of the subgroups.

If an absolutely unique group identifier is required for a group G, for instance, at debugging, we suggest concatenating all @ values of all groups located on the path $(G_0, G_{0,0}, ..., G)$ from the root G_0 of the group hierarchy tree toward G that have been created by group-splitting constructs, including G itself. The resulting sequence of group IDs, maybe stored in a dynamically allocated array, is a globally unique identifier for G, which can be easily seen by induction.

Numbering of Group-Relative Processor Identifiers

In some situations one needs a consecutive numbering of the group-relative processor identifiers $ from 0 to the group size minus one within the current group, for example, for simple parallel loops:

```
int i;
sh int p = groupsize();
for (i=$; i<123; i+=p ) pprintf("%d\n", i);
```

[14]In principle, only the header of the `join` statement must be replicated; the "bus tour" operation may be factored out in a (synchronous) function, and the same can also be applied to the `else` part.

Note that in this example, if consecutive numbering of `$` does not hold, some iterations may be left out while others may be executed multiple times.

In contrast to the group rank `$$`, consecutive numbering for `$` is not enforced automatically by Fork. Even if the programmer has renumbered `$` at creation of a group, some processors that initially belonged to the group may have left it earlier, such as by a `return` statement. Finally, the group-relative processor identifier `$` can be modified arbitrarily by the programmer. Hence, if consecutive numbering is required, we suggest using `$$`.

On the other hand, it is sometimes necessary to avoid changing the numbering of processors even if some processors leave the group. In these cases, `$$` should not be used to identify processors, as it may change accordingly. Instead, the programmer should, in time, fix the desired processor ID, either by

```
sh int p = groupsize();
$ = $$;
```

or by explicit renumbering

```
sh int p = 0;
$ = mpadd( &p, 1 );  // compute p and renumber $ from 0 to p-1
```

Accessing Objects Shared by Different Groups

While all processors inside a leaf group operate strictly synchronously, different leaf groups will, in general, work asynchronously with respect to each other. This may cause problems when two different leaf groups want to access the same shared object. Such a shared object may be either global or allocated by a common predecessor group of these groups in the group hierarchy tree. If the accesses are not atomic, and at least one of the groups wants to *write* to such an object, the accesses to it become a critical section.

For instance, look at the test program in Figure 4.17:

The `if` statement causes the root group to be split into two subgroups; these execute their branches in parallel. Executing this program with four processors, we obtain

```
PRAM P0 = (p0, v0)> g
common.re = 23, common.im = 90
Stop nach 12779 Runden, 851.933 kIps
```

This is probably not what the programmer of this code had in mind; this programmer would probably prefer that access to `common` be atomic, which means that `common` finally has either the value {`45,90`} or {`23,17`}, depending on which of the groups did its assignment last.

Such conflicts can again be managed using locks. Nevertheless, the implementation introduced in Section 4.2.8 cannot be reused without modification, as all processors of a group should get access to the critical section simultaneously and synchronously. Thus we need a *group lock* mechanism.

A simple solution using the existing lock mechanisms may be to select for each group one processor that should act as a representative for its group and be responsible for the

```
sh struct cmplx_int {
  int re;
  int im;
} common;

void main( void )
{
 start {
   if ($ % 2) {
     common.re = 45;
     common.im = 2 * common.re;
   }
   else {
     common.im = 17;
     common.re = 23;
   }
   seq printf("common.re = %d, common.im = %d\n",
              common.re, common.im);
 }
}
```

FIGURE 4.17: Example for non-atomic, asynchronous access of a data structure by different groups.

locking/unlocking mechanism. This is done by masking the locking/unlocking operations by `seq` statements, as shown in Figure 4.18.
Now we get the desired output:

```
PRAM P0 = (p0, v0)> g
common.a = 45, common.b = 90
Stop nach 13179 Runden, 753.086 kIps
```

Jumps

All statements causing non-block-structured control flow, namely, `goto`, `break`, `return`, and `continue`, appear to be problematic with respect to synchronous execution, because the jumping processors may miss to enter or leave groups on the "normal" way (via subgroup construction or subgroup merge).

However, for jumps of type `break`, `continue`, and `return`, the target group is statically known: it is a predecessor of the current leaf group in the group hierarchy tree. For `break` and `return`, the compiler generates additional code that guarantees that the processors to exit the proper number of subgroups and that the strict synchronicity invariant is maintained. `continue` is allowed only in asynchronous loops.

Jumps across synchronization points usually will introduce a deadlock. In particular, `goto` jumps whose target is not in the current group's activity range are strongly discouraged; the target group may not yet have been created at the time of executing the jump. Even worse, the target group may not be known at compile time.

```
sh struct cmplx_int {
  int re;
  int im;
  fair_lock fglock;   /* used as a fair group lock */
} common;

void main( void )
{
 start {
   seq fair_lock_init(&(common.fglock));
   if ($ % 2) {
      seq  fair_lockup( &(common.fglock));
      common.re = 45;
      common.im = 2 * common.re;
      seq  fair_unlock( &(common.fglock));
   }
   else {
      seq  fair_lockup( &(common.fglock));
      common.im = 17;
      common.re = 23;
      seq  fair_unlock( &(common.fglock));
   }
   seq printf("common.re = %d, common.im = %d\n",
              common.re, common.im);
 }
}
```

FIGURE 4.18: Asynchronous access to the group-global shared variable `common` by different groups is protected by a group lock.

On the other hand, synchronous `goto` jumps within the current group's activity range should not lead to problems, as all processors jump simultaneously to the same target position, and will still reach the official release point of the group. For this reason, `goto` jumps are under the programmer's responsibility.

Shared Memory Fragmentation

As static program analysis for C (and thus also for Fork) is generally hard, even a clever compiler will be often unable to analyze the space requirements of the subgroups and thus, following a worst-case strategy, assign equal-sized slices of the parent group's shared memory subsection to the child groups. Hence, it is not wise to have multiple, nested `fork` or private `if` statements on the recursive branch of a recursive procedure (e.g., parallel depth-first-search), unless such a construction is unavoidable. Otherwise, after only very few recursion steps, the remaining shared memory fraction of each subgroup will reach an impracticably small size, which results in a shared stack overflow.

A programmer who statically knows that a group will consist of only one processor should switch to asynchronous mode because this avoids the expensive overhead of continued subgroup formation and throttling of computation by continued shared memory space fragmentation.

The `asm` Statement

As some older C compilers, the current Fork implementation offers the `asm` statement that allows the inclusion of inline assembler sections directly into the C source.

`asm()` takes a constant string as argument; the string is directly copied to the assembler file produced by the compiler. Errors in the assembler string will be reported by the assembler `prass`.

Additionally, accesses to a local variable[15] or function parameter `var` can be expressed in this string as `%var`, as in the following example:

```
unsigned int rc = lock->readercounter;
asm("bmc    0\n\
     gethi  0x40000000,r31 /*__RW_FLAG__*/\n\
     syncor r31,%rc\n\
     nop    /*delay*/");
```

Fork admits this low-level programming with inline assembler for efficiency reasons, but it should be used only as a last resort, as it makes programs less portable and less understandable. Moreover, it disables any compiler optimizations on that part of the program, since the compiler does not really “understand” what happens inside the `asm` code—it just trusts the programmer.

Short-Circuit Evaluation of && and `||`

Fork follows the C semantics of evaluating the binary logical operators && and `||` in a short-circuit way. This means that, for instance, in

```
sync void crash ( pr int a )
{
    pr int c = a && foo(...);
    ...
}
```

the call to `foo()` is not executed by the processors that evaluated `a` to a zero value. In sequential C programming, this may be a pitfall for programmers if side effects in the argument's evaluation or in the execution of `foo()` are not foreseen. In the synchronous mode of Fork, this may additionally lead to unintended asynchronity and thus to race conditions, or even to a deadlock if `foo()` contains a synchronization point. As there exist obvious workarounds to avoid such situations, the current implementation renounces on an explicit splitting of the

[15]In the current implementation this replacement mechanism works only for private and shared formal parameters, for private local variables, and for shared local variables defined at the top level of a synchronous function. Global variables (private and shared) and block local shared variables cannot be resolved by this method. These restrictions are caused by the fact that addresses of global variables cannot be generated within a single SB-PRAM instruction, and that block local shared variables do not have a unique addressing scheme.

current group into subgroups to maintain strict synchronicity. Rather, the compiler emits a warning in this case.

A straightforward workaround would be to rewrite the logical expression above as

```
sync void no_crash ( pr int a )
{
   pr int c;
   if (a)                /*group splitting*/
          c = foo(...);
   else   c = 0;
   ...
}
```

As an alternative, one may, if the side effects of short-circuit evaluation are not explicitly desired, replace a logical operation `A&&B` by the equivalent expression

```
((A?1:0)*(B?1:0))
```

and the logical operation `A||B` by the equivalent bitwise operation

```
(A|B)?1:0)
```

which are free of race conditions.

Furthermore, one may, if possible, switch to asynchronous mode during the evaluation of the logical expression if `foo()` could be rewritten as an asynchronous function.

4.2.11 Graphical Trace File Visualization

Fork offers the possibility to trace and visualize program runs.

A ***(trace) event*** is a tuple (typ, pid, t, g) that denotes for a processor *pid* belonging to a current group g a change in the type of its activity to *typ* at time t. Possible types of activity during the execution of Fork programs are: (1) doing "useful" work, (2) waiting at barriers, (3) waiting at lock acquire operations, and (4) group reorganization. The user may define further events that may be of interest for the application program, such as sending or receiving messages, accesses to user-defined data structures, shared memory allocation, etc. Events that occur during program execution within a user-defined time interval are recorded in a central *trace buffer* that can later be written to a file and processed further (post-mortem analysis).

In order to activate the tracing feature, the user program must be compiled and linked with the `-T` option. The program is then instrumented with additional code that records for each event the values of *typ*, *pid*, t, and g, and stores them in the trace buffer.

The start and endpoints of tracing can be set arbitrarily in the source program, as follows:

```
start {
 initTracing( 100000 );  // allocates central trace buffer
 ... (no tracing here)
 startTracing();         // starts tracing
```

```
void main(void)
{
 start {
   initTracing( 10000 );
   startTracing();
   seq fair_lock_init(&(common.fglock));
   if ($ % 2) {
      seq  fair_lockup( &(common.fglock));
      common.re = 45;
      common.im = 2 * common.re;
      seq  fair_unlock( &(common.fglock));
   }
   else {
      seq  fair_lockup( &(common.fglock));
      common.im = 17;
      common.re = 23;
      seq  fair_unlock( &(common.fglock));
   }
   stopTracing();
   seq
      printf("common.re = %d, common.im = %d\n",
             common.re, common.im);
   writeTraceFile( "glock.trv", "group lock example" );
 }
}
```

FIGURE 4.19: The `main` function of the group lock example program, with calls to the tracing routines inserted.

```
 ... (automatically recording events in trace buffer)
 stopTracing();          // stops tracing
 ... (no tracing here)
 writeTraceFile("filename.trv","title"); //writes trace buffer
}
```

The synchronous function `startTracing` starts the recording of events. A default trace buffer size is used unless the call to `startTracing` is preceded by a call `initTracing(`L`)` that reallocates the central trace buffer with length L (in memory words) on the permanent shared heap. Note that, in the current implementation, each event requires 3 words in the trace buffer, and that the number of events usually depends linearly on the number of processors.

After calling the synchronous function `stopTracing`, events are no longer recorded in the trace buffer. The events stored in the trace buffer can be written to a file (*trace file*) by calling the synchronous `writeTraceFile` function, which takes the trace file name and an optional title string as parameters. Writing the trace buffer to a file is done in parallel, at acceptable speed.

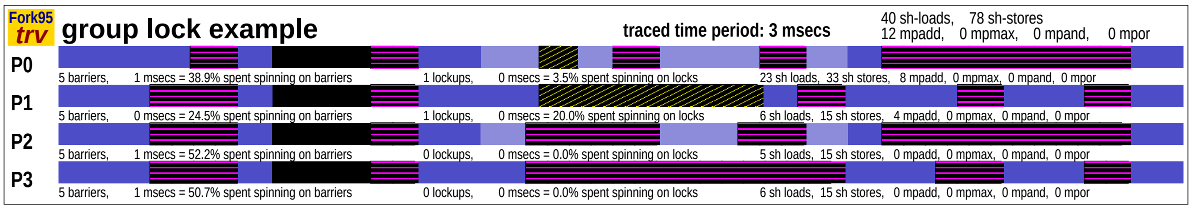


FIGURE 4.20: Processor–time diagram generated by `trv` for the group lock program in Figure 4.19.

Once the execution of the instrumented Fork program is finished, the trace file written is submitted to the `trv` tool (in `fork/bin`) by

```
trv filename.trv
```

where `trv` is a utility program supplied with the Fork compiler that generates processor–time diagrams from traces of Fork programs. `trv` creates a file named `filename.fig` in FIG format that can be viewed and edited by `xfig` (version 3.0 or higher), a free, vector-oriented drawing program for X-Windows, or converted to numerous other graphics file formats using `fig2dev`. For instance, a postscript version of `filename.fig` is created by

```
fig2dev -Lps filename.fig > filename.ps
```

For different output devices, different colors may be preferable. For instance, on a color screen or color printer one prefers different, bright colors, while these are hard to distinguish when printed on a grayscale printer. For this reason, there is a color variant of `trv`, called `trvc`, while `trv` itself produces colors that are more suitable for grayscale devices, as the trace images printed in this book[16].

In a processor–time diagram (also known as a *Gantt chart*) generated by `trv` resp. `trvc`, a time bar is drawn from left to right for each processor (see Figures 4.20 and 4.21). The left end corresponds to the point of time where `startTracing` was called, and the right end corresponds to the point of time where `stopTracing` was called. A time bar for a processor

[16]Color versions of the `trv`-generated diagrams in this section can be found on the Fork web page, `www.informatik.uni-trier.de/~kessler/fork`.

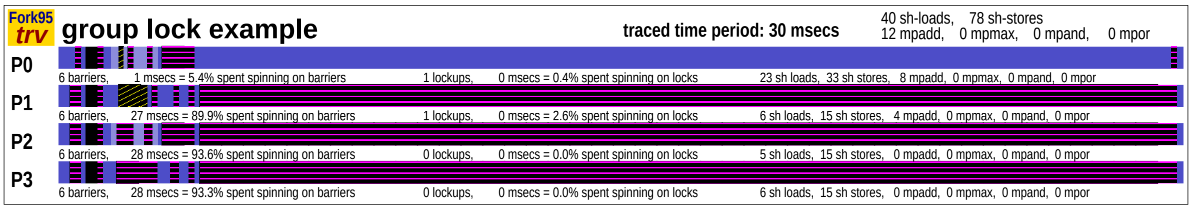


FIGURE 4.21: Processor–time diagram generated by `trv` for the group lock program in Figure 4.19, where the time-consuming `printf` call is integrated in the traced time period. The diagram in Figure 4.20 corresponds to the first few milliseconds of the traced time period.

is tiled by a sequence of colored rectangles that correspond to phases. A ***phase*** is the time interval between two subsequent events of the same processor. The color of the rectangle representing a phase indicates the type of activity of the processor in this phase:

- Phases of "useful" work are drawn in a color chosen in the range between blue and green. All processors of the same group have the same color (`trvc`) resp. intensity (`trv`).
- Work phases in groups created by the `join` statement are drawn in light green (`trvc`) resp. light shaded (`trv`).
- Idle phases due to waiting at a barrier (either implicit or explicit) are drawn in red (`trvc`) resp. black with horizontal stripes (`trv`).
- Idle phases due to waiting at a lock acquire operation, like `simple_lockup()`, `fair_lockup()`, or `rw_lockup()`, are drawn in yellow (`trvc`) resp. black with diagonal stripes (`trv`).
- Wait phases at entry to a `join` statement and subgroup creation phases are drawn in black.
- White is reserved for other events in Fork standard libraries that may be in the future considered for tracing, such as memory allocation or message passing.

The default colors may be changed by the user.

User-defined events are also possible, by calling the routine `traceEntry(`*typ*`)` routine with an integer parameter *typ* indicating a user-defined event type between 8 and 31. Phases started by such a user event will be drawn in the corresponding color taken from the standard color table of `xfig`. The 8 predefined event types (between 0 and 7) can be looked up in `fork.h` unless explicitly specified by the user.

The image generated by `trv` also shows detailed information on the number of barriers, lockups, accumulated waiting times at barriers and lockups, and the numbers of various shared memory access operations for each processor below its time bar. The overall numbers of shared memory accesses are shown in the top line.

As an example, let us reconsider the group lock example program from Section 4.2.10. The calls to the tracing routines are set such that the lengthy `printf` call is not traced, see the code in Figure 4.19. The resulting trace image is shown in Figure 4.20. This diagram offers a very fine resolution in time, as the traced time period is only about 3 ms, which corresponds to only a few CPU cycles per millimeter on the horizontal axis. A much coarser resolution (by a factor of 10) is already obtained if the time-consuming `printf` call is integrated in the traced time interval, because the `printf` call executed by processor 0 dominates the execution time; see Figure 4.21.

Discussion The two-step approach with the trace buffer as a temporary data structure for event data was chosen to keep the overhead incurred by tracing during program execution as small as possible. This is particularly important for asynchronous applications that access

```
 701283 871425 574623 358953 736578 1005344 427186 175294
 983639 879684 597400 261401 605659 81098 248455 405579
Sorting...
 81098 175294 248455 261401 358953 405579 427186 574623
 597400 605659 701283 736578 871425 879684 983639 1005344
```

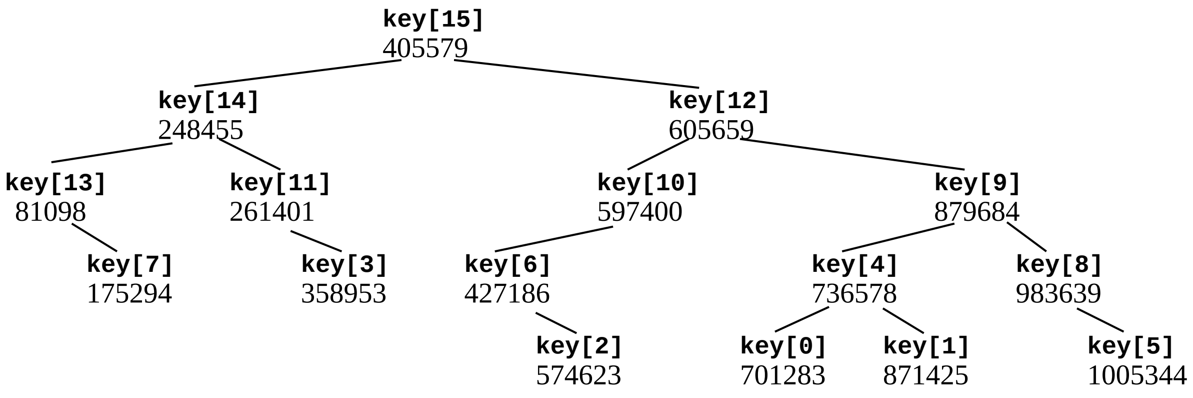


FIGURE 4.22: Output of an example run with $N = 16$ randomly generated integers, and the tree built by the Quicksort program.

shared data structures, since the tracing overhead may introduce delays that change the order of access compared to a nontracing version of the same program.

Due to the instrumentation, a program takes slightly more time when recording tracing information. However, the tracing overhead per event (tracebuffer entry) is reasonably small (see Section 5.1.13]). In any case, the execution time of programs that are not compiled and linked with `-T` is not affected by the tracing feature.

`trv` is tailored toward the usage with Fork and the SB-PRAM. There exist several other performance visualization tools for parallel processing, which are designed for and used with mainly message passing environments like MPI and PVM. Examples are Paragraph [HE91], Upshot [HL91], Pablo [RAN$^+$93], AIMS [YHL93], VAMPIR [Pal96], VisTool from the TRAPPER system [SSK95], or vendor-specific systems like VT [IBM] or ParAide [RAA$^+$93]. While these systems often support further statistical analysis and multiple types of diagrams and views of trace data, `trv` is a simple, low-overhead program that focuses just on the processor–time diagram.

Zooming in trace images generated by `trv` is possible within `xfig` by adjusting the zoom scale (up to a certain limit), and by narrowing the time interval between the `startTracing` and `stopTracing` calls before running the program. Arbitrary zooming and scrolling could be provided by an additional viewing tool running on top of `trv`. An extension of `trv` by adding arcs for messages sent in a user-supplied message passing library on top of Fork, such as the implementation of the MPI core routines presented in [B1, Chap. 7.6] is straightforward by making use of the user-definable events.

Some examples for the application of `trv`

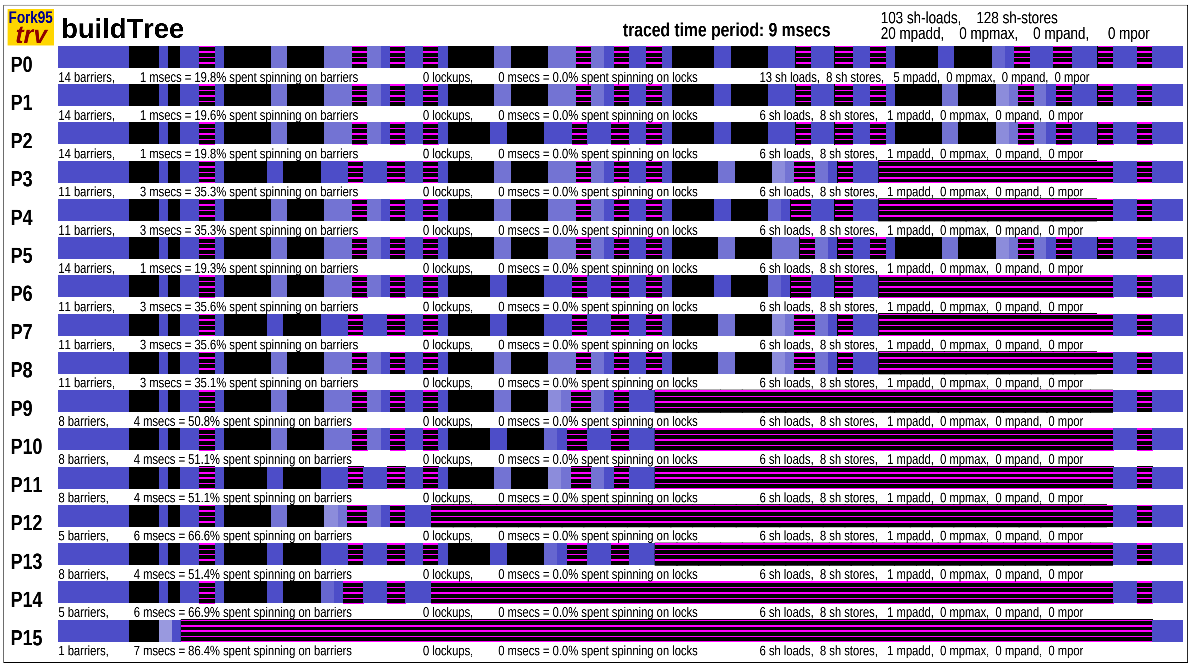
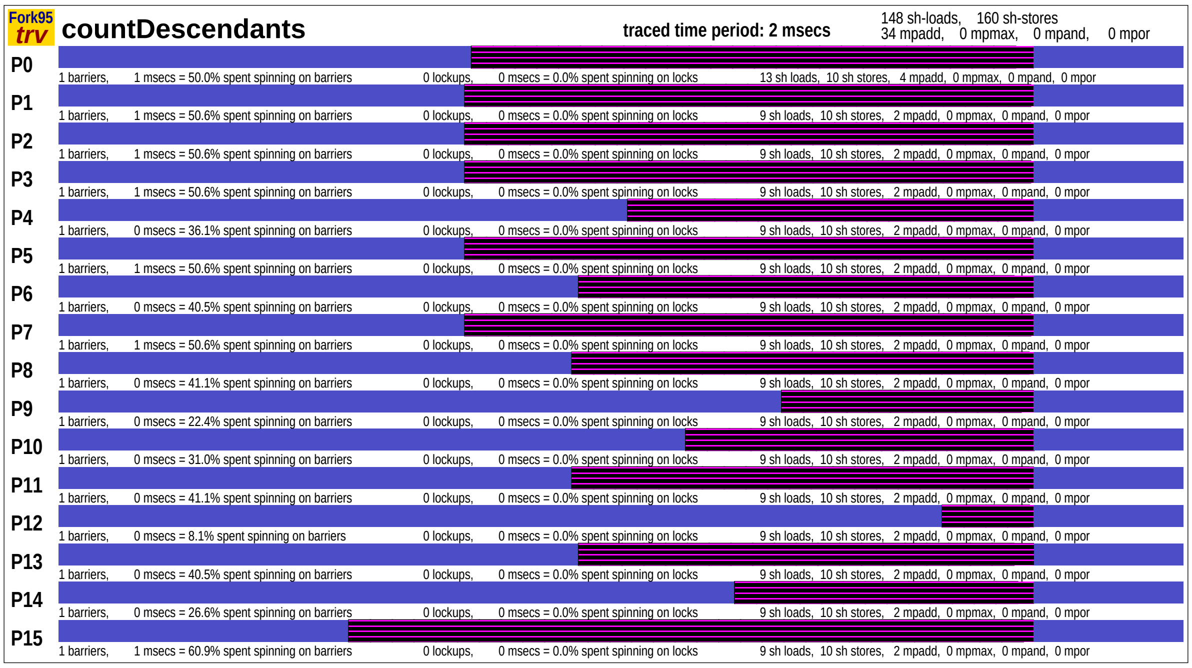


FIGURE 4.23: Processor–time diagram for the tree construction phase, run with $N = 16$ processors. It can be well observed that processors that have found the position of “their” `key` element in the tree leave the `while` loop and wait for the others. Processor 15 is associated with the root node of the tree. The `while` loop makes four iterations, one for each level of the tree (see Figure 4.22).

CRCW Quicksort algorithm As a first nontrivial example, we consider the CRCW Quicksort algorithm by Chlebus and Vrto [CV91], see also Chapter 1 of *Practical PRAM Programming* [B1].

N items, initially stored in an array, are to be sorted by their *keys* (e.g., integers). In sequential, this could be done by the well-known Quicksort algorithm [Hoa62]: if N is one, the array is sorted, and we are done; otherwise we choose a key from the array, called the *pivot*, partition the array into three subarrays consisting of the keys smaller than, equal to, and larger than the pivot, respectively, and finish by applying the algorithm recursively to the two subarrays with the smaller and the larger keys. If the pivot is chosen randomly, or the input keys are assumed to be stored in random order in the input array, Quicksort can quite easily be shown to run in an expected number of steps bounded by $O(N \log N)$ [CV91].

The CRCW Quicksort algorithm by Chlebus and Vrto [CV91] assumes that N processors are available, each one associated with an element of the input array. The computation constructs a binary tree that corresponds to the tree-like structure of the recursive calls of the Quicksort algorithm, and stores it in index arrays. Each tree node corresponds to a contiguous subarray to be sorted. Leaves of the tree correspond to sorted arrays storing only a single integer. The tree is constructed top-down. First a root is chosen, and all remaining keys/processors are made children of this root. The child processors compare in parallel their key to that of the parent node to determine whether they belong to the left or to the right subtree. Further, for



```
P0   1 barriers,   1 msecs = 50.0% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks   13 sh loads, 10 sh stores,  4 mpadd, 0 mpmax, 0 mpand, 0 mpor
P1   1 barriers,   1 msecs = 50.6% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P2   1 barriers,   1 msecs = 50.6% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P3   1 barriers,   1 msecs = 50.6% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P4   1 barriers,   0 msecs = 36.1% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P5   1 barriers,   1 msecs = 50.6% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P6   1 barriers,   0 msecs = 40.5% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P7   1 barriers,   1 msecs = 50.6% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P8   1 barriers,   0 msecs = 41.1% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P9   1 barriers,   0 msecs = 22.4% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P10  1 barriers,   0 msecs = 31.0% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P11  1 barriers,   0 msecs = 41.1% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P12  1 barriers,   0 msecs = 8.1% spent spinning on barriers    0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P13  1 barriers,   0 msecs = 40.5% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P14  1 barriers,   0 msecs = 26.6% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
P15  1 barriers,   1 msecs = 60.9% spent spinning on barriers   0 lockups,   0 msecs = 0.0% spent spinning on locks    9 sh loads, 10 sh stores,  2 mpadd, 0 mpmax, 0 mpand, 0 mpor
```

FIGURE 4.24: Processor–time diagram for a run of the top-down pass in the parallel Quicksort implementation, with $N = 16$ processors. The grey parts of the processor–time bars visualize "useful" work of a processor. It can be well observed that this time is proportional to the height of the associated node in the tree (except for the root node that does not participate).

both left and right subtrees a new parent (root) node is chosen, and the remaining processors are all made children of these nodes. This process continues in a `while` loop until each node has found its proper place in the tree. We label the nodes of the computation tree by the pivot element. After the tree computation phase, each processor can determine from the tree where its key has to go in order to put the input keys into sorted order. These indices are computed by a parallel bottom-up traversal, followed by a parallel top-down traversal of the tree. The implementation of the algorithm in Fork is given in [B1, Chap. 1].

Figure 4.22 shows a run of the Quicksort program with $N = 16$ processors for an array of N randomly generated integers. The constructed tree is also shown in Figure 4.22. Figure 4.23 shows the `trv`-generated image for the tree construction phase. Figure 4.24 visualizes the top-down pass, the bottom-up pass looks similar.

Synchronous N-Queens program The N-Queens problems consists of finding all possibilities of placing N queens on a $N \times N$ checkerboard such that they do not interfere with each other. A (sequential) exhaustive search method traverses the solution space as follows: Initially, there are N possibilities for by placing a queen in the first row. For each of these, one checks recursively for all possibilities to place another queen in the second row without interfering with that in the first row, and so on. Recursion is limited in depth by N. Whenever all N queens could be placed in a depth-N configuration, this solution is reported.

The synchronous parallel implementation in Fork searches this decision tree in parallel:

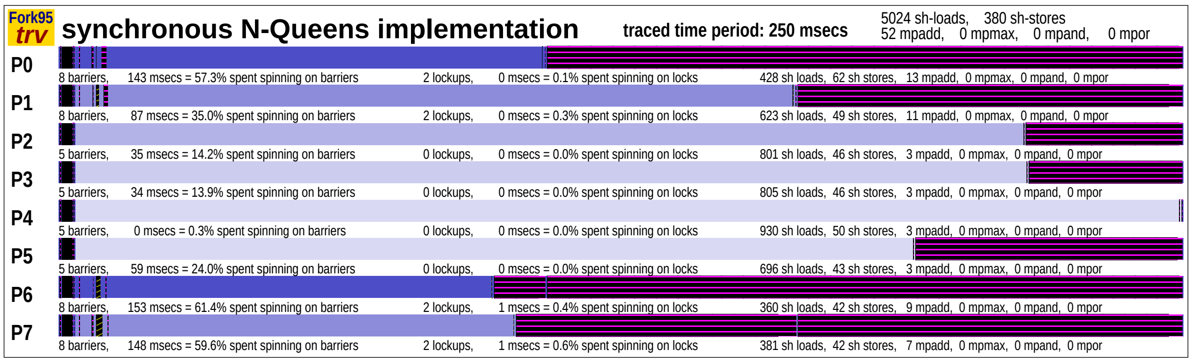


FIGURE 4.25: Processor–time diagram for the synchronous N-Queens program with $N = 6$ and $p = 8$.

First, it splits the initial group of p processors into N subgroups of size $\lceil N/p \rceil$ or $\lfloor N/p \rfloor$, and each subgroup then performs a recursive call of the search routine independently. The same method is applied in the recursive search routine. As soon as a subgroup reaches size 1, the algorithm switches to the sequential version of the search routine.

Figure 4.25 shows the diagram obtained for $N = 6$ with $p = 8$ processors; the trace file contains 144 events. It appears that the load distribution is quite uneven (long idle times at barriers for the processors belonging to the first two subgroups).

Asynchronous N-Queens program This implementation solves the N-Queens program by maintaining a shared queue of tasks, implemented as a parallel FIFO queue [B1, Chap. 7]. A *task* is represented by a partial positioning of $1 \le k < N$ queens which contains no interference, that is, it corresponds to a call of the recursive search function. A call to a testing function dequeues a task, checks all possibilities for placing a new queen, and generates and enqueues a subtask for the feasible cases. If all N queens could be placed for a subtask, the corresponding checkerboard is printed. Simultaneos output of several checkerboards computed at approximately the same time is arranged by collecting processors that want to print a solution by the `join` statement of Fork, over a certain time interval. The synchronous parallel output routine is started when either a maximum number (M) of solutions has been found or a time counter has exceeded a certain value. Hence, up to M solutions are printed concurrently to the screen.

As example, we run $p = 8$ processors on the case $N = 6$, which has 4 solutions (see the screen output in the Appendix, Figure 4.26). The generated trace has 2518 events, most of which are due to locking the shared FIFO queue when enqueuing or dequeuing a task. As Figure 4.27 shows, the first two solutions happen to be found at the same time (by P2 and P7) and are printed concurrently, while the next two are printed separately when they have been computed (by P1, P3). Load balancing is very good, due to the fine-grained tasks and the shared task queue. Most of the time in the join body is consumed by a `printf` call that prints the headline “Next k solutions...”, see Figure 4.26.

```
Computing solutions to the 6-Queens problem...

-------------------- Next 2 solutions (1..2): ----------------
|..Q...|...Q..
|.....Q|Q.....
|.Q....|....Q.
|....Q.|.Q....
|Q.....|.....Q
|...Q..|..Q...
-------------------- Next 1 solutions (3..3): ----------------
|.Q....
|...Q..
|.....Q
|Q.....
|..Q...
|....Q.
-------------------- Next 1 solutions (4..4): ----------------
|....Q.
|..Q...
|Q.....
|.....Q
|...Q..
|.Q....

Solutions: 4
TIME: 101428 cc
```

FIGURE 4.26: The screen output of the asynchronous N-Queens program for $N = 6$. By collecting near-simultaneously found solutions with `join`, parallel screen output leads to less sequentialization.

Further examples More `trv`-generated images, together with the corresponding Fork source programs, are available at the Fork homepage

`http://www.informatik.uni-trier.de/~kessler/fork.html`

4.2.12 The History of Fork

The design of Fork has been developed in several evolutionary steps from 1989 to 1999, as shown in Figure 4.28. In 1999 the language has reached a final state with version 2.0 of the variant Fork95, which is referred to simply as Fork and described in this book.

The first approach to the design of Fork, called FORK [HSS92], was a rather theoretical one. Pointers, dynamic arrays, nontrivial data types, input/output, and nonstructured control flow were sacrificed in order to facilitate correctness proofs [Sch91, RS92] and static program analysis [Käp92, Wel92, Sei93]. In this way, however, the language became completely unusable. A compiler [Käp92, Wel92, Lil93, Sei93] was never completed.

In order to provide a full-fledged language for real use, we have added all the language features that are well known from sequential imperative programming. We decided to build

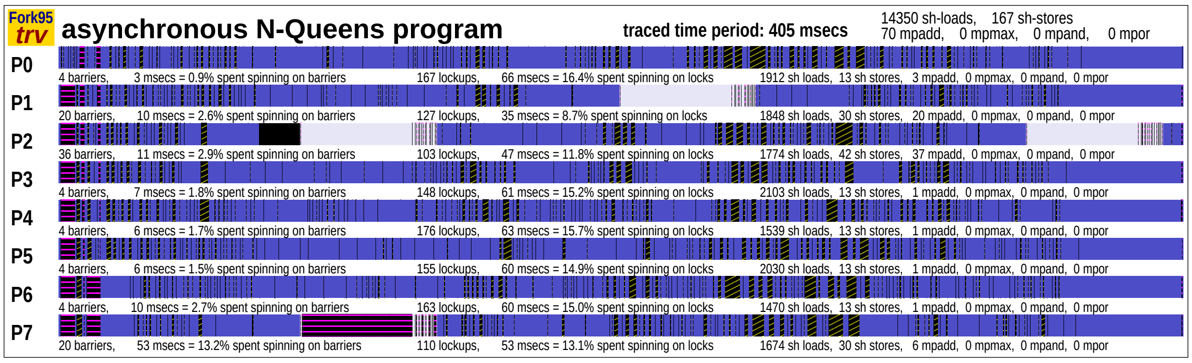


FIGURE 4.27: Processor–time diagram for the asynchronous N-Queens program with $N = 6$ and $p = 8$. P2 and P7 join for concurrent output. P7 waits for P2 to print the headline before they jointly process the N output lines.

upon an existing and well-known sequential language since this enables better acceptance of the language and facilitates the integration of existing sequential code into parallel implementations. Thus, for the new Fork dialect Fork95 we decided [M4] to extend the ANSI-C syntax—instead of clinging to the original one which was an idiosyncratic mixture of Pascal and Modula. This also meant that (for the sequential parts) we had to adopt C's philosophy.

With Fork95 we introduced the possibility of locally asynchronous computation to save synchronization points and to enable more freedom of choice for the programming model. First existing only locally inside the bodies of `farm` statements [M4] that were originally intended only as an instrument for performance tuning and encapsulating of I/O, the asynchronous mode soon became an equally important component of the language [C8,J3]. Asynchronous and synchronous mode complement each other. Static association of synchronous resp. asynchronous mode of execution with program regions avoids confusion and eliminates a potential source of bugs. Switching forth and back became highly flexible by the `join` statement that was added in 1997 [C14,I4]. The third type of synchronicity, `straight`, was added in 1999 [B1] to avoid duplicate definitions of functions like `syncadd` that should be accessible in synchronous as well as asynchronous mode.

A second important decision was to abandon the tremendous runtime overhead of additional processor emulation (see Section 5.2.1) by limiting the number of processes to the hardware resources, resulting in a very lean code generation and runtime system. In the old

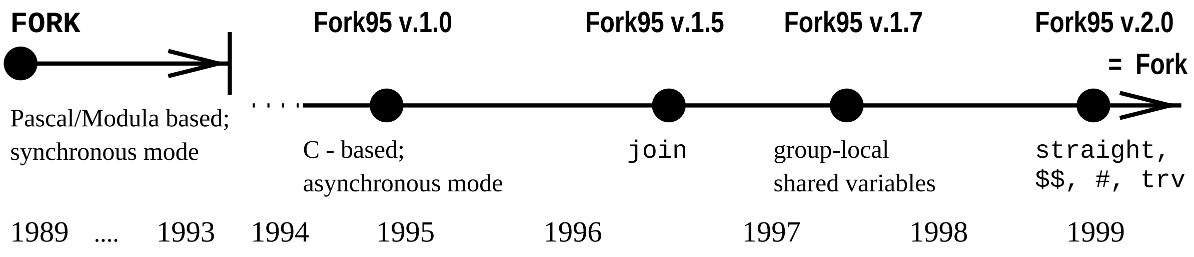


FIGURE 4.28: The evolution of Fork.

`FORK` proposal [HSS92], the `start` statement was used with a shared parameter expression, `start(`e`)`, to start a sub-PRAM of e processors while freezing the current configuration. If the value of e was allowed to exceed the hardware resources, additional PRAM processors had to be emulated in software; otherwise, this did not really add more expressiveness to the language design. As we shall see in Chapter 5, the combination of exact synchronous program execution with virtual processing, especially for a weakly typed base language such as C, is extremely expensive to realize. In pm2 the same decision was made.

Third, we restricted the language support for exact synchronicity to the leaf groups only. In contrast, the old `FORK` proposal also supported larger group hierarchies running exactly synchronous. At any point of the program and for each (leaf) group, the compiler kept track of the so-called maximally synchronous group; namely, the root of the subtree in the group tree all of whose leaf groups currently run exactly synchronous with this group considered (see Section 5.2.1). Originally intended to avoid some unnecessary synchronization points, it appeared that, because of the compile time and runtime overhead, this theoretical flourish was not useful in practice, decreased performance, and confused the programmer.

Fourth, Fork95 supports pointers; the declaration and usage of pointers is simple and very flexible. Unfortunately, with our decision for C we also inherited its unpleasant, weak type system. This disables many compile-time analyses that were possible with the old proposal [Wel92, Käp92, Sei93]. Nevertheless we think that this is a price to be paid for practically useful parallel programming.

Later evolution steps added to Fork95 the `join` construct [C14,I4], the possibility to declare shared variables within any block in a synchronous region, and the declaration of `straight` synchronicity.

With these extensions, the language, now named just Fork, has reached a stable state. A further development, as discussed in Section 4.2.13, would imply fundamental changes of the general concept and/or in the basis language.

4.2.13 The Future of Fork

Object-oriented programming provides cleaner programming interfaces and simplifies the development of software libraries. Hence, we would favor an object-oriented extension of Fork, such as one based on a subset of C++.

The implementation of Fork for non-PRAM target machines will be discussed in Section 5.2. Unfortunately, it appears that the Fork design—in particular synchronous execution, concurrent write, and sequential shared memory consistency—causes an implementation for asynchronous shared memory machines or even distributed memory architectures to produce quite inefficient target code, as we will see in Section 5.2. As a consequence, we have developed alternative language designs that relax some of these features in order to allow for easier and more efficient implementations.

For instance, ForkLight [C16], which we discuss in Section 4.4, relaxes the synchronous execution from the operator level to the (extended) basic block level (*control-synchronous mode of execution*). The programmer becomes responsible to protect data dependencies within basic blocks by additional `barrier` statements or suitable semaphore mechanisms. Nevertheless, sequential shared memory consistency is maintained.

NestStep [C18,I8,J7], which we discuss in Section 4.5, relaxes synchronicity to even larger units that are defined by an explicit `step` statement, which is a generalized and nestable variant of the BSP superstep. The `neststep` statement of NestStep roughly corresponds to the combination of the `start`, `fork`, and `farm` statements of Fork, that is, it allows to define subgroups that install scopes of barriers and sharing. Also, NestStep no longer assumes the availability of a shared memory. Instead, it emulates a distributed shared memory on top of a message passing system, where memory consistency is relaxed to `step` boundaries, in compliance with the BSP model.

4.3 A Plea for Structured Parallel Programming in Fork

Structured parallel programming [Col89, Pel98] can be enforced by restricting the many ways of expressing parallelism to compositions of only a few, predefined patterns, so-called skeletons. ***Skeletons*** [Col89, DFH$^+$93] are generic, portable, and reusable basic program building blocks for which parallel implementations may be available. They are typically derived from higher-order functions as known from functional programming languages. A skeleton-based parallel programming system, like P3L [BDO$^+$95, Pel98], SCL [DFH$^+$93, DGTY95], SkIL [BK96], HSM [Mar97], or $\mathcal{HDC}$ [HLG$^+$99], usually provides a relatively small, fixed set of skeletons. Each skeleton represents a unique way of exploiting parallelism in a specifically organized type of computation, such as data parallelism, parallel divide-and-conquer, or pipelining. By composing these, the programmer can build a structured high-level specification of parallel programs. The system can exploit this knowledge about the structure of the parallel computation for automatic program transformation [GP99], resource scheduling [BDP94], and mapping. Performance prediction is also enhanced by composing the known performance prediction functions of the skeletons accordingly. The appropriate set of skeletons, their degree of composability, generality, and architecture independence, and the best ways to support them in programming languages have been intensively researched in the 1990s and are still issues of current research.

In *Practical PRAM Programming* [B1, Chap. 7] we provide Fork implementations for several important skeletons. Table 4.5 gives a survey. In this chapter we focus on skeleton functions for dataparallel computations and reductions, for sake of brevity. Note that we call these Fork functions that simulate skeleton behavior just *skeleton functions,* not skeletons. We call the programming style characterized by expressing parallelism (where possible) by using only these skeleton functions the *skeleton-oriented style of parallel programming* in Fork, which uses only the features available in the basis language. This should be seen in contrast to *parallel programming in a skeleton language,* which additionally requires special skeleton language elements.

Composition of skeletons may be either nonhierarchical, by sequencing using temporary variables to transport intermediate results, or hierarchical by (conceptually) nesting skeleton functions, that is, by building a new, hierarchically composed function by (virtually) inserting the code of one skeleton as a parameter into that of another one. This enables the elegant compositional specification of multiple levels of parallelism. In a declarative programming environment, such as in functional languages or separate skeleton languages, hierarchical

Skeleton function	Nestable	Meaning	Introduced in
`map`	–	Unary dataparallel operation	Section 4.3.1
`map1`	–	Scalar–vector dataparallel operation	[B1, Chap. 7]
`map2`	–	Binary dataparallel operation	Section 4.3.1
`Map`	$\surd$	Unary dataparallel operation	[B1, Chap. 7]
`Map2`	$\surd$	Binary dataparallel operation	Section 4.3.1
`reduce`	–	Parallel reduction	Section 4.3.3
`Reduce`	$\surd$	Parallel reduction	Section 4.3.3
`prefix`	–	Parallel prefix operation	[B1, Chap. 7.2]
`Prefix`	$\surd$	Parallel prefix operation	[B1, Chap. 7.2]
`divide_conquer`	–	Parallel divide-and-conquer computation	[B1, Chap. 7.3]
`taskQueue`	–	Asynchronous parallel task queue	[B1, Chap. 7.5]
`pipe`	–	Pipelining with global step signal	[B1, Chap. 7.7]
`Pipe`	$\surd$	Pipelining with global step signal	[B1, Chap. 7.7]

TABLE 4.5: Survey of the skeleton functions that have been implemented for Fork, see also the description in *Practical PRAM Programming* [B1].

composition gives the code generator more freedom of choice for automatic transformations and for efficient resource utilization, such as the decision of how many parallel processors to spend at which level of the compositional hierarchy. Ideally, the cost estimations of the composed function could be composed correspondingly from the cost estimation functions of the basic skeletons.

The exploitation of *nested parallelism* specified by such a hierarchical composition is quite straightforward if a fork-join mechanism for recursive spawning of parallel activities is applicable. In that case, each thread executing the outer skeleton spawns a set of new threads that execute also the inner skeleton in parallel. This may result in very fine-grained parallel execution and shifts the burden of load balancing and scheduling to the run-time system, which may incur tremendous space and time overhead.

In a SPMD language like Fork, nested parallelism can be exploited by suitable group splitting. A common strategy is to exploit as much parallelism as possible at the outer levels of nested parallelism, use group splitting to encapsulate inner levels of nested parallelism, and switch to a sequential code variant as soon as a group consists of only one processor.

With the `fork` statement, Fork provides substantial support of nested parallelism. Hence it is possible to define statically and dynamically nestable skeleton functions. Static hierarchical composition of skeleton functions will be discussed in more detail in Section 4.3.1; in short, hierarchical composition in Fork works by encapsulating the inner skeleton function into the definition of a synchronous function f that is passed as function parameter to the outer skeleton function G. We provide a nonnestable and a nestable variant for most skeleton functions. Dynamically nested parallelism is internally exploited in recursive skeleton functions like `divide_conquer` [B1, Chap. 7.3].

Parallel programming with skeletons may be seen in contrast to parallel programming using parallel library routines. For nearly any parallel supercomputer platform, there are several packages of parallel subroutine libraries available, in particular for numerical computations on large arrays. Also for library routines, reusability is an important purpose. Nevertheless, the usage of library routines is more restrictive than for skeletons, because they exploit parallelism only at the bottom level of the program's hierarchical structure, that is, they are not compositional, and their computational structure is not transparent for the programmer. Instead, they are internally highly optimized for performance. Nevertheless, generic library routines and preprocessor macros can simulate the effect of skeletons at a limited degree in Fork, as we will show in the following sections.

In the following, we focus on data parallelism as an example domain. The implementation and application of the other skeleton functions listed in Table 4.5 can be found in *Practical PRAM Programming* [B1, Chap. 7].

4.3.1 Data Parallelism

Data parallelism denotes the concurrent application of the same operation f on all elements of a homogeneous regular collection of input data d, which may be given in a form such as a list or an array. Alternatively, this can be regarded as a single *dataparallel operator* F, applied to the collection d of input data as a whole. The domain of elementwise application is determined by the input data domains, but may additionally be restricted by the programmer.

The simplest form of data parallelism arises where the element computations are mutually independent from each other. Hence the order of their execution does not matter. Let us assume for the first that the result x of the dataparallel computation is not being written to a variable that occurs on its right-hand side. Otherwise we have some kind of fixpoint computation, whose parallel execution requires special scheduling or buffering techniques and will be considered later.

In an imperative environment like Fortran90/95, HPF, or Fork, such collections of elemental data are typically realized as arrays, and the elementwise application of f is done by dataparallel loops:

```
sh data x[N], d[N];
sh int p = #;
int i;
extern straight data f( data );
...
forall(i, 0, N, p)
   x[i] = f( d[i] );
...
```

Here, the iterations of the parallel `for` loop form N elemental computations, indexed by the private loop variable `i`. As long as f is implemented as an asynchronous function, the entire `forall` loop could run in asynchronous mode:

```
farm
  forall(i, 0, N, p)
    x[i] = f( d[i] );
```

Generalizations to higher-dimensional arrays and more operand arrays are straightforward.

Some languages like Fortran90 offer language constructs that allow to handle arrays as a whole:

```
x = f( d );       // Fortran90-like array syntax
```

In other words, the function f is *mapped* to the input data. In a functional environment, one could express this in terms of the application of a *higher-order function* map to f:

x = map f d

By convention, map denotes a skeleton that specifies (and implements) parallelism by elementwise application of an arbitrary (unary) argument function f to input data d in a generic way. The result does not depend on the order of processing the elements of d. Hence, applications of f to different elements of d may be computed independently on different processors. Nevertheless, termination of map must guarantee the completion of all elemental f computations, so that the result x can be used by subsequent computations.

Although the C-based Fork has no true higher-order functions, it allows to simulate, at a quite limited degree, the usage of higher-order functions by passing a pointer to the function f as an additional parameter. Generic data types are enabled by using `void *` pointers for the operand and the result array, and by passing the element size (in terms of PRAM memory words) as a parameter. A call to a generic `map` implementation in Fork may thus look like

```
x = map( f, d );
```

Hence, the generic `map` function defined below can be regarded as an implementation of the map skeleton.

```
/** generic map routine for unary functions:
 */
sync void map( straight sh void (*f)(void *, void *),
               sh void **x, sh void **d,
               sh int N, sh int elsize )
{ int i;
  forall( i, 0, N, # )
     f( x+i*elsize, d+i*elsize );
}
```

This implementation hides some details such as the consecutive renumbering[17] of the processors, the order in which the elemental computations are performed, or the decision as to which elemental computations are assigned to which processor.

The worst-case time complexity of a call to `map` with problem size n executed by a group of p processors is

$$t_{\texttt{map}\ f}(n,p) = O\left(\left\lceil\frac{n}{p}\right\rceil \cdot t_f\right) \tag{4.1}$$

For a dataparallel operation with two operand arrays, we provide a variant of `map`:

[17]In order to make `map` work in any calling context, the renumbering must be applied.

```
/** generic map routine for binary functions:
 */
sync void map2( sh straight (*f)(void *, void *, void *),
                sh void **x, sh void **d1, sh void **d2,
                sh int n, sh int elsize )
{ int i;
  forall( i, 0, n, # )
     f( x+i*elsize, d1+i*elsize, d2+i*elsize );
}
```

The worst-case time complexity of `map2` is the same as for `map`:

$$t_{\texttt{map2}\ f}(n,p) = O\left(\left\lceil \frac{n}{p} \right\rceil \cdot t_f\right) \tag{4.2}$$

Here is an example call for floatingpoint vector addition:

```
straight void fadd( void *c, void *a, void *b )
 { *(float *)c = *(float *)a + *(float *)b; }

sync float *fvadd( sh float *x, sh float *y, sh int n )
{
 sh float *z;
 int i;
 seq z = (float *)shmalloc( n * sizeof(float) );
 map2( fadd, (void**)z, (void**)x,(void**)y, n,sizeof(float));
 return z;
}
```

A second variant of implementing skeletons in Fork uses the C preprocessor. Macros are generic by default; hence taking care of the element sizes is not required here. Our `map` skeleton, rewritten as a macro, then looks as follows:

```
#define MAP( f, x, d, n ) \
{ \
  int p = #; \
  int i; \
  farm \
    forall(i, 0, n, p ) \
      f( x[i], d[i] ); \
}
```

4.3.2 Nestable Skeleton Functions

A fundamental alternative to such function or macro implementations of skeletons is to have primitives like `map` available as built-in declarative language constructs and thus move the burden of exploiting implicit parallelism from the programmer to the compiler. This is done

```
sync void Map2(
  sync sh void (*f)(sh void*, sh void*, sh void*, sh int),
  sh void **x, sh void **d, sh void **d2, sh int N, sh int m,
  sh int mx, sh int md, sh int md2, sh int elsize )
{
 sh int p = #;
 if (p < N)
   fork( p; @=$; ) {
     sh int t;
     for (t=@; t<N; t+=p)
       f( x+t*mx*elsize, d+t*md*elsize, d2+t*md2*elsize, m);
   }
 else
   fork( N; @=$%N; )
     f( x+@*mx*elsize, d+@*md*elsize, d2+@*md2*elsize, m);
}
```

FIGURE 4.29: `Map2`, the nestable variant of `map2` for binary element functions.

by skeleton languages like P3L, SkIL, SCL, or HSM. There, skeletons are an essential part of the language itself; there are strict rules about their interoperability with the "classical" language elements. In some skeleton languages, skeletons may be hierarchically composed. The compiler obtains complete information about the entire structure of the parallel computation hierarchy. Hence, it is able to derive cost estimations and has built-in expert knowledge to decide about transformations or mapping and scheduling strategies. The extension of Fork by a separate skeleton language layer, for instance, as a combination with HSM as proposed by Marr [Mar97], is an issue of ongoing research and development.

In a purely imperative framework like Fork, the structural composition of skeletons must, at the same time, also serve as a detailed description of the imperative structure of the (parallel) computation. At a first glance this seems to be a prohibitive restriction. Nevertheless we will see that the existing features of Fork do allow for an implementation of nestable skeleton functions that exploits nested parallelism, without requiring any language extension.

For the `map2` skeleton function, for instance, a nestable variant `Map2` is given in Figure 4.29. The main difference to `map2` is that the parameter function `f` is synchronous, takes shared parameters, and needs an additional parameter `m` that holds the extent of the inner computation, called the *inner problem size*, while the *outer problem size* is the extent of the map computation itself, `n`. Also, `Map2` needs a stride parameter for each parameter: `mx` for `x` (the result), `md` for `d` (the first operand), and `md2` for `d2` (the second operand). If the result or an operand is a vector, the corresponding stride parameter is 1. If it is a scalar, its stride parameter is 0. If it is a matrix (see later), its stride parameter is the row or column length (depending on the storage scheme of the matrix). We will see an application of `Map2` for parallel matrix–vector multiplication in Section 4.3.4.

The worst-case time complexity of a call to `Map` or `Map2` with outer problem size n and inner problem size m depends on the number of processors executing it:

$$t_{\texttt{Map}\ f}(n,m,p) = \begin{cases} O\left(t_f\left(m, \left\lfloor \frac{p}{n} \right\rfloor\right)\right), & p > n \\ O\left(\frac{n}{p} \cdot t_f(m,1)\right), & p \leq n \end{cases} \tag{4.3}$$

where $t_f(m,p')$ denotes the worst-case time complexity of f when executed with p' processors.

4.3.3 Reductions

A ***reduction*** denotes a computation that accumulatively applies a binary operation f to all elements of a collection of data items, and returns the accumulated value. A typical representative is a global sum computation of all elements in an array, where f is the usual addition. In sequential, a reduction usually looks like

```
int i;
data s, d[n];
...
s = d[0];
for (i=1; i<n; i++)
   s = f( s, d[i] );
```

The structure of this computation is shown in Figure 4.30*a*.

A generic sequential reduction function can hence be written as follows:

```
#include <string.h>     // needs prototype for memcpy()

void seq_reduce( straight void (*f)(void *, void *, void *),
                 void *s, void **d, int n, int elsize )
{
 int t;
 if (n<=0) return;
 memcpy( s, d, elsize );    // initialize s by d[0]
 for (t=1; t<n; t++) {
    f( s, s, d+t*elsize );
 }
}
```

An efficient parallelization of a reduction is only possible if f is *associative*. Then, instead of scanning d sequentially element by element, partial sums s_1, s_2 of disjoint parts of d can be recursively computed concurrently, and the partial sums are then combined by f. Hence, we obtain a treelike computation, as shown in Figure 4.30*b*. It seems most appropriate to split a given data collection into two parts d_1 and d_2 of (approximately) equal size, as this will result in a balanced computation tree T of logarithmic depth. All computations on the same level of T are independent of each other and may thus be processed in parallel; they only depend on the results computed in the next deeper level.

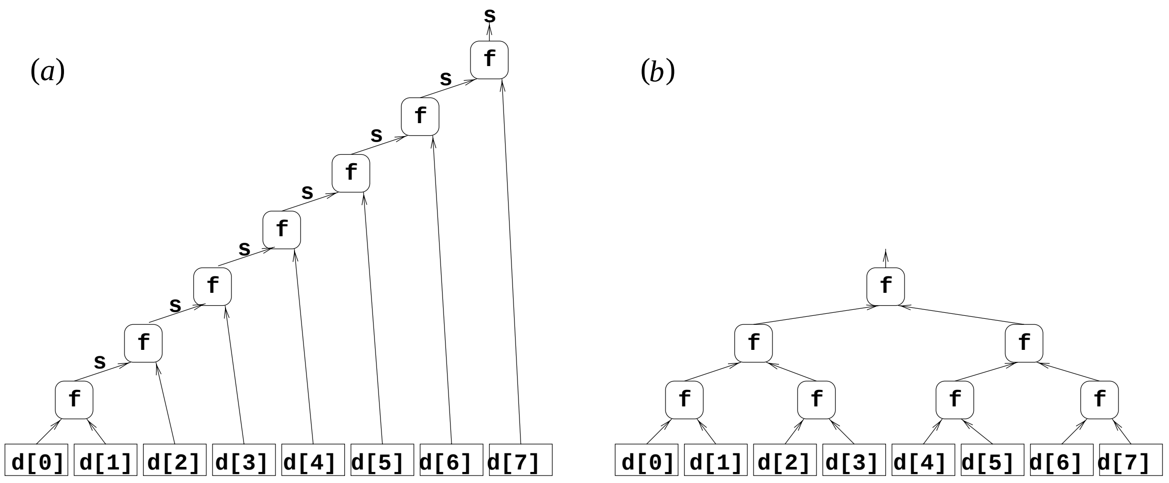


FIGURE 4.30: The computation structure of a sequential (*a*) and a parallel (*b*) reduction.

We may formulate this directly as an imperative, recursive function in Fork [B1, Chap. 7.3]. However, a recursive implementation results in many function calls and many parameters being passed. Instead, we give an iterative implementation `preduce` in Figure 4.31. The p processors process the tree level by level, starting at level $l = 1$ immediately above the leaves. In each level $l \geq 1$, the intermediate results of the previous level are stored in the operand array `d` (in-place computation) in $2k$ elements, with $k = 2^{\lceil \log n \rceil - l}$, stored in every tth element of `d`, with $t = 2^l$. These $2k$ elements are pairwise added to form k intermediate results at level l, which are stored again in every $(2t)$th element of `d`. Hence, the number of elements to be added is halved by processing a level. After level $\lceil \log n \rceil$ has been processed, `d[0]` contains the global sum.

Usually, there are fewer processors available than there are data elements to be reduced. We refine the algorithm as follows. In a first phase, each processor sequentially computes

```
/** In-place f-reduction for an array d of n<=2p items
 *  of size elsize, to be executed by p processors.
 */
sync void preduce( sync sh void (*f)(void *, void *, void *),
                   sh void **d, sh int n, sh int elsize )
{
 sh int t;
 $ = $$;
 // iterative computation upwards the tree:
 for (t=1; t<n; t = t<<1 )   // sequ. loop over tree levels
   if (2*$*t+t < n)
      f( d + (2*$*t)*elsize,
         d + (2*$*t)*elsize,  d + (2*$*t+t)*elsize );
}
```

FIGURE 4.31: A fully parallel reduction in Fork.

```
#include <string.h>     // needs prototype for memcpy()

sync void reduce( straight void (*f)(void*, void*, void*),
                  sh void *s,  sh void **d,  sh int n,
                  sh int elsize )
{
 sh int p = #;
 sh void **temp = (void **)shalloc( 2*p*elsize );
 int ss, myslice;
 $ = $$;

 if (2*p < n) {
   // partition the data vector d into p slices:
   farm {
     ss = (int)((float)(n+p-1) / (float)(p));  //==ceil(n/p)
     if ($*ss >= n)          myslice = 0;
     else if (($+1)*ss > n)  myslice = n - $*ss;
     else                    myslice = ss;
   }
   // concurrently do sequential reduction for each slice:
   farm
     seq_reduce( f, temp+$*elsize, d+$*ss*elsize, myslice,
                 elsize );
   n = p;   // extent of temp for preduce
 }
 else // copy data in parallel to temp array:
   farm {
     if ($ < n)
       memcpy( temp+$*elsize, d+$*elsize, elsize );
     if (p+$ < n)
       memcpy( temp+(p+$)*elsize, d+(p+$)*elsize, elsize );
   }
 preduce( f, temp, n, elsize );
 seq
   memcpy( s, temp, elsize );    // write result to *s
 shallfree();  // release temp
}
```

FIGURE 4.32: Implementation of the `reduce` function for generic parallel reductions in Fork.

a local reduction on a data slice of approximate size n/p, where n is the size of the data collection being reduced, and p is the number of available processors. This results in an intermediate collection of p partial result values. In a second phase, these p partial results are combined by `preduce()` as above in $\lceil \log p \rceil$ time steps. Expressed in terms of skeletons, the first phase of this algorithm consists of a map over p sequential reduction computations, and the second phase is a parallel reduce computation on the intermediate results. The resulting implementation, `reduce`, is shown in Figure 4.32.

Note that the details of the implementation, like temporary storage allocation, recursive or iterative computation, index computations, handling of special cases, or performance tuning are encapsulated and hidden from the programmer.

As an example, the following function `fdot` computes the *dot product* of two floating-point vectors of size n:

```
straight void fadd( void *c, void *a, void *b )
 { *(float *)c = *(float *)a + *(float *)b; }

straight void fmul( void *c, void *a, void *b )
 { *(float *)c = *(float *)a * *(float *)b; }

sync void fdot( sh float*s, sh float*x, sh float*y, sh int n)
{
 sh float *tmp = (float *)shalloc( n * sizeof(float) );
 map2( fmul, (void **) tmp, (void **)x, (void **)y,
       n, sizeof(float) );
 reduce( fadd, s, (void **) tmp, n, sizeof(float) );
 shallfree();
}
```

As time complexity of `fdot` we obtain

$$t_{\texttt{fdot}}(n,p) = t_{\texttt{map2 fmul}}(n,p) + t_{\texttt{reduce fadd}}(n,p) + O(1) = O\left(\frac{n}{p} + \log p\right) \tag{4.4}$$

The generality must be paid with a certain overhead due to the out-of-line definition of the function f and the generic pointer usage. This is why several programming languages offer special reductions for common cases of f as built-in language primitives or library routines. An example is the intrinsic sum function `sum` for floatingpoint arrays in Fortran90, or the integer sum function `syncadd` in Fork.

Also, `reduce` may be implemented as a preprocessor macro as well.

As in the case of `map`, a nestable variant `Reduce` of the `reduce` skeleton function can be defined, see [B1, Ex. 7.17]. The worst-case time complexity of `Reduce`, with outer problem size n and inner problem size m, depends on the number of processors executing it:

$$t_{\texttt{Reduce}f}(n,m,p) = \begin{cases} O\left(\log n \cdot t_f\left(m, \left\lfloor \frac{p}{n} \right\rfloor\right)\right), & p > n \\ O\left(\left(\frac{n}{p} + \log p\right) \cdot t_f(m,1)\right), & p \le n \end{cases} \tag{4.5}$$

In the same way as for the generic reductions we can define nestable and non-nestable skeleton functions for generic parallel prefix computations. These and the other skeleton functions listed in Table 4.5 are given in *Practical PRAM Programming* [B1, Chap. 7].

4.3.4 Composing Skeleton Functions

Now we show how skeleton functions can be composed to structured parallel programs, such that nested parallelism is exploited if enough processors are available for executing the program. As an example, we consider matrix–vector multiplication.

The matrix–vector product of a matrix $A \in \mathbb{R}^{n,m}$ with a vector $\vec{b} \in \mathbb{R}^m$ is defined as a vector $\vec{c} \in \mathbb{R}^n$ with

$$c_i = \sum_{j=1}^{m} A_{i,j} b_j, \qquad i = 1, \ldots, n$$

Each component of the result vector $\vec{c}$ is determined by a dot product of the corresponding row vector $\vec{a}_i$ of A with the corresponding component of $\vec{b}$.

Using our `fdot` routine defined in Section 4.3.3, we can hence simply write

```
sync void fmatvec( sh float *c, sh Matrix A, sh float *b,
                   sh int n, sh int m )
{
 Map2((sync void(*)(sh void*, sh void*, sh void*, sh int))fdot,
      (void **)c, (void **)(A->data), (void *)x, n, m,
      1, m, 0, sizeof(float) );
}
```

where we exploit the fact that the matrix elements of A are stored row major in `A->data`, i.e. as a linear sequence of the n row vectors of length m each. The access stride for the vector `c` is set to 1, the stride for the matrix data array is equal to the row length m, and the stride for the vector `b` is set to zero, as the same vector b is to be dot-multiplied with each row vector of A. The nestable `Map2` skeleton function (see Figure 4.29) passes the inner problem size m to the `fdot` routine. Note that the deviation using the `fdot` definition is necessary, as an inline specification of a sequence of functions is syntactically not possible in Fork.

Hence, we have created a composition of three skeletons on two levels: `Map2` at the top level, and (see the definition of `fdot` above) the sequence of `map2` and `reduce` at the inner level. By inserting formulas (4.3) and (4.4), the worst-case time complexity of this composition is

$$\begin{aligned} t_{\texttt{fmatvec}}(n,m,p) &= t_{\texttt{Map2 fdot}\ m}(n,p) \\ &= \begin{cases} O\left(t_{\texttt{fdot}}\left(m, \left\lfloor \frac{p}{n} \right\rfloor\right)\right), & p > n \\ O\left(\frac{n}{p} \cdot t_{\texttt{fdot}}(m,1)\right), & p \le n \end{cases} \\ &= \begin{cases} O\left(\frac{nm}{p} + \log \frac{p}{n}\right), & p > n \\ O\left(\frac{nm}{p}\right), & p \le n \end{cases} \end{aligned} \tag{4.6}$$

Alternatively, the matrix–vector product can be regarded as a plus–reduction of m vectors of length n each, which produces a vector-valued result. In this case, m is the outer problem size and n the inner problem size. This can be implemented using the nestable variant of the `reduce` skeleton, `Reduce` to sum up vectors, using a parallel function `fvadd` for adding two floatingpoint vectors as a parameter that, in turn, can be implemented using the `map2` skeleton function. Consequently, this yields a composition of two skeletons with two hierarchy levels: `Reduce` at the top level, and `map2` at the inner level. If less than m processors are available, all are dedicated to the top level (i.e., the reduction), and the inner level is executed sequentially. Otherwise, there are about p/m processors available for each instance of the `map2` computation. Hence, the worst-case time complexity of this variant is as follows:

$$\begin{aligned} t_{\texttt{fmatvec'}}(m,n,p) &= t_{\texttt{Reduce fvadd } n}(m,p) \\ &= \begin{cases} O\left(\left(\frac{m}{p} + \log m\right) \cdot t_{\texttt{fvadd}}(n,1)\right), & p < m \\ O\left(\log m \cdot t_{\texttt{fvadd}}\left(n, \left\lceil \frac{p}{m} \right\rceil\right)\right), & p \geq m \end{cases} \\ &= \begin{cases} O\left(\frac{nm}{p} + n \log p\right), & p < m \\ O\left(\log m \cdot \frac{nm}{p}\right), & p \geq m \end{cases} \end{aligned} \tag{4.7}$$

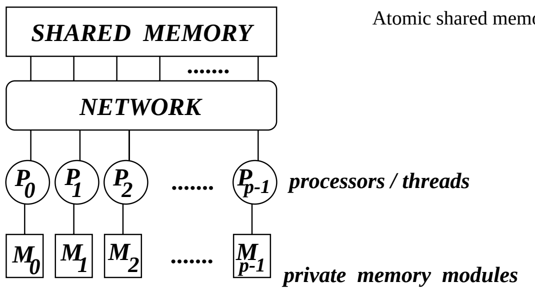
Comparing this to equation (4.7), it appears that for small numbers of processors the asymptotic time complexities are more or less the same, while for large p the first variant outperforms the second, because the `map` / `Map` algorithm is cost-optimal while the `reduce` / `Reduce` algorithm is not [B1, Chap. 2]. the latter produces more idle cycles (see Figure 4.30*b*). Hence it seems to be advisable to exploit as much parallelism as possible along the `map` axis (with extent n) and only the remaining parallelism along the `reduce` axis (with extent m), and thus we favor the first variant.

4.4 ForkLight **Language Design**

Parallel computer manufacturers and research groups recently devised several types of massively parallel (or at least scalable) machines simulating or emulating a shared memory. Most of these machines have nonuniform memory access time (NUMA). Some of these still require program tuning for locality in order to perform efficiently, e.g. Stanford DASH [LLG$^+$92], while others use multithreading to hide the memory access latency and high-bandwidth memory networks, and thus become more or less independent of locality issues, such as the Tera MTA [ACC$^+$90].

So far there is only one synchronous, massively parallel shared memory architecture that offers uniform memory access time (UMA), the SB-PRAM [ADK$^+$93]. No massively parallel MIMD machine that is commercially available today is UMA or synchronous in the PRAM sense. Rather, the common features are the following:

- the user sees a large amount of threads (due to scalable architecture and multithreading)
- the user sees a monolithic shared memory (due to a hidden network)
- there is no common clock


SHARED MEMORY
....
NETWORK
P_0 P_1 P_2 .... P_{p-1}
processors / threads
M_0 M_1 M_2 .... M_{p-1}
private memory modules

Atomic shared memory access primitives:	`SMwrite`	(write a value into a shared memory cell)
	`SMread`	(read a value from a shared memory cell)
	`atomic_add`	(add an integer to a cell)
	`fetch_add`	(add an integer to a cell and return its previous value)
furthermore:	`shmalloc`	(shared memory allocation)
	`beginparallelsection`	(spawn p-1 threads)
	`endparallelsection`	(kill spawned threads)

FIGURE 4.33: Asynchronous PRAM model with shared memory access operations. Assuming that the processors' native load/store operations are used for accessing private memory, the seven operations listed on the right hand side are sufficient to handle shared memory parallelism as generated by the ForkLight compiler. Other auxiliary functions for thread handling, like inspecting the thread ID or the total number of threads, could also be implemented by the runtime library of the compiler.

- the memory access time is nonuniform, but more or less independent of locality (due to multithreading)
- program execution is asynchronous (due to the previous two items, and because of system features like virtual memory, caching, and I/O)
- there is efficient hardware support for atomic *fetch&op* instructions

A typical representative of this class of parallel machines is the Tera MTA [ACC$^+$90].

In order to abstract from particular features and to enhance portability of parallel algorithms and programs, one often uses a *programming model* that describes the most important hardware properties. Suitable parameterization allows for straightforward estimates of execution times; such estimations are the more accurate, the more the particular parallel hardware fits the model used. Typically, also a standard programming interface is defined for such models, e.g. MPI for the message passing model, HPF for dataparallel programming of distributed memory machines, BSPlib for the BSP model, or Fork for the PRAM model.

In this case, the programming model is the Asynchronous PRAM introduced in the parallel theory community in 1989 [Gib89, CZ89, CZ95]. An ***Asynchronous PRAM*** (see Figure 4.33) is a MIMD parallel computer with a sequentially consistent shared memory. Each processor runs with its own private clock. No assumptions are made on uniformity of shared memory access times. Thus, much more than for a true PRAM, the programmer must explicitly take care of avoiding race conditions (nondeterminism) when accessing shared memory locations or shared resources (screen, shared files) concurrently. We add to this model some atomic *fetch&op* instructions like `fetch_add` and atomic update instructions like `atomic_incr`, which are required for basic synchronization mechanisms and typically supported by the parallel machines in consideration [18]. Note however that this is not an inadmissible modification

[18]Efficient support for atomic operations is provided, for instance, in Tera MTA, HP/Convex Exemplar

of the original Asynchronous PRAM model, since software realizations for these primitives are also possible (but at the expense of significant overhead [ZYC96]). In short, this programming model is closely related to the popular PRAM and BSP models but offers, in our view, a good compromise, as it is closer to real parallel machines than PRAMs and easier to program than BSP.

Programming languages especially designed for "true" PRAMs, such as Fork, cannot directly be used for Asynchronous PRAMs, as their efficient implementation relies on unit memory access time and instruction-level synchronous program execution.

In this section we describe the parallel programming language ForkLight [C16] for the Asynchronous PRAM model, which retains a considerable part of the programming comfort known from Fork while dropping the requirement for strictly synchronous execution on the expression operator level, which would inevitably lead to poor performance on any existing asynchronous machine. Rather, synchronicity is relaxed to the basic block level. We work out its similarities and main differences to Fork. The compilation of ForkLight is discussed in Section 5.3.

4.4.1 SPMD Execution

As Fork, ForkLight follows the SPMD model of parallel execution, that is, all p available processors (or threads) are running just from the beginning of `main` execution. Thus, ForkLight does not need a **spawn** resp. **forall** statement to spawn new parallel threads from a current one. Coordination is provided, for instance, by composing the processors into groups. The processors of a group can allocate grouplocal shared variables and objects. They work *control-synchronously*, that is, all branches of control flow are synchronized. In order to adapt to finer levels of nested parallelism, a group can be (recursively) subdivided into subgroups. In this way, ForkLight supports statically and dynamically nested parallelism in the same way as Fork. Control-synchronous execution can locally be relaxed towards totally asynchronous computation where this is desired by the programmer, such as for efficiency reasons. Also, a clever compiler will aim at removing as many synchronization points as possible without courting the risk of race conditions.

4.4.2 Variables and Expressions

As in Fork, variables are classified either as *private* or as *shared*; the latter are again to be declared with the storage class qualifier `sh`.

The *total number of started processors* is accessible through the constant shared variable

```
__P__
```

The *physical processor ID* is accessible through the function

```
_PhysThreadId()
```

[SMS96], and SB-PRAM.

4.4.3 Control-Synchronous and Asynchronous Program Regions

ForkLight offers two different program execution modes that are statically associated with source code regions: control-synchronous mode in control-synchronous regions, and asynchronous mode in asynchronous regions.

In *control-synchronous mode*, which is the equivalent of the synchronous mode of Fork, ForkLight maintains the control-synchronicity invariant:

> ***Control-synchronicity***
>
> *All processors belonging to the same (active) group work on the same activity block.*

In ForkLight, the implicit synchronization points that define the activity blocks are more or less set in a way that associates activity blocks with *extended basic blocks*. A basic block can be extended by an interspersed asynchronous region, such as the body of an `async` or a `seq` statement (see later), for instance.

Subgroup creation and implicit barriers occur only in control-synchronous mode.

In *asynchronous mode*, control-synchronicity is not enforced. The group structure is read-only; shared variables and automatic shared heap objects cannot be allocated. There are no implicit synchronization points. Synchronization of the current group can, though, be explicitly enforced by a call to the `barrier()` function, or, to optically emphasize it in the program code, by a barrier statement like

```
===========
```

i.e., a sequence of at least three `=`'s.

Initially, one processor on which the program has been started by the user executes the startup code and initializes the shared memory. Then it spawns the other processors requested by the user program. All these processors start execution of the program in asynchronous mode by calling `main()`.

Functions are classified as either control-synchronous (to be declared with type qualifier `csync`) or asynchronous (this is the default). `main()` is asynchronous by default.

A control-synchronous function is a control-synchronous region, except for (blocks of) statements explicitly marked as asynchronous by `async` or as sequential by `seq`.

An asynchronous function is an asynchronous region, except for statements explicitly marked as control-synchronous by `start` or `join()`.

The `async` statement is the equivalent of the `farm` statement in Fork:

```
async  <stmt>
```

causes the processors to execute `<stmt>` in asynchronous mode. In other words, the entire `<stmt>` (which may contain loops, conditions, or calls to asynchronous functions) is considered to be part of the "basic" (w.r.t. control-synchronicity) block containing this `async`. There is no implicit barrier at the end of `<stmt>`. If the programmer desires one, (s)he may use an explicit barrier (see above).

As in Fork, the `seq` statement

```
seq  <stmt>
```

causes `<stmt>` to be executed by exactly one processor of the current group; the others skip `<stmt>`. But, in contrast to the `seq` statement in Fork, there is no implicit barrier at the end of `<stmt>`.

Asynchronous functions and `async` blocks are executed in asynchronous mode, except for (blocks of) statements starting with the `start` statement

```
start <stmt>
```

The `start` statement, only permitted in asynchronous mode, switches to control-synchronous mode for its body `<stmt>`. It causes all available processors to barrier-synchronize, to form one large group, and execute `<stmt>` simultaneously and in control-synchronous mode, with unique processor IDs `$` numbered consecutively from 0 to `__P__`-1.

As in Fork, there is a `join` statement (see Section 4.2.9) in ForkLight with the same semantics, except that only control synchronicity holds for the bus group instead of strict synchronicity.

In order to maintain the static classification of code into control-synchronous and asynchronous regions, within an asynchronous region, only `async` functions can be called. In the other way, calling an `async` function from a control-synchronous region is always possible and results in an implicit entering of the asynchronous mode.

Shared local variables can only be declared / allocated within control-synchronous regions. In particular, asynchronous functions must not allocate shared local variables. In contrast to Fork there are no shared formal parameters in ForkLight.

Control-synchronous functions contain an implicit groupwide barrier synchronization point at entry and another one after return, in order to reinstall control-synchronous execution also when the function is left via different `return` statements.

4.4.4 Groups and Control Synchronicity in ForkLight

ForkLight programs are executed by *groups* of processors, as introduced in Section 4.1.

As in Fork, subgroups of a group can be distinguished by their *group ID*. The group ID of the leaf group a processor is member of can be accessed through the shared variable `@`. `join` and `start` initialize `@` to 0. The group-relative processor ID `$` and the group rank `$$` are defined as in Fork. The values of `$`, `$$`, and `@` are automatically saved when deactivating the current group, and restored when reactivating it. Thus, these values remain constant within the activity region of a group.

Also, all processors within the same group have access to a common shared address subspace. Thus, newly allocated shared objects exist once for each group allocating them.

As in Fork, a processor can inspect the number of processors belonging to its current group using the routine

```
int groupsize();
```

or by accessing the read-only variable `#`.

Groups and control flow

Without special handling control flow could diverge for different processors at conditional branches such as `if` statements, `switch` statements, and loops. Only in special cases it can be statically determined that all processors are going to take the same branch of control flow. Otherwise, control-synchronicity could be lost. In order to prevent the programmer from errors based on this scenario, ForkLight guarantees control-synchronicity by suitable group splitting. Nevertheless, the programmer may know in some situations that such a splitting is unnecessary. For these cases, (s)he can specify this explicitly.

We consider an expression to be ***stable*** if it is guaranteed to evaluate to the same value on each processor of the group for all possible program executions, and *unstable* otherwise.

An expression containing private variables (e.g., `$`) is generally assumed to be unstable. But even an expression e containing only shared variables may be also unstable: Since e is evaluated asynchronously by the processors, it may happen that a shared variable occurring in e is modified (maybe as a side effect in e, or by a processor outside the current group) such that some processors of the group (the "faster" ones) use the old value of that variable while others use the newer one, which may yield different values of e for different processors of the same group. The only shared variable which can always be assumed to be stable is `@`, because it is read-only, thus expressions that only involve `@` and constants are stable.

Technically, the compiler defines a conservative, statically computable subset of the *stable expressions* as follows:

(1) `@` is a stable expression.
(2) A constant is a stable expression. (This includes shared constant pointers, e.g. arrays.)
(3) The pseudocast `stable(`e`)` is stable for any expression e (see below).
(4) If expressions e_1, e_2 and e_3 are stable, then also the expressions $e_1 \oplus e_2$ for $\oplus \in \{+, -, *, /, \%, \&, |, \&\&, ||\}$, $\ominus e_1$ for $\ominus \in \{-, \hat{}, !\}$, and $e_1 ? e_2 : e_3$ are stable.
(5) All other expressions are regarded as unstable.

Conditional branches with a stable condition expression do not affect control-synchronicity. Otherwise control-synchronicity can generally be lost; this could result in unforeseen race conditions or deadlock. For this reason, unstable branches in control-synchronous mode lead to a splitting of the current group into subgroups—one for each possible branch target. Control synchronicity is then only maintained within each subgroup. Where control flow reunifies again, the subgroups cease to exist, and the previous group is restored. There is no implicit barrier at this program point. [19]

For a two-sided `if` statement with an unstable condition, for instance, two subgroups are created. The processors that evaluate their condition to true join the first, the others join the second subgroup. The branches associated with these subgroups are executed concurrently.

For a loop with an unstable exit condition, one subgroup is created that contains the iterating processors.

Unstable branch conditions in control-synchronous mode may nevertheless be useful for performance tuning. Thus, there is the possibility of a pseudocast

[19] As a rule of thumb: Implicit barriers are, in control-synchronous mode, generated only at branches of control flow, not at reunifications.

```
stable (<expr>)
```

which causes the compiler to treat expression `<expr>` as stable. In this case the compiler assumes that the programmer knows that possible unstability of `<expr>` will not be a problem in this context, for instance because (s)he knows that all processors of the group will take the same branch. [20]

Splitting a group into subgroups can also be done explicitly, using the `fork` statement, which is only admissible in control-synchronous regions. Executing

fork (e_1; @=e_2) <stmt>

means the following: First, each processor of the group evaluates the stable expression e_1 to the number of subgroups to be created, say g. Then the current leaf group is deactivated and g subgroups $g_0, ..., g_{g-1}$ are created. The group ID of g_i is set to i. Evaluating expression e_2 (which is typically unstable), each processor determines the index i of the newly created leaf group g_i it will become member of. If the value of e_2 is outside the range $0, ..., g-1$ of subgroup IDs, the processor does not join a subgroup and skips `<stmt>`[21]. The IDs `$` of the processors are renumbered consecutively within each subgroup from 0 to the subgroup size minus one. Each subgroup gets its own shared memory subspace, thus shared variables and heap objects can be allocated locally to the subgroup. — Now, each subgroup g_i executes `<stmt>`. When a subgroup finishes execution, it ceases to exist, and its parent group is reactivated as the current leaf group. Unless the programmer writes an explicit barrier statement, the processors may immediately continue with the following code.

Groups and jumps

Regarding jumps, ForkLight behaves like Fork. For `break`, `continue`, and `return` jumps, the target group is statically known; it is an ancestor of the current leaf group in the group hierarchy tree. In this case, the compiler will provide a safe implementation even for the control-synchronous mode.

By a `goto` jump, control-synchronicity is, in principle, not lost. However, the target group may not yet have been created at the time of executing the jump. Even worse, the target group may not be known at compile time. Nevertheless, as long as source and destination of a `goto` are known to be within the activity scope of the same (leaf) group, there is no danger of deadlock. For this reason, we have renounced to forbid `goto` in control-synchronous regions, but all other cases cause a warning to be emitted.

Other statements affecting control flow

The short-circuit evaluation of the binary logical operators && and `||` and for the `?:` operator applies also to ForkLight. Hence, similar problems as described for Fork in Section 4.2.10 arise also in ForkLight. In order to avoid unintended asynchrony or deadlocks, &&, `||` and

[20]Alternatively, there is the option to declare such control constructs as asynchronous by `async`.

[21]Note that empty subgroups (with no processors) are possible; an empty subgroup's work is immediately finished, though.

`?:` with unstable left operands are illegal in control-synchronous mode. Clearly there exist obvious workarounds for such situations, as shown in Section 4.2.10.

4.4.5 Pointers and Heaps

The usage of pointers is slightly more restricted than in Fork since in ForkLight the private address subspaces are not embedded into the global shared memory but addressed independently. Thus, shared pointer variables must not point to private objects. As it is, in general, not possible to statically verify whether the pointee is private or shared, dereferencing a shared pointer containing a private address will lead to a run time error. Nevertheless it is legal to make a shared or a private pointer point to a shared object.

ForkLight supplies three kinds of heaps, as in Fork. First, there is the usual, private heap for each processor, where space can be allocated and freed by the (asynchronous) functions `malloc` and `free` known from C. Second, there is a global, permanent shared heap where space can be allocated and freed accordingly using the asynchronous functions `shmalloc` and `shfree`. And finally, ForkLight provides one automatic shared heap for each group. This kind of heap is intended to provide fast temporary storage blocks which are local to a group. Consequently, the life range of objects allocated on the automatic shared heap by the control-synchronous `shalloc` function is limited to the life range of the group by which that `shalloc` was executed. Thus, such objects are freed automatically when the group allocating them is released.

Pointers to functions are also supported in ForkLight. Dereferencing a pointer to a control-synchronous function is only legal in control-synchronous mode if it is stable.

4.4.6 Standard Atomic Operations

Atomic *fetch&op* operations, also known as *multiprefix* computations when applied in a fully-synchronous context with priority resolution of concurrent write accesses to the same memory location, have been integrated as standard functions called `fetch_add`, `fetch_max`, `fetch_and` and `fetch_or`, in order to give the programmer direct access to these powerful operators. They can be used in control-synchronous as well as in asynchronous mode. Note that the order of execution for concurrent execution of several, say, `fetch_add` operations to the same shared memory location is not determined in ForkLight.

For instance, determining the size p of the current group and consecutive renumbering 0,1,...,$p-1$ of the group-relative processor ID `$` could also be done by

```
csync void foo( void )
{
  int myrank;
  sh int p = 0;
  ============= //guarantees p is initialized
  myrank = fetch_add( &p, 1 );
  ============= //guarantees p is groupsize
  ...
}
```

where the function-local integer variable p is shared by all processors of the current group.

The implementation of *fetch&add* is assumed to work atomically. This is very useful to access semaphores in asynchronous mode, such as simple locks that sequentialize access to some shared resource where necessary. Like Fork, ForkLight offers several types of locks in its standard library: simple locks, fair locks, and reader–writer locks.

There are further atomic memory operations atomic_*op* available as void routines. They may be used for integer global sum, bitwise OR, bitwise AND, maximum, and minimum computations.

4.4.7 Discussion: Support of Strict Synchronicity in ForkLight?

There are some cases where strict synchronicity would allow more elegant expression of parallel operations. For instance, a parallel pointer doubling operation would, in synchronous mode of Fork, just look like

```
a[$] = a[a[$]];
```

under the assumption that the shared integer array a has as many elements as there are processors available. In order to guarantee correct execution in ForkLight, one has to introduce temporary private variables and a barrier statement:

```
int temp = a[a[$]];
==============
a[$] = temp;
```

Nevertheless, we claim that this is not a serious restriction for the programmer, because (s)he must use the second variant also in Fork if not enough processors are available to perform the operation in one step (i.e., if the number N of array elements exceeds the number of processors):

```
int i, *temp, prsize;
prsize = (int)ceil(N/groupsize()));
temp = (int *)malloc(prsize);
for (i=0; i<prsize && $*prsize+i<N, i++)
   temp[i] = a[a[$*prsize+i]];
============ // only required in ForkLight
for (i=0; i<prsize && $*prsize+i<N, i++)
   a[$*prsize+i] = temp[i];
free(temp);
```

A more elegant interface for the programmer would be to allow sections of strict synchronous mode in ForkLight. This could be achieved by an extension of the current ForkLight design by a statement

```
strict <stmt>
```

that causes fully synchronous execution for a statement, as in

```
/* Parallel Mergesort in ForkLight.    C.W. Kessler 8/98
 * Sorts N elements using p processors.
 * Assumes that all elements are different;
 * otherwise the result will be wrong.
 */
#include <ForkLight.h>   // required for ForkLight programs
#include <stdio.h>       // host stdio.h
#include <stdlib.h>      // host stdlib.h

#define THRESHOLD 1

/** print an array a of size n sequentially
 */
void print_array( int *a, int n )
{
 int i;
 printf("Array %p of size %d:\n", a, n);
 for (i=0; i<n; i++) printf(" %d", a[i]);
 printf("\n");
}

/** compare function used by the sequential qsort() routine
 */
int cmp( const void *a, const void *b )
{
  if      (*(int *)a < *(int *)b) return -1;
  else if (*(int *)a > *(int *)b) return 1;
       else                       return 0;
}

/** in sequential compute the rank of key within
 *  array of size n, i.e. # array-elements < key
 */
int get_rank( int key, int *array, int n )
{
 int left = 0,  right = n-1,  mid;
 if (key >= array[n-1]) return n;
 if (key == array[n-1]) return n-1;
 if (key <= array[0]) return 0;
 while (left < right-1) {   /*binary search*/
    // always maintain array[left] <= key < array[right]
    mid = (right+left)/2;
    if (key < array[mid]) right = mid;
    else                  left = mid;
 }
 if (key==array[left]) return left;
 else                 return left+1;
}
```

FIGURE 4.34: Parallel Mergesort in ForkLight (1).

```
/** merge array src1 of size n1 and src2 of size n2
 *  into one array dest of size n1+n2.
 *  Assertions: p>1, n1*n2>=1, dest array is allocated.
 */
csync void merge( int *src1, n1, *src2, n2, *dest)
{
 sh int iter, iter2;
 sh int *rank12, *rank21;  /*temp. rank arrays*/
 int i,  p = groupsize();
 rank12 = (int *)shalloc( n1 * sizeof(int) );
 rank21 = (int *)shalloc( n2 * sizeof(int) );
 seq iter = 0;
 seq iter2 = 0;
 =================
 async
   /* self-scheduling par. loop over rank computations: */
   for (i=fetch_add(&iter,1); i<n1; i=fetch_add(&iter,1))
     rank12[i] = get_rank( src1[i], src2, n2 );
 =================
 async
   for (i=fetch_add(&iter2,1); i<n2; i=fetch_add(&iter2,1))
     rank21[i] = get_rank( src2[i], src1, n1 );
 =================
 /* copy elements to dest using the rank information */
 async for (i=$$; i<n1; i+=p)  dest[i+rank12[i]] = src1[i];
 async for (i=$$; i<n2; i+=p)  dest[i+rank21[i]] = src2[i];
}

/** mergesort for an array of size n.
 *  The sorted array is to be stored in
 *  sortedarray which is assumed to be allocated.
 */
csync void mergesort( int *array, n, *sortedarray )
{
 int i,  p = groupsize();
 if (stable(p==1)) {
     qsort( array, n, sizeof(int), cmp );
     async for (i=0; i<n; i++)  sortedarray[i] = array[i];
 }
 else
 if (stable(n<=THRESHOLD)) { // parallelism doesn't pay off
     seq qsort( array, n, sizeof(int), cmp );
     ================
     async for (i=$$; i<n; i+=p)  sortedarray[i] = array[i];
 }
 else {
    sh int *temp = (int *)shalloc( n * sizeof(int) );
    fork (2; @=$$%2)
       mergesort( array + @*(n/2),
                  (1-@)*(n/2) + @*(n-n/2), temp + @*(n/2));
    //============= implicit barrier at merge() call:
    merge( temp, n/2, temp+n/2, n-n/2, sortedarray );
 }
}
```

FIGURE 4.35: Parallel Mergesort in ForkLight (2).

```
void main( void )
{
 start {
   sh int *a, *b;
   int j,  N = 100;
   a = (int *) shalloc( N * sizeof(int) );
   b = (int *) shalloc( N * sizeof(int) );
   ================== // arrays allocated
   async for (j=$; j<N; j+=__P__)  a[j] = N - j;
   ================== // array a initialized
   seq print_array( a, N );
   ================== // array a printed
   mergesort( a, N, b );
   ================== // mergesort completed
   seq print_array( b, N );
 }
}
```

FIGURE 4.36: Parallel Mergesort in ForkLight (3).

```
.....
strict {
   a[$] = a[a[$]];
}
.....
```

A drawback of `strict` is that the compiler must perform data dependence analysis and generate code that protects the correct order of accesses to the same shared memory location by locks or barriers, as described in Chapter 5. Worst-case assumptions may even lead to complete sequentialization. `strict` would be admissible only in control-synchronous regions. Functions called from `<stmt>` would be executed in control-synchronous rather than fully synchronous mode.

4.4.8 Example: Parallel Mergesort

The parallel mergesort algorithm sorts an array of n items (here: integers) with p processors by splitting the array into two halves, sorting these recursively in parallel by $p/2$ processors each, and then reusing the same processors for merging the two sorted subarrays in parallel into one sorted array. The recursion stops if either the size of the array to be sorted falls below a certain threshold, or if the number of processors available for sorting becomes 1. In both cases a sequential sorting algorithm (such as the native `qsort` routine) may be applied. The run time is $O(n/p \cdot \log p \log n)$, assuming that an optimal sequential sorting algorithm is used for the sequential parts.

A ForkLight implementation of the parallel mergesort algorithm is given in Figures 4.34 and 4.35. A `main` function that combines these routines to a complete ForkLight program is given in 4.36.

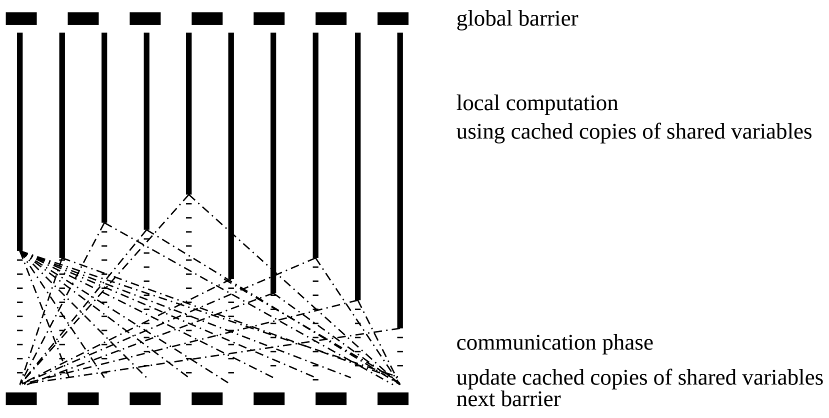

global barrier
local computation
using cached copies of shared variables
communication phase
update cached copies of shared variables
next barrier

FIGURE 4.37: A BSP superstep

4.5 NestStep **Language Design**

Now let us consider a parallel target architecture that has no shared memory at all. Using virtual shared memory techniques, a shared memory can be simulated by the compiler and the runtime system on top of a distributed memory architecture. Nevertheless, sequential memory consistency must usually be paid by a high overhead. Additionally, synchronization is even more expensive where it must be simulated by message passing. Hence, the issues of synchronization and memory consistency should be considered and solved together. This is the approach taken for the design and implementation of the parallel programming language NestStep.

As its two predecessors Fork and ForkLight, NestStep has been designed as a set of extensions to existing imperative programming languages, like Java or C. It adds language constructs and runtime support for the explicit control of parallel program execution and sharing of program objects.

4.5.1 The BSP Model

The *BSP (bulk-synchronous parallel) model*, as introduced by Valiant [Val90] and implemented e.g. by the Oxford BSPlib library [HMS$^+$98] for many parallel architectures, structures a parallel computation of p processors into a sequence of *supersteps* that are separated by global barrier synchronization points.

A superstep (see Figure 4.37) consists of (1) a phase of local computation of each processor, where only local variables (and locally held copies of remote variables) can be accessed, and (2) a communication phase that sends some data to the processors that may need them in the next superstep(s), and then waits for incoming messages and processes them. Note that the separating barrier after a superstep is not necessary if it is implicitly covered by (2). For instance, if the communication pattern in (2) is a complete exchange, a processor can proceed to the next superstep as soon as it has received a message from every other processor.

In order to simplify cost predictions, the BSP machine model is characterized by only four parameters: the number p of processors, the overhead g for a concurrent pairwise exchange of a one word message by randomly chosen pairs of processors, which roughly corresponds to the inverse network bandwidth, the minimum time L (message latency) between subsequent synchronizations, and the processor speed, s.

Additionally, some properties of the program dramatically influence the runtime, in particular the amount of communication. Let h_i denote the maximum number of words sent or received by processor i, $0 \leq i < p$. A communication pattern that bounds the maximum communication indegree and outdegree of any processor by h is called *h-relation*. For instance, the situation where each processor i sends one integer to its neighbor ($(i+1)\%p$) implies a 1-relation. Broadcasting one integer from one processor to the other $p-1$ processors is treated as a $(p-1)$-relation.

If we denote the time for the computation part of a superstep by w_i for processor i, with $w = \max_{0\leq i<p} w_i$, we obtain the following formula for the overall execution time of a superstep:

$$t = \max_{\text{procs.}P_i} w_i + \max_{\text{procs.}P_i} h_i \cdot g + L = w + h \cdot g + L$$

Note that the parameters L and g are usually functions of p that depend mainly on the topology and the routing algorithm used in the interconnection network. For instance, on a bus network, g is linear in p, because all messages are sequentialized, and thus the realization of a 1-relation takes linear time.

The BSP model, as originally defined, does not support a shared memory; rather, the processors communicate via explicit message passing. NestStep provides a software emulation of a shared memory: Shared scalar variables and objects are replicated across the processors, arrays may be distributed. In compliance to the BSP model, sequential memory consistency is relaxed to and only to superstep boundaries. Within a superstep only the local copies of shared variables are modified; the changes are committed to all remote copies at the end of the superstep. A tree-based message combining mechanism is applied for committing the changes, in order to reduce the number of messages and to avoid hot spots in the network. As a useful side effect, this enables on-the-fly computation of parallel reductions and prefix computations at practically no additional expense. For space economy, NestStep also provides distribution of arrays in a way similar to Split-C [CDG$^+$93].

In the BSP model there is no support for processor subset synchronization, i.e. for nesting of supersteps. Thus, programs can only exploit one-dimensional parallelism or must undergo a *flattening* transformation that converts nested parallelism to flat parallelism. However, automatic flattening by the compiler has only been achieved for SIMD parallelism, as e.g. in NESL [BHS$^+$94]. Instead, NestStep introduces static and dynamic nesting of supersteps, and thus directly supports nested parallelism.

Although considered unnecessary by a large fraction of the BSP community, there are several good reasons for exploiting nested parallelism:

- "Global barrier synchronization is an inflexible mechanism for structuring parallel programs" [McC96].

- For very large numbers of processors, barrier-synchronizing a subset of processors is faster than synchronizing all processors.

- If the parallel machine is organized as a hierarchical network defining processor clusters, this multi-level structure could be exploited by mapping independently operating processor subsets to different clusters.

- Most parallel programs exhibit a decreasing efficiency for a growing number of processors, because finer granularity means more communication. Thus it is better for overall performance to run several concurrently operating parallel program parts with coarser granularity simultaneously on different processor subsets, instead of each of them using the entire machine in a time-slicing manner.

- The communication phase for a subset of processors will perform faster as the network is less loaded. Note that e.g. for global exchange, network load grows quadratically with the number of processors. Moreover, independently operating communication phases of different processor subsets are likely to only partially overlap each other, thus better balancing network traffic over time.

- Immediate communication of updated copies of shared memory variables is only relevant for those processors that will use them in the following superstep. Processors working on a different branch of the subset hierarchy tree need not be bothered by participating in a global update of a value that they don't need in the near future and that may be invalidated and updated again before they will really need it.

4.5.2 NestStep **Language Design Principles**

NestStep is defined by a set of language extensions that may be added, with minor modifications, to any imperative programming language, be it procedural or object oriented. In our first approach in 1998 [C18,J7], NestStep was based on the Java language [GJS96], called NestStep-Java in the following. The implementation of the runtime system of NestStep-Java (see Section 5.4) was written in Java as well. However, we were so disappointed by the poor performance of Java, in particular the slow object serialization, that we decided for a redesign based on C in 2000, which we call NestStep-C in the following. In the course of the redesign process, we changed the concept of distributed arrays and abolished the so-called volatile shared variables, which simplified the language design and improved its compatibility with the BSP model. The Java-based version NestStep-Java is to be changed accordingly. Note that the NestStep extensions could as well be added to (a subset of) C++ or Fortran. In particular, the basis language needs not be object oriented: parallelism is not implicit by distributed objects communicating via remote method invocation, as in Java RMI [Far98], but expressed by separate language constructs.

The sequential aspect of computation is inherited from the basis language. NestStep adds some new language constructs that provide shared variables and process coordination. Some restrictions on the usage of constructs of the basis language must be made for each NestStep variant. For instance, in NestStep-C, unrestricted pointers (see Section 4.5.7) are excluded; in NestStep-Java, the usage of Java threads is strongly discouraged.

NestStep processes run, in general, on different machines that are coupled by the NestStep language extensions and runtime system to a virtual parallel computer. This is why we prefer to call them *processors* in the following, as we did also for Fork and ForkLight.

As in Fork and ForkLight, NestStep processors are organized in groups, as described in Section 4.1. The `main` method of a NestStep program is executed by all available processors of the partition of the parallel computer the program is running on. Groups can be dynamically subdivided during program execution, following the static nesting structure of the supersteps.

4.5.3 Supersteps and Nested Supersteps

The term ***step statement*** covers both the `step` statement and the `neststep` statement of NestStep. Step statements denote supersteps that are executed by entire *groups* of processors in a bulk-synchronous way. Hence, a step statement always expects all processors of the current group to arrive at this program point.

The relationship between processor groups and step statements is determined by the following invariant:

> ***Superstep synchronicity***
>
> *All processors of the same group are guaranteed to work within the same step statement.*

The step statements also control shared memory consistency within the current group, as will be explained in Section 4.5.5.

The `step` statement

`step` *statement*

implicitly performs a groupwide barrier synchronization at the beginning and the end of *statement*. Note that these barriers are only conceptual. In any case, the implementation will try to avoid duplicate barriers and integrate barriers into combine phases.

Supersteps can be nested, as visualized in Figure 4.38. The nestable variant of the `step` statement is the `neststep` statement. A `neststep` statement deactivates and splits the current group into disjoint subgroups. Each subgroup executes *statement*, independently of the others, as a superstep. The parent group is reactivated and resumes when all subgroups finished execution of the `neststep` body. As `neststep`s can be nested statically and dynamically, the group hierarchy forms a tree at any time of program execution, where the leaf groups are the currently active ones.

The first parameter of a `neststep` statement specifies the number of subgroups that are to be created. It needs not be a constant, but its value must be equal on all processors of an executing group. An optional second parameter indicates how to determine the new subgroup for each processor. Creating a single subgroup may make sense if only a part of the group's processors is admitted for the computation in the substep, e.g. at one-sided conditions.

`neststep (` k `)` *statement*

splits the current group (let its size be p) into k subgroups of size $\lceil p/k \rceil$ or $\lfloor p/k \rfloor$ each. The k subgroups are consecutively indexed from 0 to $k-1$; this subgroup index can be read in *statement* as the *group ID* `@`. Processor i, $0 \leq i \leq p-1$ of the split group joins subgroup $i \text{mod} k$. If $p < k$, the last $k-p$ subgroups are not executed.

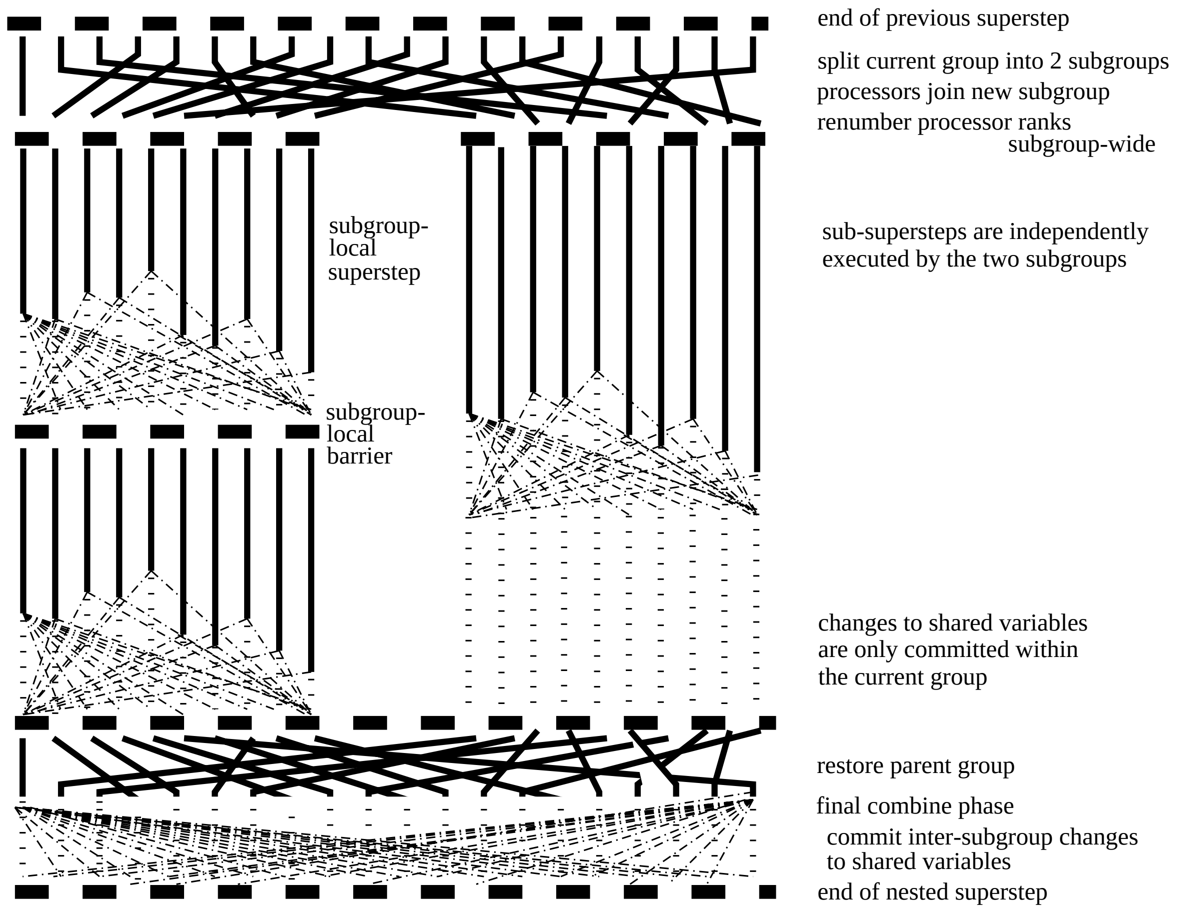

end of previous superstep
split current group into 2 subgroups
processors join new subgroup
renumber processor ranks
subgroup-wide
subgroup-local superstep
sub-supersteps are independently executed by the two subgroups
subgroup-local barrier
changes to shared variables are only committed within the current group
restore parent group
final combine phase
commit inter-subgroup changes to shared variables
end of nested superstep

FIGURE 4.38: Nesting of supersteps, here visualized for a `neststep(2,...) ...` statement splitting the current group into two subgroups. Dashed horizontal lines represent implicit barriers.

In order to avoid such empty subgroups, the programmer can specify by

`neststep (` k`, #>=1 )` *statement*

that each subgroup should be executed by at least one processor. In that case, however, some of the subgroups will be executed serially by the same processors. Thus the programmer should not expect all subgroups to work concurrently. Also, the serial execution of some subgroups needs not necessarily be in increasing order of their group indices `@`.

In some cases a uniform splitting of the current group into equally-sized subgroups is not optimal for load balancing. In that case, the programmer can specify a weight vector to indicate the expected loads for the subgroups:

`neststep (` k`, weight )` *statement*

Here, the weight vector `weight` must be a replicated shared array (see later) of k (or more) nonnegative floats. If the weight vector has less than k entries, the program will abort with an error message. Otherwise, the subgroup sizes will be chosen as close as possible to the weight ratios. Subgroups whose weight is 0 are not executed. All other subgroups will have at least one processor, but if there are more (k') nonzero weights than processors (p) in the split group, then the $k' - p$ subgroups with least weight will not be executed.

The group splitting mechanisms considered so far rely only on locally available information and are thus quite efficient to realize [BGW92]. Now we consider a more powerful construct that requires groupwide coordination:

`neststep ( k, @=` *intexpr* `)` *statement*

creates k new subgroups. Each processor evaluates its integer-valued expression *intexpr*. If this value e is in the range $[0...k-1]$, the processor joins the subgroup indexed by e. Since the subgroup to join is a priori unknown (*intexpr* may depend on runtime data), the proper initialization of the subgroups (size, ranks) inherently requires another groupwide combine phase[22] that computes multiprefix-sums (cf. [C16]) across the entire split group.

If for a processor the value e of *intexpr* is not in the range $[0..k-1]$, then this processor skips the execution of *statement*.

4.5.4 Supersteps and Control Flow

The usual control flow constructs like `if`, `switch`, `while`, `for`, `do`, `?:`, && and `||` as well as jumps like `continue`, `break`, and `return` (and exceptions in Java) can be arbitrarily used within step statements.

Nevertheless the programmer must take care that step statements are reached by all processors of the current group, in order to avoid deadlocks. As long as the conditions affecting control flow are *stable*, i.e. are guaranteed to evaluate to the same value on each processor of the current group, this requirement is met. Where this is not possible, for instance, where processors take different branches of an `if` statement and step statements may occur within a branch, as in

```
if (cond)          // if stmt1 contains a step:
     stmt1();      //  danger of deadlock, as
else       stmt2();// these processors don't
                   //  reach the step in stmt1
```

a `neststep` statement must be used that explicitly splits the current group:[23]

```
neststep(2; @=(cond)?1:0) // split group into 2 subgroups
  if (@==1) stmt1();      // a step in stmt1 is local
  else      stmt2();      //  to the first subgroup
```

[22]This combine phase may, though, be integrated by the compiler into the combine phase of the previous step statement if k and *intexpr* do not depend on shared values combined there.

[23]It is, in general, not advisable here to have such insertion of group splitting `neststep`s automatically by the compiler, since the inner step statements to be protected may be hidden in method calls and thus not statically recognizable. Also, paranoic insertion of group-splitting `neststep`s at any point of potential control flow divergence would lead to considerable inefficiency. Finally, we feel that all superstep boundaries should be explicit to the NestStep programmer.

Similar constructs hold for `switch` statements. For the loops, a step statement within the loop body will not be reached by all processors of the group if some of them stop iterating earlier than others. Thus, the current group has to be narrowed to a subgroup consisting only of those processors that still iterate. A `while` loop, for instance,

```
while ( cond ) // may lead to deadlock
   stmt();     // if stmt() contains a step
```

can be rewritten using a private flag variable as follows:

```
boolean iterating = (cond);
do
   neststep < 1; @ = iterating? 0:-1 >
      stmt();    // a step inside stmt is local
                 // to the iterating subgroup
while (iterating = iterating && (cond));
```

Note that the condition `cond` is evaluated as often and at the same place as in the previous variant. Once a processor evaluates `iterating` to 0, it never again executes an iteration. Note also that now the iterations of the `while` loop are separated by implicit barriers for the iterating group.

Processors can jump out of a step statement by `return`, `break`, `continue`, or an exception. In these cases, the group corresponding to the target of the jump is statically known and already exists when jumping: it is an ancestor of the current group in the group hierarchy tree. On their way back through the group tree from the current group towards this ancestor group, the jumping processors cancel their membership in all groups on this path. They wait at the step statement containing their jump target for the other processors of the target group.

Jumps across an entire step statement or into a step statement are forbidden, since the group associated with these jump target step statements would not yet be existing when jumping. As there is no `goto` in Java, such jumps are mostly syntactically impossible. Nevertheless, processors jumping via `return`, `break` and `continue` may skip subsequent step statements executed by the remaining processors of their group. For these cases, the compiler either warns or recovers by applying software shadowing of the jumping processors to prevent the other processors from hanging at their next step statement.

Jumps within a leaf superstep, that is, within a step statement that contains no other step statement, are harmless.

Group inspection

For each group there is on each processor belonging to it a class `Group` object. In particular, it contains the group size, the group ID, and the processor's rank within the group. (see Table 4.6). The `Group` object for the current group is referenced in NestStep programs by `thisgroup`. `thisgroup` is initialized automatically when the group is created, updated as the group processes step statements, and restored to the parent group's `Group` object when the group terminates.

At entry to a step statement, the processors of the entering group are ranked consecutively from 0 to the group size minus one. This group-local processor ID can be accessed

Publically accessible fields and access methods:

`int`	`size`	size of this group at entry to the current step
`int`	`rank`	my rank in this group at entry to current step
`int`	`sizeinit`	size of this group when it was created
`int`	`rankinit`	my initial rank when this group was created
`int`	`gid`	this group's ID
`Group`	`parent`	ref. to my `Group` object for the parent group
`int`	`depth`	depth of this group in the hierarchy tree
`int`	`path`	string indicating the group's hierarchy path
`int`	`count`	counter for supersteps executed by this group

NestStep shorthands for ease of programming:

`thisgroup`	expands to `Group.myGroup`
`#`	expands to `thisgroup.size()`
`$$`	expands to `thisgroup.rank()`
`$`	expands to `thisgroup.rankinit()`
`@`	expands to `thisgroup.gid()`

TABLE 4.6: Fields and shorthands for `Group` objects in NestStep.

by `thisgroup.rank()` or, for convenience, just by the symbol `$$`. Correspondingly, the current group's size `thisgroup.size()` is abbreviated by the symbol `#`, and the group ID `thisgroup.gid()` by the symbol `@`.

If some processors leave a group earlier by `break`, `continue` or `return`, the `size` and `rank` fields of the group will be updated at the end of the corresponding superstep.

g`.depth()` determines the depth of the group *g* in the group hierarchy tree. The parent group of a group *g* can be referenced by *g*`.parent()`. This allows to access the `Group` object for any ancestor group in the group hierarchy tree. *g*`.path()` produces a string like `0/1/0/2` that represents the path from the root group to the calling processor's current group in the group hierarchy tree by concatenating the `gid`s of the groups on that path. This string is helpful for debugging. It is also used by the NestStep run-time system for uniquely naming group-local shared variables and objects.

Sequential parts

Statements marked sequential by a `seq` keyword

`seq` *statement*

are executed by the group leader only. The *group leader* is the processor with rank `$$==0`. Note that, as in ForkLight, the `seq` statement does not imply a barrier at its end. If such a barrier is desired, `seq` has to be wrapped by a `step`:

`step seq` *statement*

Note that, if the initially leading processor leaves the group before the "regular" end of a step statement via `break`, `continue` or `return`, another processor becomes leader of the group after the current sub-step, and thus responsible for executing future unparameterized seq statements. Nevertheless, maintaining a leader of the group involves some run-time overhead, as we will see in Section 5.4.

Discussion: Explicit versus implicit group splitting

In Fork and ForkLight, the maintenance of the synchronicity invariant is performed automatically by the implementation, using implicit subgroup creation and implicit barriers. In contrast, the policy of NestStep is to make all subgroup creations and all barriers explicit, hence the programmer must be aware of them. This break in the tradition is motivated as follows:

- *Clarity.* The programmer should have an idea of the complexity of the resulting group hierarchy tree.
- *Performance.* On a distributed memory system, subgroup creation and barriers are expensive operations; their cost scales at least logarithmically in the group size. In contrast, on a tightly coupled system such as the SB-PRAM, such operations required only a small constant overhead. The programmer should be aware of these high-overhead constructs by having them marked explicitly in the program code.
- *Optimizations.* Some subgroup creations can be avoided because the programmer has additional knowledge about his application that the compiler cannot deduce from static program analysis. Also, the programmer can choose the weakest possible variant of the neststep statement that still matches the desired subgroup creation effect.

Clearly, implicit subgroup creation and implicit barriers would provide more programming comfort, especially for Fork and ForkLight programmers. We could imagine a flexible solution in future implementations of NestStep by offering a compiler option that enables implicit subgroup creation and barriers, as known from Fork.

4.5.5 Sharing Variables, Arrays, and Objects

Variables, arrays, and objects are either *private* (local to a processor) or *shared* (local to a group). Sharing is specified by a type qualifier sh at declaration and (for objects) at allocation.

Shared base-type variables and heap objects are replicated. Shared arrays may be replicated or distributed; replication is the default. Pointers, if supported by the base language, are discussed in Section 4.5.7.

By default, for a *replicated* shared variable, array, or object, one local copy exists on each processor of the group declaring (and allocating) it.

Shared arrays are stored differently from private arrays and offer additional features. A shared array is called *replicated* if each processor of the declaring group holds a copy of all elements, and *distributed* if the array is partitioned and each processor owns a partition exclusively. Like their private counterparts, shared arrays are not passed by value but by reference. For each shared array (also for the distributed ones), each processor keeps its element size, length and dimensionality at runtime in a local array descriptor. Hence, bound-checking (if required by the basis language) or the `length()` method can always be executed locally.

For nondistributed arrays and for objects it is important to note that sharing refers to the entire object, not to partial objects or subarrays that may be addressed by pointer arithmetics. This also implies that a shared object cannot contain private fields; it is updated as an entity[24].

Sharing by replication and combining

For replicated shared variables, arrays, and objects, the step statement is the basic control mechanism for shared memory consistency:

> ***Superstep consistency***
>
> *On entry to a step statement holds that on all processors of the same active group the private copies of a replicated shared variable (or array or heap object) have the same value. The local copies are updated only at the end of the step statement.*

Note that this is a deterministic hybrid memory consistency scheme: a processor can be sure to work on its local copy exclusively within the step statement.

Computations that may write to shared variables, arrays, or objects, must be inside a step statement.

Replicated shared variables

At the end of a step statement, together with the implicit groupwide barrier, there is a groupwide *combine phase*. The (potentially) modified copies of replicated shared variables are combined and broadcast within the current group according to a predefined strategy, such that all processors of a group again share the same values of the shared variables. This ***combine strategy*** can be individually specified for each shared variable at its declaration, by an extender of the `sh` keyword:

- `sh<0> type x;` declares a variable `x` of arbitrary type `type` where the copy of the group leader (i.e., the processor numbered 0) is broadcast at the combine phase. All other local copies are ignored even if they have been written to.

- `sh<?> type x;` denotes that an arbitrary updated copy is chosen and broadcast. If only one processor updates the variable in a step, this is a deterministic operation.

- `sh<=> type x;` declares a shared variable for which the programmer asserts that it is always assigned an equal value on all processors of the declaring group. Thus, no additional combining is necessary for it.

[24]With Java as basis language, shared objects must be `Serializable`.

- `sh<+> arithtype x;`, applicable to shared variables of arithmetic types, means that all local copies of `x` in the group are added, the sum is broadcast to all processors of the group, and then committed to the local copies. This is very helpful in all situations where a global sum is to be computed, e.g. in linear algebra applications. There are similar predefined combine strategies for multiplication, bitwise AND / OR, and user-defined binary associative combine functions.

  Here is an example:

  ```
  sh<+> float sum;  // automat. initialized to 0.0
  step
     sum = some_function( $ );
  // here automatic combining of the sum copies
  seq System.out.println("global sum: " + sum );
  ```

- `sh<*> arithtype x;` works in the same way for global multiplication,
- `sh<&> int x;` for global bitwise AND computation, and
- `sh<|> int x;` for global bitwise OR computation, the latter two being defined for the integral types or `boolean`.
- `sh<foo> type x;` specifies the name of an arbitrary associative user-defined method `type foo(type,type)` as combine function. Clearly `foo` itself should not contain references to shared variables nor steps.

Since combining is implemented in a treelike way (see Section 5.4), prefix sums can be computed on-the-fly at practically no additional expense. In order to exploit these powerful operators, the programmer must specify an already declared private variable of the same type where the result of the prefix computation can be stored on each processor:

```
float k;
sh<+:k> float c;
```

The default sharity qualifier `sh` without an explicit combine strategy extender is equivalent to `sh<?>`.

The declared combine strategy can also be overridden for individual steps by a `combine` annotation at the end of the step statement. Note however that within each superstep there can only one combine strategy be associated with each certain shared variable or shared array element. In fact, programs will become more readable if important combinings are explicitly specified. For instance, at the end of the following step

```
sh int a, b, c;   // default combine strategy <?>
int k;
......
step {
   a = 23;
   b = f1( $ );
   c = f2( b );   // uses modified copy of b
}
combine ( a<=>, b<+:k>, c<+> );
```

```
#include <stdio.h>
#include <Neststep.h>

#define N 1000000

float f( float x )
{
  return  4.0 / (1.0 + x*x );
}

void main( int argc, char *argv[] )
{
  sh float pi;
  sh float h<=>;   // shared constant
  int i;

  step {
    h = 1.0 / (float)N;
    for (i=$$; i<N; i+=#)
      pi += f( h*((float)i - 0.5));
  }
  combine ( pi<+> );

  step {
    pi = pi*h;
    if ($$==0)
      printf("PI=%f\n", pi );
  }
  combine ( pi<=> );
}
```

FIGURE 4.39: Example code: Pi calculation in NestStep

the local copy b_i^{new} of b on processor i is assigned the sum $\sum_{j=1}^{p} b_j$ and the private variable k on processor i is assigned the prefix sum $\sum_{j=1}^{i} b_j$. Also, the copies of c are assigned the sum of all copies of c. The combining for a can be omitted for this `step`.

Example: Pi calculation

The example program in Figure 4.39 performs a calculation of π following an integration approach. The local sums are collected in the local copies of the shared variable `pi` in the first superstep. After that superstep, the local contributions are added, and now each copy of `pi` contains the global sum of all copies. In the second superstep (which is only needed for didactic purposes, not for the computation itself), `pi` is scaled and the result is printed. Another combining of the `pi` copies at the end of the second superstep is not necessary, although they have been written to; this is indicated by the `pi<=>` combine method that skips combining of `pi` for that superstep.

Replicated shared arrays

For a replicated shared array, each processor of the declaring group holds copies of all elements. The combining strategy declared for the element type is applied elementwise. The declaration syntax is similar to standard Java or C arrays:

```
 sh<+> int[] a;
```

declares a replicated shared array of integers where combining is by summing the corresponding element copies.

A shared array can be allocated (and initialized) either statically already at the declaration, as in

```
sh int a[4] = {1, 3, 5, 7};
```

or later (dynamically) by calling a constructor (in NestStep-Java)

```
a = new sh int[N];
```

or

```
a = new_Array ( N, Type_int );
```

in NestStep-C, respectively.

Discussion: Volatile Shared Variables, Arrays, Objects

The first approach to NestStep offered support for so-called *volatile shared variables*, arrays, and objects, for which the implementation guarantees sequential consistency. This feature has been removed in the current design of NestStep.

A shared variable declared `volatile` is not replicated. Instead, it is owned exclusively by one processor of the declaring group. This owner acts as a data server for this variable. Different owners may be specified for different shared volatile variables, in order to balance congestion. Other processors that want to access such a variable will implicitly request its value from the owner by one-sided communication and block until the requested value has arrived. Thus, accessing such variables may be expensive. Explicit prefetching is enabled by library routines; hence, the communication delay may be padded with other computation. Beyond its ordinary group work, the owner of a volatile shared variable has to serve the requests. Because all accesses to volatile shared variables are sequentialized, *sequential consistency* is guaranteed for them even within a superstep. Moreover, atomic operations like *fetch&add* or *atomic_add* are supported by this mechanism.

The same, user-relaxable sequential consistency scheme was used for distributed arrays in the original approach, such that, in principle, each array element has exactly one owner processor that is responsible for serving all access requests to it from remote processors. The run-time library offered bulk-mirror and bulk-update functions for entire sections of arrays, which results in faster communication but relaxes sequential consistency.

Volatile shared variables have been intended for serving as global flags or semaphores that implement some global state of computation and thus should be able to be read and written to by any processor at any time, and that the result of a write access is made globally visible immediately.

```
int a[7]<%>;
int b[7]</>;
```

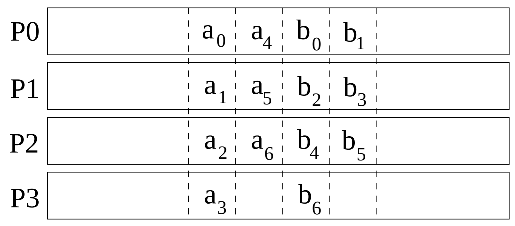


FIGURE 4.40: Cyclic and block distribution of array elements across the processors of the declaring group.

On the other hand, volatile shared variables do not have a direct correspondence to the BSP model, because the BSP model supports sequential consistency only at the border between two subsequent supersteps. Moreover, the implementation of the—originally required—sequential consistency for the distributed arrays may lead to extremely inefficient code; note that many distributed shared memory systems use weaker forms of consistency to be efficient in practice. For these reasons, volatile shared variables, arrays, and objects have been dropped in the current NestStep design.

4.5.6 Distributed Shared Arrays

Only shared arrays can be distributed. Typically, large shared arrays are to be distributed to save space and to exploit locality. While, in the early NestStep approach, distributed arrays were volatile by default, such that each array element was sequentially consistent, the current language standard defines for distributed arrays the same superstep consistency as for replicated shared variables, arrays, and objects. The modifications to array elements become globally visible only at the end of a superstep. Hence, the processor can be sure to work on its local copy exclusively until it reaches the end of the superstep. For concurrent updates to the same array element, the corresponding declared (or default) combine policy is applied, as in the scalar case. This strategy produces more efficient code and is more consistent with respect to the treatment of scalar variables. Also, it is no longer necessary that the programmer takes over control for locality and enforce explicit local buffering of elements by the now obsolete bulk-mirror and bulk-update methods for performance tuning purposes.

The various possibilities for the distribution of arrays in NestStep are inspired by other parallel programming languages for distributed memory systems, in particular by Split-C [CDG$^+$93].

Distributed shared arrays are declared as follows. The distribution may be either in contiguous blocks or cyclic. For instance,

```
sh int[N]</> b;
```

denotes a block-wise distribution with block size $\lceil N/p \rceil$, where p is the size of the declaring group, and

```
sh int[N]<%> a;
```

denotes a cyclic distribution, where the owner of an array element is determined by its index modulo the group size. Such distributions for a 7-element array across a 4 processor group are shown in Figure 4.40.

Multidimensional arrays can be distributed in up to three “leftmost” dimensions. As in Split-C [CDG+93], there is for each multidimensional array A of dimensionality d a statically defined dimension k, $1 \leq k \leq d$, $k \leq 3$, such that all dimensions $1, ..., k$ are distributed (all in the same manner, linearized row-major) and dimensions $k+1, ..., d$ are not, i.e. $A[i_1, ..., i_d]$ is local to the processor owning $A[i_1, ..., i_k]$. The distribution specifier (in angle brackets) is inserted between the declaration brackets of dimension k and $k+1$. For instance,

```
sh int A[4][5]<%>[7]
```

declares a distributed array of $4 \cdot 5 = 20$ pointers to local 7-element arrays that are cyclically distributed across the processors of the declaring group.

The distribution becomes part of the array’s type and must match e.g. at parameter passing. For instance, it is a type error to pass a block-distributed array to a method expecting a cyclically distributed array as parameter. As NestStep-Java offers polymorphism in method specifications, the same method name could be used for several variants expecting differently distributed array parameters. In NestStep-C, different function names are required for different parameter array distributions.

Scanning local index spaces

NestStep provides parallel loops for scanning local iteration spaces in one, two and three distributed dimensions. For instance, the `forall` loop

```
forall ( i, a )
   stmt(i, a[i]);
```

scans over the entire array `a` and assigns to the private iteration counter `i` of each processor exactly those indices of `a` that are locally stored on this processor. For each processor, the loop enumerates the local elements upward-counting. The local iteration spaces may be limited additionally by the specification of a lower and upper bound and a stride:

```
forall ( i, a, lb, ub, st )
   stmt(i, a[i]);
```

executes only every `st`th local iteration `i` located between `lb` and `ub`. Downward counting local loops can be obtained by setting `lb`>`ub` and `st`< 0.

Generalizations `forall2`, `forall3` for more than one distributed dimension are straightforward, using a row-major linearization of the multidimensional iteration space following the array layout scheme.

Array syntax, as in Fortran90 and HPF, has been recently added to NestStep to denote fully parallel operations on entire arrays. For example, `a[6:18:2]` accesses the elements `a[6]`, `a[8]`, ..., `a[18`.

For each element `a[i]` of a distributed array, the method `a.owner(i)` resp. the function `owner(a[i])` returns the ID of the owning processor. Ownership can be computed either already by the compiler or at run-time, by inspecting the local array descriptor.

The boolean predicate `a.owned(i)` resp. `owned(a[i])` returns true iff the evaluating processor owns `a[i]`.

```
void parprefix( sh int a[]</> )
{ int *pre;      // local prefix array
  int Ndp = N/p; // assuming p divides N for simplicity
  int myoffset;  // prefix offset for this processor
  sh int sum=0;  // constant initializers are permitted
  int i, j = 0;
  step {
    pre = new_Array( Ndp, Type_int );
    sum = 0;
    forall ( i, a ) {  // locally counts upward
       pre[j++] = sum;
       sum += a[i];
    }
  } combine( sum<+:myoffset> );
  j = 0;
  step
    forall ( i, a )
      a[i] = pre[j++] + myoffset;
}
```

FIGURE 4.41: Computing parallel prefix sums in NestStep-C.

Example: Parallel prefix computation

Figure 4.41 shows the NestStep-C implementation of a parallel prefix sums computation for a block-wise distributed array a.

Accessing distributed array elements

Processors can access only those elements that they have locally available.

A nonlocal element must be fetched from its owner by an explicit request at the beginning of the superstep that contains the read access (more specifically, at the end of the previous superstep). In particular, potential concurrent updates to that element during the current superstep, either by the owner or by another remote processor, are not visible to the processor; it obtains the value that was globally valid at the beginning of the current superstep. The necessary fetch instructions can be generated automatically by the compiler where the indices to be requested can be statically determined. Alternatively, the programmer can help the compiler with a prefetching directive.

A write access to a nonlocal element of a distributed shared array is immediately applied to a local copy of that element. Hence, the new value could be reused locally by a writing processor, regardless of the values written simultaneously to local copies of the same array element held by other processors. At the end of the superstep containing the write access, the remote element will be updated in the groupwide combine phase according to the combining method defined for the elements of that array. Hence, the updated value will become globally visible only at the beginning of the next superstep. Where the compiler cannot determine statically which elements are to be updated at the end of a superstep, this can be done dynamically by the runtime system. A poststore directive for the compiler could also be applied.

Prefix combining is not permitted for distributed array elements.

Bulk mirroring and updating of distributed array sections

Bulk mirroring and updating of distributed array elements avoids the performance penalty incurred by many small access messages if entire remote regions of arrays are accessed. This is technically supported by the use of array syntax and prefetching and poststoring directives (`mirror` and `update`), as the compiler cannot always determine statically which remote elements should be mirrored for the next superstep.

Example: BSP p-way Randomized Quicksort

Appendix C.1 contains a NestStep-C implementation of a simplified version (no oversampling) of a p-way randomized Combine-CRCW BSP Quicksort algorithm by Gerbessiotis and Valiant [GV94]. The algorithm makes extensive use of bulk movement of array elements, formulated in NestStep as read and write accesses to distributed arrays.

Array redistribution within a group

Array redistribution, in particular redistribution with different distribution types, is only possible if a new array is introduced, because the distribution type is a static property of an array. Redistribution is then just a parallel assignment.

Array redistribution at group splitting

Redistribution may be required at a group-splitting `neststep` statement. For that purpose, the runtime system provides the routines `importArray` and `exportArray`. `importArray` must be called immediately after entry into the subgroups, that is, before control flow may diverge[25]. Accordingly, `exportArray` should be positioned at the exit of the `neststep` statement, such that all processors of the parent group can participate in it.

For instance, consider a group that has allocated a distributed array `a`. By a `neststep(2,...)`, the group is split into two subgroups. Each of the subgroups needs to have access to some elements of `a`, but not necessarily to just those that are owned by the accessing subgroup's processors. In order to keep updates to elements of `a` group-local while maintaining a distributed storage scheme, `a` must be redistributed across each of the subgroups. The redistribution will pay off for better performance if it is amortized over several subsequent accesses that are now local.

Another problem that is solved by redistribution is the case where updates to `a` from different subgroups conflict with each other. For this reason, it is also not sufficient to create

[25]There is a chicken-and-egg problem with the specification of such redistribution, be it in the form of directives, language keywords, or calls to the runtime system, because the redistribution must reference subgroup-local distributed arrays that are declared only within the `neststep` body, while the redistribution itself is global to the subgroups, as all processors of the parent group must participate in it, and thus it should somehow precede the `neststep` body. We have decided for the solution where the specification of the redistribution is moved into the `neststep` body, after the declarations, but prior to any divergence of control flow.

new *pointers* to the existing array elements but copy the array elements themselves at subgroup creation, and copy them back at subgroup termination. Hence, group-local modifications go to the copies and will hence only be visible within the modifying subgroup.

Redistribution for a distributed array at subgroup entry is done with `importArray`:

```
sh int a[]<%> = ...
...
neststep(2, ...) {    // split group into 2 subgroups
    sh int[]<%> aa;
    aa = importArray ( a );
    ... use(aa) ...
}
```

`importArray()` virtually blocks the executing processor until it has received all elements of the array being redistributed that it is going to own. Finally, each `aa` array holds a copy of array `a`, distributed across each individual subgroup.

Usually it is not the case that each subgroup needs to access every element of `a`. Generally, only the programmer knows which elements are really needed by which subgroup. Thus, (s)he may tell this to the compiler, using array syntax to specify the necessary subarray elements to be redistributed. The last superstep in the recursive parallel Quicksort example in Figure 4.42 gives an illustration.

When the subgroups cease to exist at the end of a `neststep` execution, the arrays mirrored subgroup-wide by `importArray` are freed automatically. If values assigned to them by the subgroups should be written back to the distributed array of the ancestor group, this must be done explicitly by a corresponding group-export statement like

```
exportArray ( aa, a[6:18:2]);
```

If several subgroups export different values to the same location of `a`, it is not determined which of these values will finally be stored.

Admittedly, the realization of the redistribution as calls to library routines is probably not the best choice from a puristic point of view, because `importArray` seems to cause communication also at the *beginning* of a (nested) superstep. From a pragmatic point of view, this is less of a problem, because the `importArray` calls may be thought of as still belonging to the communication phase at the end of the preceding superstep, which is also done in the implementation. Finally, one could invent an extension of the `neststep` syntax that takes care for the redistributions.

4.5.7 Pointers

In NestStep-Java there are no pointers; references to objects are properly typed. In NestStep-C, pointer variables could be declared as shared or private. Nevertheless, the sharity declaration of a “shared pointer” variable `p` like

```
sh<+> int *p;
```

```
void qs( sh int[]<%> a )
{  // a is a distributed array of n shared integers
 sh<=> int l, e, u;
 sh<?> int pivot;
 sh float weight[2];      // replicated shared array
 int j,  n = a.length();
 if (n<=THRESHOLD) { seq seqsort( a ); return; }
 if (#==1)         { seqsort( a ); return; }
 do {       // look for a good pivot element in a[]:
   step { l = e = u = 0;
          j = randomLocalIndex( a );
          pivot = a[j];
   } // randomly selects pivot among first # elements
   step { // in parallel determine sizes of subarrays:
     forall(j,a) // parallel loop over owned elements
        if      (a[j]<pivot)  l++;
        else if (a[j]>pivot)  u++;
             else             e++;
   } combine ( l<+>, e<+>, u<+> ); // group-wide sums
 } while (! balanceOK(l,u,n)); // pivot bad: try again

 // do partial presort in place in parallel:
 partition( a, pivot, l, e, u );

 weight[0]=Math.max((float)(l)/(float)(l+u),1/(float)#);
 weight[1]=Math.max((float)(u)/(float)(l+u),1/(float)#);
 neststep( 2, weight ) {
   sh int[]<%> aa;  // subgroup-wide distribution
   thisgroup.importArray(aa, (@==0)? a.range(0,l,1)
                                   : a.range(l+e,u,1));
   qs( aa );
   if (@=0) thisgroup.exportArray(aa, a.range(0,l,1))
   else     thisgroup.exportArray(aa, a.range(l+e,u,1));
 }
}
```

FIGURE 4.42: A recursive parallel quicksort implementation in NestStep-Java. — The `partition` function, which is not shown here, uses three concurrent parallel prefix computations to compute the new ranks of the array elements in the partially sorted array. The rank array entries are incremented in parallel with the respective subarray offset, and then used as permutation vector in the runtime system function `permute` to produce a distributed temporary array that is partially sorted. Finally, the temporary array is copied back to `a` in parallel.

	Shared memory model			Message passing (MIMD)
	SIMD / dataparallel		MIMD	
Exactly synchronous execution supported	NESL, V, C*, Dataparallel C, Fortran90, Vector-C, APL, CM-Fortran	HPF, Vienna-Fortran, Fortran-D, MPP-Fortran	FORK, Fork, *ll*, pm2, Modula-2*	—
No exactly synchronous execution	—	—	ParC, Cilk, Sequent-C, PCP, LINDA, P4, OpenMP, pthreads ForkLight NestStep	CSP, Occam, PVM MPI, MPI-2

TABLE 4.7: Classification of some parallel programming languages.

refers to the *pointee*; the pointer variable itself is always private. For shared pointers we must require that (a) shared pointers may point to replicated shared variables, (whole) arrays, or objects only[26], (b) pointer arithmetics is not allowed for shared pointers, and (c) all shared pointers pointing to a (replicated) shared object x must have been declared with the same type and combine strategy as is used at the allocation of x.

Private pointers may point to any private object, or to shared replicated variables, arrays, and objects. In short, pointers can be used only as aliases for processor-local data. Hence, dereferencing a pointer never causes communication.

4.6 Other Parallel Programming Languages

In this survey (which is definitely not exhaustive), we focus mainly on imperative languages that support the shared memory programming model. Our classification is summarized in Table 4.7. More general surveys of parallel programming languages are available in the literature [BST89, ST95, GG96, ST98].

One approach of introducing parallelism into languages consists of decorating sequential programs meant to be executed by ordinary processors with library calls for communication resp. access to shared variables. Several subroutine libraries for this purpose extending C or FORTRAN have been proposed and implemented on a broad variety of parallel machines. Hence, adopting a more general view of parallel programming languages, we consider here also parallel library packages for sequential programming languages.

[26]More specifically, the shared pointer points to the local copy of the shared variable, array, or object.

4.6.1 MIMD Message Passing Environments

An early parallel programming language for a channel-based, synchronous variant of the message passing model is *CSP* [Hoa78, Hoa85]. CSP provides language constructs for the explicit decomposition of a computation into a fixed number of parallel tasks. Synchronization and communication between the tasks is explicitly coded by the programmer in the form of send and receive statements. All communication in CSP is *synchronous*, which means that the sender of a message is blocked until the receiver has acknowledged the receipt of the message. An interesting feature is the select statement that can be used to wait nondeterministically for specific messages from specific processors. The tasks are scheduled automatically to processors by the runtime system.

The language *Occam* [Inm84, JG88] is based on CSP. Occam adds configuration statements for the explicit mapping of processes to processors. Communication is synchronous and restricted to named and typed one-way links, so-called channels, between two directly connected processors. Occam has been used as the assembly language of the Inmos transputer [Inm84].

The probably most widely used library for distributed memory systems is *PVM* [Sun90, GBD$^+$94]. It offers various routines for spawning additional tasks from an existing task, aborting tasks, sending and receiving messages, and more complex communication patterns like gather, combine, and broadcast. PVM gained much popularity because it is freely available on many different parallel hardware platforms; because of compatibility reasons, PVM often is used even for programming shared memory systems, although this usually degrades performance compared with direct shared memory programming. PVM follows the fork-join style of parallel execution.

A more recent approach with similar and slightly extended functionality is the message passing interface library *MPI* [SDB93, SOH$^+$96]. There exist free and commercial implementations of MPI for many different parallel hardware platforms. MPI follows the SPMD style of parallel execution; that is, the number of processes is fixed. In MPI-2, the fork-join style of PVM was added to MPI. We have discussed some features of MPI in more detail in the context of their implementation in Fork [B1, Chap. 7.6].

4.6.2 MIMD Asynchronous Shared Memory Environments

The *P4 library*[27] and its relatives support the shared memory programming model as well. The basic primitives provided for shared memory are semaphores and locks. Moreover, it provides shared storage allocation and a flexible monitor mechanism including barrier synchronization [BL92, BL94].

Parallel threads packages for C and C++, like `pthreads` or Solaris threads, are designed mainly for multiprocessor workstations. They allow one to spawn concurrent threads in a fork-join style. Coordination of threads is by mutual exclusion locks. A similar environment is the DYNIX parallel programming library for the Sequent Symmetry. Asynchronous multithreading in a way similar to `pthreads` has also been adopted in the *Java* language.

[27]The P4 library has been implemented for the SB-PRAM [Röh96].

Linda [CG89, CGMS94] is another library approach for asynchronous MIMD programming. It models the shared memory as an associative memory, called the *tuple space*, which is accessed by the processors asynchronously by atomic put and get operations. By appropriate use of wildcards, the put and get operations provide a powerful mechanism for the coordination of asynchronous processes. There are also distributed implementations of the tuple space concept, such as in the Java extension *Jini*.

The asynchronous MIMD approach is well suited if the computations executed by the different threads of the program are "loosely coupled", that is, if the interaction patterns between them are not too complicated. Also, these libraries do not support a synchronous lockstep mode of program execution even if the target architecture does.

4.6.3 Fork-Join-Style Shared Memory MIMD Languages

MIMD programming languages adopting a fork-join execution style do generally not offer a mode of strictly synchronous program execution.

ParC [BAFR96] provides asynchronous processes that are scheduled preemptively. Parallel processes can be spawned from a single process implicitly by a `forall` loop (for concurrent execution of loop iterations) or explicitly by the `par{ :: }` statement. This results in a tree of processes where child processes share the stack of the parent process (*cactus stack*); in this way there is also an implicit sharing of variables by scoping.[28] Nevertheless, there are no global private variables. Processes are mapped automatically to processors; the programmer can optionally specify a directive to explicitly map `forall` loop iterations to processors. ParC also offers an atomic fetch&add primitive. A nice feature is that a process is able to kill all its siblings and their descendants in the process tree. On the other hand, control flow must not exit a block of statements in an abnormal way (such as `return`, `continue`, `break`).

Cilk [BJK$^+$95, BFJ$^+$96, Lei97] is a multithreaded language extending C. It is intended primarily for compilation to common distributed memory platforms. There are language constructs to define (lightweight) threads as special functions, to spawn new threads by remote function calls, and for message passing. Since version 3.0, Cilk offers a distributed shared memory model. Processors are not explicit in Cilk's programming model. The scheduling of the threads (with the goal of load balancing) is done automatically by the run-time system. Dynamic load balancing is achieved by task stealing.

Tera-C [CS90] supports multithreading by a concept called *futures*. Threads are spawned to compute a result to be stored in a certain memory location. Coordination is by an implicit lock associated with this memory location that guarantees that a subsequent read operation waits until the new value is available.

4.6.4 SPMD-Style Asynchronous Shared Memory MIMD Languages

SPMD languages, such as Denelcor HEP FORTRAN [Jor86], EPEX/FORTRAN [DGNP88], PCP [BGW92], Split-C [CDG$^+$93], or AC [CD95], start program execution with all available threads and keep their number constant during program execution. The program is usually

[28]Note that this is different from the UNIX `fork` command for spawning processes where a child process gets a *copy* of the parent's stack.

partitioned into parallel and serial sections separated by implicit barriers. Typically, these barriers are global; recursively nested parallelism with correspondingly nested barrier scopes as in Fork is not supported—usually only one global name space is supported. Hence, a parallel recursive divide-and-conquer style as suggested in other sources [BDH$^+$91, Col89, dlTK92, HR92a, HR92b] is not supported. Only PCP has a hierarchical group concept similar to that of Fork, but lacks support for synchronous execution and sequential memory consistency.

A standardization effort called *OpenMP* [Ope97] defines a set of language extensions and directives on top of FORTRAN and C. There are constructs to begin and end parallel sections, for parallel loops, for declaring shared variables, and to denote critical sections and atomic functions. The constructs have the form of compiler directives and calls to library functions; thus an OpenMP program can be compiled and executed on uniprocessor machines as well. An interesting feature of OpenMP are the so-called orphaned directives, which allow the scope of parallel constructs to be extended dynamically. The iterations of parallel loops are divided among the available parallel processors as in SPMD languages. There is no hierarchical group concept; nested parallel regions are serialized by default. We will address the shared memory consistency model of OpenMP in Section 5.2.2.

4.6.5 SIMD and Dataparallel Programming Languages

Data parallelism frequently arises in numerical computations. It consists mainly in the parallel execution of simple operations on large arrays.

SIMD languages maintain a single thread of program control. This is either naturally enforced by SIMD hardware (e.g., vector processors or array processors), or installed by the compiler when the target is a MIMD machine. To exploit SIMD parallelism, it seems quite natural to extend existing sequential programming languages by vector statements or special dataparallel operators using an array syntax and suitable intrinsic functions like vector sum, dot product, or matrix transpose. Examples are Fortran90 [MR90], CM-Fortran [AKLS88], Vector-C [LS85], or APL[Ive62] [Ive62].

Typically, all program objects are shared, and parallelism is specified in terms of special dataparallel operators or parallel loops. Nested parallelism, if supported at all, is either specified by nested parallel loops, or induced by nested data structures (like nested parallel lists) that are flattened automatically by the compiler to fit the (SIMD) hardware.

For instance, *NESL* [Ble96] is a functional dataparallel language partly based on ML. Its main data structure is the (multidimensional) list. All program objects are shared. Elementwise operations on lists are converted to vector instructions (*flattening*) and executed on SIMD machines. Related work [CSS95, ACD$^+$97] applies the hierarchical parallel list concept of NESL to the imperative languages C (resulting in a language called *V* [CSS95]) and Fortran90 (Fortran90V, [ACD$^+$97]).

C* [RS87, TPH92] is a SIMD language. It extracts parallelism from special program objects called *shapes* that are explicitly parallel arrays. A grid structure of the processors must be given; for a distributed memory machine, the compiler performs the required mapping of shape sections to the local memory modules of the processors. A group concept or recursive parallelism is not supported. The concept of parallel pointers is extremely complicated in C* since in addition to the sharity of the pointer itself, the sharity of the value pointed to is, in contrast to Fork, a part of the pointer's type specification. ViC* [CC94] is a precompiler that extends C* for out-of-core computations and thus offers a virtual memory mechanism on top of C*.

Dataparallel languages generally maintain a single thread of program control, but allow one to express slightly more general dataparallel operations using sequential and parallel loops. For instance, they allow a one-sided `if` statement or a `while` loop as in Fork, namely, non-group-splitting constructs where only a subset of the processors may participate in a computation while the others are idle. On the other hand, a two-sided `if` statement is usually translated by serialization, that is, as two subsequent one-sided `if` statements. Hence, proper task parallelism is not provided. Dataparallel languages are targeted mainly toward distributed memory SIMD and MIMD machines and thus require data layout directives to perform efficiently. Since the hardware cannot offer exact synchronicity, this is emulated by the compiler by analyzing the data dependencies and enforcing these by suitably inserted message passing code (see Section 5.2.3). Virtual processing is done implicitly by specifying array decompositions.

Dataparallel C [QH90, HQ91, HLQA92] is a dataparallel language that is synchronous at the expression level; thus a processor is either active and working synchronously with all other processors, or it is inactive. The language distinguishes between sequential and dataparallel sections of code. Dataparallel program regions are marked by a `par{}` statement. For obvious reasons, `goto` jumps across `par` boundaries are forbidden (this would have a similar effect as jumping over a synchronization point in Fork). There are three different types of sharity for a variable: `mono`, corresponding to Fork's `sh`, which means that the variable resides on the host; `poly`, which denotes a shared variable being replicated such that each processor holds a copy; and `local`, corresponding to Fork's `pr`, which means that one copy of the variable resides on each processor. Pointer declarations are still more complicated than in Fork; the sharity of the pointee must be statically known, and conversions are more restricted. A group concept does not exist.

CM-Fortran [AKLS88] extends Fortran77 with array syntax and directives for alignment and data distribution. The virtual processing supported by the operating system of the CM hardware is transparent to the programmer. Other dataparallel FORTRAN dialects are Vienna Fortran [CMZ92], Fortran-D [HKT91b], MPP-Fortran [MS94], or High-Performance Fortran (HPF) [Hig93].

Generally, the SIMD and dataparallel languages support only one global name space. Other parallel computation paradigms like a parallel recursive divide-and-conquer style are not supported. This is the domain of MIMD languages.

4.6.6 PRAM Languages

PRAM languages are MIMD languages that support synchronous execution as defined in the PRAM model. Each processor has its own program pointer, processor ID, and maybe a private address space. Moreover, sequential shared memory consistency is assumed. A PRAM language must be able to make the synchronous program execution transparent to the programmer at the language level. Such a mode of exact synchronicity must be supported by the compiler that generates code to keep program counters equal and to handle the cases where exact synchronicity may be violated as a result of diverging control flow. A hierarchical group concept supports this task considerably, as in Fork. However, some PRAM languages offer only a flat group concept, where there is only one level of exact synchronicity, only one group containing all processors.

Dataparallel variants of Modula, namely, *pm2* [Juv92b] and *Modula-2** [PT92, HHL$^+$93, PM94], support a subset of Fork's functionality. The main constructs to express parallelism are synchronous and asynchronous parallel loops. Exact synchronicity is supported only in special synchronous `forall` loops. A synchronous `forall` loop spawns a sub-PRAM with one processor for each iteration. All iterations are executed simultaneously;[29] hence nesting of parallelism is by means of nesting `forall` loops in a fork-join execution style, and sharing of variables follows the scoping rules of variable declarations in the forall loops. On the other hand, there is no explicit group concept. For asynchronous `forall` loops, which do not impose any constraints on the execution order of their iterations, Modula-2* offers a signal-wait mechanism for partial synchronization. Directives for processor allocation help the compiler to find a suitable distribution of the loop iterations across the processors. Private variables do not exist in Modula-2*. pm2 compiles to a PRAM simulator [Juv92a] while Modula-2* offers backends for several existing machines.

The only attempt of which we are aware allows both parallely recursive and synchronous MIMD programming are the imperative parallel languages FORK [HSS92], the predecessor of Fork (see Section 4.2.12, Fork itself, its successors ForkLight and NestStep, and *ll* [LSRG95].

Based on a subset of Pascal (no jumps), *ll* controls parallelism by means of a parallel *do* loop that allows a (virtual) processor to spawn new ones executing the loop body in parallel. Opposed to that, the philosophy of Fork is to take a certain set of processors and distribute them explicitly over the available tasks. Given fixed sized machines, the latter approach seems better suited to exploit the processor resources and to take advantage of synchronous execution.

4.6.7 BSP Languages

Valiant's BSP model [Val90] is an alternative to the PRAM model. Nevertheless, BSP enforces a less comfortable programming style (message passing) than does the PRAM, such that BSP programming for irregular problems becomes quite hard.

Library extensions for C supporting the BSP programming model have been developed, including those at Oxford (BSPlib, [HMS$^+$98]) and Paderborn (PUB, [BJvR99, BJvR98]); the

[29]Note that there is an important difference between a synchronous `forall` loop as in Modula-2* and a synchronous use of the `forall` macro in Fork as introduced in Section 4.2.8. In a synchronous `forall` loop in Modula-2*, all iterations are guaranteed to execute simultaneously; the compiler must perform static program analysis and apply virtualization techniques as explained in Section 5.2.1 if the number of iterations exceeds the number of available processors. This automatic virtualization means slightly more comfort for the programmer but may incur tremendous runtime overhead. In contrast, the `forall` macro of Fork is used to map a set of iterations that may, but need not, be executed in parallel, to a smaller, fixed-size group of processors by having each processor executing several iterations subsequently. Even if occurring in a synchronous region, the latter does not impose any static analysis work to the compiler, as the programmer accepts here that the scope of synchronous execution is limited to the executing group of processors and not to the overall set of iterations.

latter also supports dynamic splitting of processor groups. Processors interact by two-sided message passing or by one-sided communication [direct remote memory access (DRMA)]; collective communication is supported.

According to our knowledge, NestStep is the first proper programming language for the BSP model.

4.6.8 Integration of Task Parallelism and Data Parallelism

Several approaches have been proposed to integrate taskparallel and dataparallel features in a single programming language, in order to overcome some of the limitations of dataparallel programmming especially for irregular computations. This shift away from pure dataparallel programming was also motivated by the fact that recent parallel computing systems and supercomputer designs tend to offer massive MIMD parallelism. Bal and Haines have compared several integrated task- and dataparallel programming languages [BH98].

This combination can, in principle, be done in two different ways. One possibility is to introduce taskparallel features in an existing dataparallel language like HPF. Several approaches following this idea have been proposed, including Opus [CMRZ94], Fx [SY97], HPF2 [Sch97] (extended and implemented in Adaptor [Bra99]), and COLT$_{\text{HPF}}$ [OP99]. The construct typically adopted to allow for more individual control flow is the splitting of a group into more or less independent subgroups, as in Fork. These subgroups can then be regarded as separate submachines, and the usual features of the dataparallel basis language, like data mapping or dataparallel loops, apply to each submachine separately. In some cases like Opus, group splitting is possible only at calls to special subroutines. In other cases, like Fx or Adaptor, group splitting can be done at practically any place in the program, as in Fork. Dynamic nesting of group splitting is also possible in several approaches (Fx, Adaptor). Intersubgroup communication is more problematic in these languages. For instance, Opus allows one to define so-called shared data abstractions, a kind of shared data structure where different groups can exchange data as in a shared memory. Access to a shared data abstraction is atomic, that is, only one group can access it at the same time. Note that there is some similarity to the inter-subgroup concurrent access to globally shared data as discussed in Section 4.2.10. The taskparallel features in HPF2 are quite restricted, as there is no means for subgroups to communicate with each other, and constitute only an approved extension of the language standard. However, using the library routines defined by Brandes [Bra99], it can offer a degree of flexibility similar to that in Fx or Opus. In contrast to these "top-down" approaches which define the taskparallel units by recursively splitting a single group, the TwoL approach [RR99, FRR99, RR98, RR00] constructs the group hierarchy "bottom-up", starting with the specification of a set of dataparallel, basic program modules and then constructing taskparallel programs by (repeatedly) composing modules by sequencing or concurrent execution on different processor subsets. As there is no recursion of module composition, the resulting program can be finally represented as a series-parallel computation DAG with basic modules as nodes and (array) data flow dependence relations as edges. Scheduling this DAG for a fixed set of processors of a distributed memory architecture is a NP-hard, nonlinear static optimization problem, taking into account the complicated cost formulas for the basic modules, which are parameterized in problem sizes and array distributions, and the cost for redistributing arrays at the edges between modules where the distributions do not match.

The other possibility is to build on an existing taskparallel language such as OpenMP, Java (threads), Orca [BKT92], or an (asynchronous) MIMD version of C or FORTRAN, and introduce HPF-like directives to specify array distributions, in order to compile it to distributed memory architectures or to exploit locality of reference when compiling for cache-based shared memory machines. Currently, such an approach based on OpenMP is under investigation [CM98, LMN$^+$00]. A dataparallel extension of Orca has been described [BBJ98].

4.6.9 Skeleton Languages

The dream of making parallel programming as simple as sequential programming produced a discussion of a minimal set of extensions to sequential programming languages that should allow one to formulate a problem as a hierarchical composition of basic parallel algorithmic paradigms, such as dataparallel computation, parallel divide-and-conquer computation, parallel reduction, parallel prefix, pipelining, while abstracting from explicit parallelism and/or locality. Such extensions clearly go beyond simple library routines. They are often referred to as *skeletons* [Col89, DFH$^+$93] and regarded as higher-order functions, typically in the context of functional programming languages. Also some imperative languages have been extended by special language constructs to exploit and compose skeletons, such as P^{3}L [BDP94, Pel98], SCL [DGTY95], or Skil [BK96].

We have discussed parallel programming using skeletons and considered the implementation of several skeleton functions using Fork in Section 4.3, resulting in a structured, skeleton-oriented style of parallel programming. However, a true skeleton programming language may offer additional and more powerful features: Skeletons are not only a means for elegant and machine-independent expression of parallelism. They provide also a more powerful hint for the compiler to enable automatic transformations, generate efficient parallel code, maybe lay out data appropriately, and predict parallel run time which is again required for controlling the program transformations. A skeleton may thus be seen as a high-level interface to transfer the programmer's knowledge about the intended structure of parallel computation to the compiler, and to allow the compiler to exploit experts' knowledge for an efficient implementation on a given parallel machine.

While skeletons are intended for a top-down organization of the program, a library is better suited for bottom-up organization. At least for regular numerical applications, the bottom-up approach is suitable for passing additional knowledge to the compiler (see also Chapter 3). Ideally, the two approaches should complement each other.

4.7 Summary

Table 4.8 gives a synopsis of the most important features of the languages Fork, ForkLight and NestStep that have been described in this chapter.

As ForkLight and NestStep support dynamically nestable parallelism in the same way as Fork, the skeleton-oriented style of structured parallel programming demonstrated in Section 4.3 can be applied to ForkLight and NestStep as well.

	Fork	ForkLight	NestStep
supported machine model	PRAM	Asynchronous PRAM	BSP
memory organization	shared memory	shared memory	shared address space (explicit array distrib.)
memory consistency	sequential c.	sequential c.	superstep consistency
program execution style	SPMD	SPMD	SPMD
dyn. nested parallelism	$\surd$	$\surd$	$\surd$
group hierarchy	forest	forest	tree
group splitting	implicit/explicit dynamic	implicit/explicit dynamic	explicit static/dynamic
barrier scope	current group	current group	current group
group-wide sharing	$\surd$	$\surd$	$\surd$
concurrent write	$\surd$ resolution inherited from target architecture	not supported (asynchronous)	$\surd$ deterministic combining with programmable strategy
pointer scope	global, unique	global, restricted	local, local copies

TABLE 4.8: A synopsis of the most characteristic features of Fork, ForkLight, and NestStep.

An exception must be noted for NestStep, however. Skeletons functions for asynchronous parallel programming, in particular the asynchronous parallel task queue, can be implemented well in Fork [B1, Chap. 7.5] and ForkLight because the implementation relies on sequential memory consistency. As NestStep does not support sequential consistency, the Fork implementation of the `taskqueue` skeleton function cannot be generalized to NestStep.

Acknowledgements

The original design of the synchronous part of Fork has been given in a proposal by Hagerup, Schmitt, and Seidl [HSS92]. The redesign of Fork and the further development of the design of Fork and ForkLight was done jointly with Helmut Seidl. The locks (simple, fair and reader–writer) have been taken from a library by Jochen Röhrig for the asynchronous C programming environment of the SB-PRAM. The author thanks Murray Cole and Jesper Träff for suggestions to improve the design of Fork.

The sections 4.1, 4.2, and 4.3 follow more or less completely the presentation in Chapter 5 and Section 7.2 of *Practical PRAM Programming* [B1]. Thanks go to my co-authors Jörg Keller and Jesper Träff for proofreading that material when preparing the manuscript for Wiley. Nevertheless, the responsibility for any potentially remaining errors lies exclusively with me.

The design of NestStep contains several ideas taken from its two predecessors Fork and ForkLight. The simplified variants of the `neststep` statement for static group splitting have been inspired by corresponding constructs in PCP [BGW92]. The author thanks Vadim Engelson, Michael Gerndt, and Helmut Seidl for critical comments that helped to improve the

design of NestStep.

The section on related work was inspired by discussions with several colleagues working in the area of parallel programming languages. Thanks go also to the organizers of two seminars on high-level parallel programming at IBFI Schloß Dagstuhl, specifically, Murray Cole, Sergei Gorlatch, Chris Lengauer, Jan Prins, and David Skillicorn. Furthermore, the author thanks the organizers of a seminar on nested parallel programming environments at the university of La Laguna, specifically, Casiano Rodriguez Léon and his group.

Chapter 5

Implementation of Parallel Programming Languages

In this chapter, we focus on compilation issues for SPMD-style MIMD parallel programming languages, in particular, for Fork, ForkLight, and NestStep. We consider various types of parallel target architectures.

In Section 5.1 we discuss the compilation of Fork for the SB-PRAM, including the implementation of some important functions from the Fork standard library and the implementation of the `trv` tool. Section 5.2 introduces general techniques and problems with compiling a synchronous MIMD language such as Fork to other parallel MIMD architectures.

For the compilation of ForkLight discussed in Section 5.3 we assume a MIMD target machine with sequentially consistent shared memory and efficient support of atomic increment and atomic *fetch&increment* operations (see Fig. 4.33), such as supported by common portable shared memory platforms like the P4 library (shared memory part; [BL94]) and OpenMP [Ope97]. Special optimizations for OpenMP are proposed in Section 5.3.6. This simple and small interface enables portability across a wide range of parallel target architectures and is still powerful enough to serve as the basic component of simple locking/unlocking and barrier mechanisms and to enhance e.g. the management of parallel queues or self-scheduling parallel loops [Wil88] and occur in several routines of the ForkLight standard library.

Section 5.4 describes the compilation of NestStep for a target machine with a distributed memory.

Section 5.5 summarizes this chapter.

5.1 Compiling Fork for the SB-PRAM

The Fork compiler for the SB-PRAM, `fcc`, is partially based on `lcc 1.9`, a one-pass ANSI C compiler [FH91a, FH91b, FH95]. It generates SB-PRAM assembler code which is then processed by the assembler `prass` into COFF object code and linked by the SB-PRAM linker (see Figure 5.1).

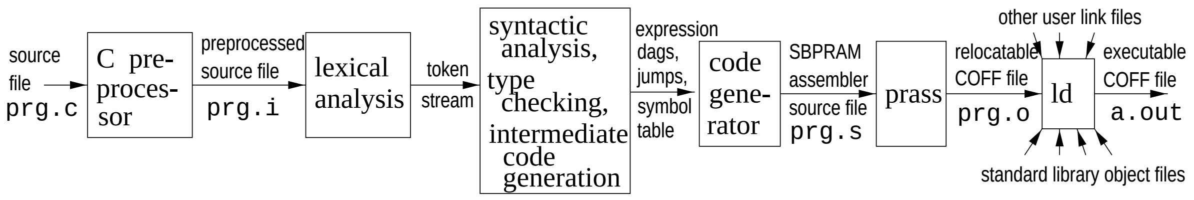


FIGURE 5.1: Phases of the Fork compiler for the SB-PRAM. The compiler driver (`fcc`) contains also calls to the C preprocessor, to the SB-PRAM assembler `prass` and the SB-PRAM linker `ld`.

5.1.1 Extensions to the C Compiler Phases and Data Structures

All compiler phases and nearly all compiler data structures had to be modified in order to allow the compilation of Fork. The lexical analysis phase has been extended to recognize the new keywords and the special symbols of Fork. The symbol table entries were extended by a field indicating the sharity of a variable and a few other flags. The type table entries for function types were extended by a field for the synchronicity and another one for the parameter sharities, which could be stored as a single-word bitvector, as C limits the number of function parameters to 32. Correspondingly, type checking had to be extended. The parser was modified to allow for declarations of sharity and synchronicity declarators, and for parsing new expressions and statements added by Fork. Correspondingly, numerous new expression tree and dag operators have been introduced. Also, the intermediate code generator must keep track of region synchronicity, shared group frame depths, and addressing of block local shared variables (see Section 5.1.6). Finally, a new compiler driver had to be written.

Most of the changes made to `lcc 1.9` concern the code generator, on which we focus in the following sections. Furthermore, a large amount of work was necessary to implement the runtime system and the standard library.

5.1.2 Shared Memory Organization and Frame Layout

The program loader of the SB-PRAM allocates the shared memory amount requested in the `.ldrc` configuration file [B1, Chap. 4] and copies to the beginning of this section the non-text segments of `a.out`: some organizational entries such as the number of processors, allocated memory size and the segment sizes, then the shared and private `.data` segments for initialized variables, the shared and private `.bss` segments for noninitialized variables, and the program arguments section. Note that the `.text` segment, which contains the program code, is stored in the separate program memory by the loader.

The startup code for the runtime system reserves shared memory space for the permanent shared heap and for the private address subspaces of the processors, loads the BASE registers of each processor, installs a shared stack and heap for the root group at the beginning, respectively at the end of the remaining shared memory block, and a private stack in each processor's private memory subspace. Each processor copies (in parallel) the private `.data` segment to the beginning of its private memory section. A shared stack pointer `sps` and a private stack pointer `spp` are permanently kept in registers on each processor.

`spp`⟶		
	last private local variable ⋮ first private local variable	not initialized
`fpp`⟶	group rank `$$`	
	old `eps` *	callee-saved
	old `gps` *	
	old `gpp` *	
	old `fps` *	
	old `fpp` *	
	old `pc`	call-saved
	last saved register ⋮ first saved register	caller-saved
	last private argument ⋮ first private argument	only if the callee has private arguments not passed in registers;
`app`⟶	old `app`	caller-saved

FIGURE 5.2: Organization of the private procedure frame for a function. The fields for the old `fps`, `gpp`, `gps` and `eps` exist only for synchronous functions. `fpp` needs not be stored if the function has no local variables.

As in common C compilers, a procedure frame is allocated at a function call on each processor's private stack. It holds private function arguments, which are pointed to by a private argument pointer `app`, saved registers, return address, and private local variables, pointed to by a private frame pointer `fpp` (see Figure 5.2). The group rank `$$` is also stored there.

`sps`⟶		
	last shared local variable ⋮ first shared local variable	the shared locals defined at top level of the function
	synchronization cell	shared group frame for the group of calling processors
`fps`, `gps`⟶	old `gps`	
	last shared argument ⋮ first shared argument	only if the callee has shared arguments;
`aps`⟶	old `aps`	caller-saved

FIGURE 5.3: Organization of the shared procedure frame for a synchronous function.

In addition, for a call to a synchronous function, a shared procedure frame (Figure 5.3) is allocated on the group's shared stack. In particular, it stores shared function arguments (pointed to by `aps`) and shared local variables declared at top level of the function (pointed to by `fps`).

The private heap is installed at the end of the private memory subspace of each processor. For each group, its group heap is installed at the end of its shared memory subspace. The group heap pointer `eps` to its lower boundary is saved at each subgroup-creating operation that splits the shared memory subspace further, and restored after returning to that group. Testing for shared stack or heap overflow thus just means to compare `sps` and `eps`.

`sps`$\longrightarrow$		
	last group-local shared variable ⋮ first group-local shared variable	the shared variables defined locally to the construct building this group
`fps`*,	synchronization cell	initialized by # processors in this group
`gps`$\longrightarrow$	old gps	points to parent shared group frame

FIGURE 5.4: Organization of the shared group frame. The `fps` is set only by `start` and `join`.

To keep the necessary information for groups, the compiler generates code to build shared and private group frames at subgroup-creating constructs. A *shared group frame* (Figure 5.4) is allocated on each group's shared memory subspace. It contains the synchronization cell, which normally contains the exact number of processors belonging to this group. At a barrier synchronization point, each processor atomically decrements this cell and waits until it sees a zero in the synchronization cell, and then atomically reincrements the synchronization cell, see Section 5.1.8.

The private components of the data structure for a group are stored in a *private group frame* on the private stack of each processor of the group. It contains fields for `$`, saved values of `eps` and `sps`, a reference to the parent group's private group frame, and some debug information. For pragmatic reasons, the group ID `@` is also stored in the private group frame instead of the shared group frame.

Intermixing procedure frames and group frames on the same stack is not harmful, since subgroup-creating language constructs like the private `if` statement, private loops, and the `fork` statement are always properly nested within a function. Hence, separate stacks for group frames and procedure frames are not required.

5.1.3 Translation of `start` and `join`

`start` and `join` allocate special group frames, as they must provide the possibility of allocating shared variables and group heap objects for the newly created group.

The code generation schema for `join` follows immediately the description in Section 4.2.9.

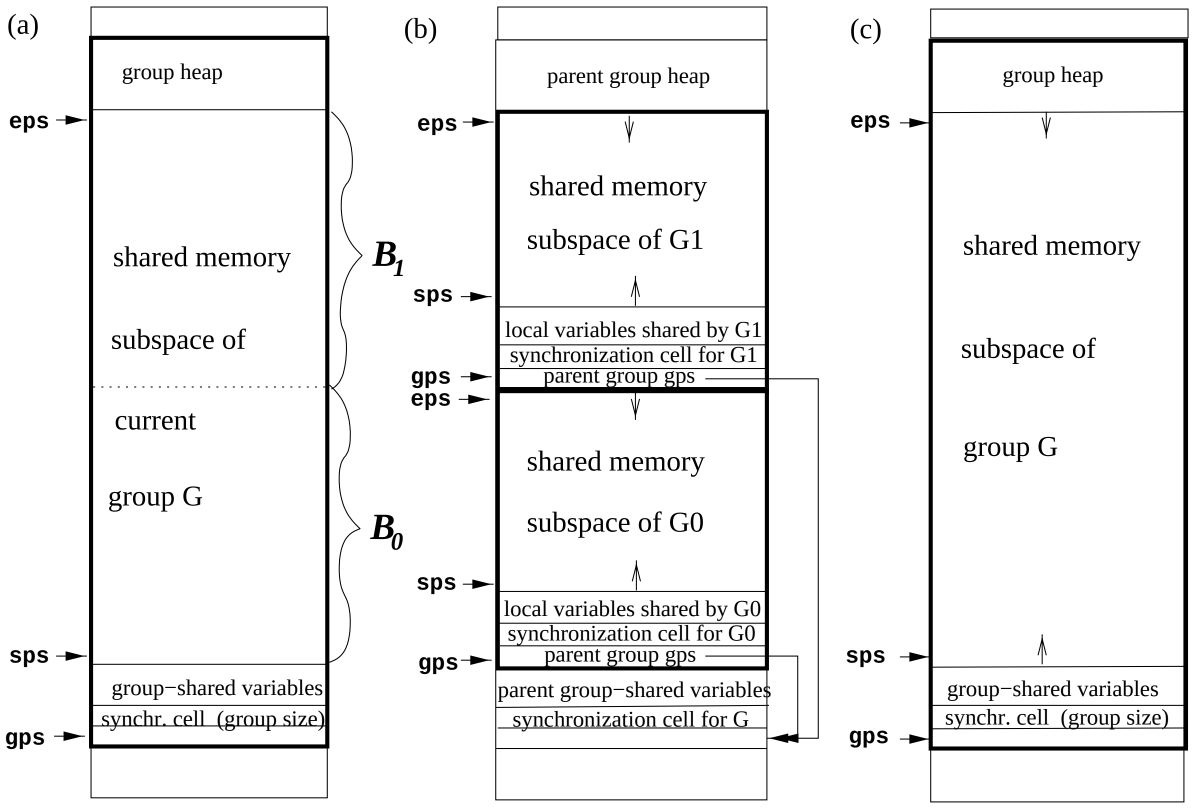


FIGURE 5.5: Group splitting at the synchronous private two-sided `if` statement: (*a*) situation before deactivating the parent group; (*b*) situation after the two subgroups have been created and activated; (*c*) situation after reactivating the parent group.

5.1.4 Translation of the Private `if` Statement

For the translation of the group-splitting constructs we consider the two-sided `if` statement as an example:

`if` (e) `statement0 else statement1`

with a private (or potentially private) condition e is translated into the following pseudocode[1] to be executed by each processor of the current group:

1. Divide the remaining free shared memory space of the current group (located between the shared stack pointer `sps` and the shared heap pointer `eps`) into two equal-sized blocks B_0 and B_1 (see Figure 5.5*a*).

2. Code for the evaluation of the condition e into a register reg.

[1]Readers interested in the SB-PRAM code can find the corresponding part of the code generator in `fork/gen.c` and the generated code by inspecting the commented assembler source file `program.s` of a Fork test program `program.c` compiled with `-S`, by searching for the keyword `split` (for the private `if` statement) and `mkgrp` (for the `fork` statement).

3. Allocate a new private group frame on the private stack. Store the current values of `gpp`, `eps`, and `sps` there, and copy the parent group's entry for `$` to the new frame. Then set `gpp` to the new private group frame.

4. If $(reg == 0)$ go to step (11).

5. Set the shared stack pointer `sps` and the group heap pointer `eps` to the limits of B_0. Set the new subgroup ID `@` to 0.

6. Allocate a new shared group frame on that new group stack. Store the shared group frame pointer `gps` of the parent group there, and set `gps` to the new shared group frame. Preset the synchronization cell `gps[1]` by zero.

7. Determine the new group's size and ranks by `mpadd(gps+1,1)`

8. Allocate space for group local shared variables defined in the activity region of the subgroup entering `statement0` (see Figure 5.5*b*).

9. Code for `statement0`.

10. Go to step 16.

11. Set the shared stack pointer `sps` and the group heap pointer `eps` to the limits of B_1. Set the new subgroup ID `@` to 1.

12. Allocate a new shared group frame on that new group stack. Store there the shared group frame pointer `gps` of the parent group, and set `gps` to the new shared group frame. Preset the synchronization cell `gps[1]` by zero.

13. Determine the new group's size and ranks by `mpadd(gps+1,1)`

14. Allocate space for group local shared variables defined in the activity region of the subgroup entering `statement1` (see Figure 5.5*b*).

15. Code for `statement1`.

16. Remove the shared and the private group frame, and restore the shared stack pointer `sps`, the group heap pointer `eps`, and the group pointers `gps` and `gpp` from the private group frame (see Figure 5.5*c*).

17. Call the groupwide barrier synchronization routine (see Section 5.1.8).

The translation scheme for the `fork` statement and for the remaining subgroup-creating constructs is similar.

Note that for synchronous loops with a private exit condition it is not necessary to split the shared memory subspace of the group executing the loop, since processors that stop iterating earlier are just waiting for the other processors of the iterating subgroup to complete loop execution.

The space subdivision among the subgroups (in this example, into two halves) assumes that the space requirements are not statically known. The shared group space fragmentation implied by this worst-case assumption for the group-splitting constructs can be partially avoided, for example, if the space requirements of a subgroup can be statically analyzed, or if the available shared memory subspace of the parent group is distributed only among those subgroups that are executed by at least one processor.

5.1.5 Groups and Control Flow

Processors that leave the current group on the "unusual" way via `break`, `return`, and `continue`, have to cancel their membership in all groups on the path in the group hierarchy tree from the current leaf group to the group corresponding to the target of that jump statement. The number of these groups is, in each of these three cases, a compile-time constant. For each group on that path, including the current leaf group, its private group frame (if existing) is released, its synchronization cell has to be decremented by a `mpadd` instruction, and the shared stack, group, and heap pointers have to be restored. Finally, the jumping processors wait at the synchronization point located at the end of the current iteration (in the case of `continue`), at the end of the surrounding loop (in the case of `break`), or directly after the call of the function (in the case of `return`), respectively, for the other processors of the target group to arrive at that point and to resynchronize with them.

As an optimization of the latter point, the Fork compiler counts the number of the so-called *harmful* return statements in a synchronous function. A return statement is called harmful if it may cause asynchronous return from a synchronous function, that is, if it is nested in a subgroup-creating construct in that function or occurs within an asynchronous region, such that not all processors of the group that entered that function may leave it together at the same time via the same return statement. Note that the natural exit point of the function does not belong to this category. If there is no harmful return statement, the barrier synchronization at return to the caller can be omitted.

In synchronous regions, a `goto` jump is executed simultaneously by all processors of a group, and hence does not affect the synchronous execution. Nevertheless, the target of `goto` may be outside the activity region of the current group, that is, it may jump across the usual group termination point. In that case, the program is likely to hang up. Hence, `goto` is not forbidden but discouraged in synchronous mode.

A `goto` in an asynchronous region is not harmful if the jump target is located in the same asynchronous region (i.e., in the body of the same `farm` or `seq` statement or an asynchronous function called from there, with no barrier existing between the source and the target of the jump). Otherwise, if the jumping processor skips a barrier statement or the barrier at the normal endpoint of the asynchronous region, the other processors of its group will wait there forever.

In C, the semantics of the expression operators &&, `||`, `?:` is defined by short-circuit evaluation. As discussed in Section 4.2.10, this is problematic in synchronous regions. The current implementation renounces on creating group frames and synchronization for these operators and emits a warning instead, which should remind the programmer to use one of the straightforward workarounds described in Section 4.2.10.

5.1.6 Accessing Shared Local Variables

The code (and hence, also the cost) of adressing a local shared variable depends on the site of definition as well as of the site of the access.

```
sync void foo( ... )
{  sh int i; // exists once for each group entering foo()
   fork (...)  {
      sh int j; // exists once for each subgroup
      fork (...) {
         sh int k; // exists once for each subgroup
         k = i * j ... ;
      }
   }
}
```

In synchronous functions, the shared variables defined at the top level of the function (e.g., variable `i` in the example code above) are addressed directly, relative to the `fps`, and thus one instruction is sufficient.

In the other cases, the chain of `gps` pointers has to be traversed backward from the current group's shared group frame to the shared group frame of that group that defined that variable and hence stores it in its shared group frame. The cost of addressing is $2x + 1$ machine cycles where x denotes the number of group frames between the current and the defining group's frame (including the defining group's frame). In the example above, calculation of the address of `j` is takes 3 machine cycles[2] while one instruction is sufficient for the calculation of the address of `k`.

In order to avoid jumping along long chains of `gps` pointers at every access to a group local shared variable, the `gps` values for all visible ancestor groups could be stored in a table [BAW96]. This increases slightly the overhead of a synchronous function call (for allocating the table) and of subgroup creation (for entering the `gps` value), but this is likely to pay off where many accesses to ancestor group local shared variables can now be done faster.

5.1.7 Runtime Overheads and Optimizations

Table 5.1 shows the time requirements of the different constructs of the language according to the current implementation. These formulas can be used for the precise static analysis of the runtime requirements of a Fork program.

Some of the entries in Table 5.1 need further explanation. On the SB-PRAM, integer and floatingpoint division must be implemented in software; this explains the large (and variable) figures for t_{idiv}, t_{imod}, and t_{fdiv}. Integer division and integer modulo computation is fast if the divisor is a power of 2; in that case the code branches to a shift operation or a bitwise AND operation, respectively. Also, in a synchronous region, an extra barrier synchronization is necessary after integer divisions with private operands, as the number of iterations made

[2]Dereferencing the `gps` pointer to obtain the parent group's shared group frame pointer requires a `ldg` instruction with one delay slot. Then a constant displacement is added to obtain the address of `j`.

TABLE 5.1: Runtimes for the Fork Language Constructs (without Tracing Code)

Language construct	Time in SB-PRAM machine cycles[a]
Exact group local barrier	$t_{\text{sync}} \leq 16 + t_{\text{lcall}}(0)$ (excluding wait time)
”, with renumbering `$$`	$t_{\text{sync}} \leq 18 + t_{\text{lcall}}(0)$ (excluding wait time)
Program startup code	$150 + 4 \cdot$ size of the private `.data` section
`start` S`;`	$27 + t_{\text{sync}} + t_S$
`join(`$e1; e2; e3$`)` S	$t_{\texttt{join}(e1,e2,e3,S)} =$ $= 120 + t_{\texttt{shmalloc}} + t_{\texttt{shfree}} + t_{\text{sync}} + t_{e2} + t_{e3} + t_S$
`if` (e) $S1$`; else` $S2$`;` (synchronous, e private)	$t_{\texttt{if}(e,S1,S2)} =$ $= 32 + t_e + \max(t_{S1} + \sigma_{S1}, t_{S2} + \sigma_{S2}) + t_{\text{sync}}$
`while(`e`)` S (synchronous, e private)	$t_{\texttt{while}(e,S)}(n_{\text{iter}}) =$ $= 8 + (6 + \sigma_S + t_S + t_e) \cdot n_{\text{iter}} + t_e + t_{\text{sync}}$
`for(`$e1; e2; e3$`)` S (synchronous, e_2 private)	$t_{\texttt{for}(e1,e2,e3,S)}(n_{\text{iter}}) =$ $= 8 + (6 + \sigma_S + t_S + t_{e2} + t_{e3}) \cdot n_{\text{iter}} + t_{e2} + t_{\text{sync}}$
`fork(`k`;@=`e_2`;$=`e_3`)` S`;`	$40 + t_{\text{idiv}}(k) + t_{e2} + t_{e3} + \max_{@=0}^{k-1}(t_S) + t_{\text{sync}}$
`farm` S	$t_S + t_{\text{sync}}$
`seq` S	$t_S + t_{\text{sync}} + 3$
Call to synchronous function	$t_{\text{scall}}(n_{\text{args}}) = 41 + 2\#(\text{saved registers}) + n_{\text{args}} + t_{\text{sync}}$
Call to asynchr./straight function	$t_{\text{acall}}(n_{\text{args}}) = 10 + 2\#(\text{saved registers}) + n_{\text{args}}$
Call to standard library function	$t_{\text{lcall}}(n_{\text{args}}) = 4 + n_{\text{args}}$
Enter block with shared locals	1
Integer division by k /modulo k	$t_{\text{idiv}}(k) = t_{\text{imod}}(k) = 12 + t_{\text{lcall}}(3)$ if k is a power of 2 $t_{\text{idiv}}(k) = t_{\text{imod}}(k) \leq 300 + t_{\text{lcall}}(3)$, otherwise (depending on k)
Floatingpoint division	$t_{\text{fdiv}} = 29 + t_{\text{lcall}}(2)$
Read pr. local var./ parameter	3
Read `@`, `$`, `$$`	3
Read sh. top-level local variable	5
Read sh. parameter, #	5
Read sh. group local variable	5 + 2 #(groups on path declaring→using group)
Read pr. global variable	4
Read sh. global variable	6
Assign to pr. local var./`$`	2
Assign to pr. parameter	2
Assign to sh. top-level local var.	4
Assign to sh. parameter	4
Assign to sh. group local var.	4 + 2 #(groups on path declaring→using group)
Assign to pr. global variable	3
Assign to sh. global variable	5
Load a 32-bit constant	2
`shalloc`	$6 + t_{\text{lcall}}(1)$
`shallfree`	3
`alloc`	$12 + t_{\text{lcall}}(1)$

[a] The figures assume the worst case with respect to alignment to the modulo flag. σ_S is 1 if S defines shared variables local to the group active at entry to S, and 0 otherwise. n_{iter} is the number of iterations made by a loop, n_{args} denotes the number of arguments passed to a function.

by the division algorithm is data-dependent. Floatingpoint division is implemented by an unrolled Newton iteration algorithm; its execution time is always the same.

The cost of function calls could be slightly reduced by passing some private arguments in registers (this is standard for most library functions).

Group split optimizations

Group split optimizations (as Käppner and Welter [Käp92, Wel92] did for the old `FORK` standard) may address the potential waste of memory in the group-splitting step of `fork` and private `if` statements. For instance, group splitting can be avoided if runtime analysis of the condition shows that all processors happen to take the same branch, that is, one of the subgroups is empty. If the shared memory requirements (subgroup stack and heap) of one branch are statically known, all remaining memory can be left to the other branch.

Synchronous loops with a shared condition may get stuck in a deadlock if a subset of the iterating processors jumps out of the loop by a `break` or `return` statement. As a conservative solution, a barrier synchronization for the loop group is hence also required at the end of loops with a shared loop exit condition if they contain a `break` or `return` inside a group splitting statement that may appear in the loop body. In certain situations, a `continue` in loops with shared exit condition may cause similar trouble, which can be avoided only if a subgroup for the iterating processors is created in that case.

Interprocedural analysis could also ascertain for (synchronous) functions whether or under what calling contexts the return value will be equal for all processors calling it. This information could be used to improve the static approximation of shared and private expressions, and may hence potentially reduce the number of subgroup creations.

Optimization of barrier calls

In the presence of an `else` part, the barrier synchronization at the end of a private `if` statement could be saved if the number of instruction cycles to be executed in both branches is statically known, because the SB-PRAM guarantees synchronous execution at the instruction level. In this case the shorter branch can be padded by a sequence or loop of `nop` instructions. The barrier can also be left out at the end of a `fork` statement if it is statically known that all subgroups take exactly the same number of instruction cycles.

Finally, the time for a barrier could be further reduced if its code were inlined in the caller.

The subgroup creation for synchronous function execution with the barrier synchronization immediately after a synchronous function call should avoid asynchrony due to processors leaving the function early via `return`. Clearly, the barrier can be suppressed if the callee does not contain `return` statements nested in subgroup-creating constructs.

The barrier at the end of a subgroup-creating construct can be omitted if a second barrier at the end of an enclosing subgroup-creating construct follows it immediately.

Generally, there are two variants of barrier synchronization that can be applied in the compiler: one that renumbers the group rank `$$`, and one that does not. The former takes two more machine cycles (two additional multiprefix operations) than the latter. Obviously, renumbering is not necessary where the ranks would not change, namely, at points where no processor

```
_barrier:
bmc      0                /*force modulo        0*/
add      R0,-1,r30        /*sync,               0*/
mpadd    gps,1,r30        /*sync,               1*/
FORKLIB_SYNCLOOP:
ldg      gps,1,r30        /*sync:loop,          0*/
getlo    1,r31            /*sync:1, replaces nop 1*/
add      r30,0,r30        /*sync:cmp ret>0?,    0*/
bne      FORKLIB_SYNCLOOP /*sync:loop if not all 1*/
ldg      gps,1,r30        /*sync:get sync cell,  0*/
mpadd    gps,1,r31        /*repair sync cell,    1*/
add      r30,0,r30        /*compare with 0,      0*/
bne      FORKLIB_SYNCHRON /*sync:jump if late,   1*/
nop                       /*sync:delay by two,   0*/
nop                       /*sync:if fast,        1*/
FORKLIB_SYNCHRON:
```

FIGURE 5.6: The barrier code used in the runtime system of the Fork compiler.

may rejoin nor leave the current group. An example situation where the nonreranking barrier is sufficient is the barrier after divisions by a private value.

Optimization of modulo alignment code

Alignment to the `modulo` flag for shared stack and heap accesses is done in a quite conservative way. Program flow analysis may be applied to find out where the value of the `modulo` flag is statically known; in these cases nothing needs to be done if the `modulo` flag will have the desired value, and a single `nop` is inserted otherwise. With a one-pass compiler like `fcc`, however, general program flow analysis is hardly possible. Instead, some postprocessing of the generated assembler code with a peephole optimizer could be applied to optimize for simple, frequently occurring situations.

Now we discuss the implementation of some important functions of the Fork standard library. We omit the details of the standard C library (which had to be implemented for the SB-PRAM as well, and partially from scratch) and focus here on the points that directly concern parallel execution.

5.1.8 Implementation of Barrier Synchronization

The exact barrier synchronization routine used by the Fork compiler for group-wide synchronization is also used for the implementation of the `barrier` statement. Its SB-PRAM assembler code is given in Figure 5.6 [B1, Chap. 4].

This routine consists of an atomic decrement (`mpadd -1`) of the synchronization cell, the synchronization loop (labeled with `SYNCLOOP` and an atomic reincrement (`mpadd +1`) with a postsynchronization phase. The synchronization loop causes each processor to wait until the synchronization cell `gps[1]` reaches a zero value. Due to the sequential memory consistency, each processor will eventually observe this event within the next four clock cycles, and exit the loop. The SB-PRAM offers a globally visible status register bit, the so-called modulo flag, that toggles in each machine cycle (it is the least significant bit of the global cycle counter). Because the processors are modulo-aligned at the entry to the routine, they leave the synchronization loop in two wave fronts separated by two clock cycles. In the postsynchronization phase, the early-leaving processors are delayed by two cycles (the two `nop` instructions); hence, at exit of the routine, all processors are exactly synchronous.

5.1.9 Implementation of the Simple Lock

The Fork implementation of the `simple_lock` data type and the routines

```
typedef int simple_lock, *SimpleLock;
SimpleLock new_SimpleLock( void );
simple_lock_init( SimpleLock );
simple_lockup( SimpleLock );
simple_unlock( SimpleLock );
```

has been described in Section 4.2.8. In the Fork library, the `simple_lockup()` routine is implemented in SB-PRAM assembler,[3] and the other two are just macros defined in `fork.h`. The space required for a `SimpleLock` instance is only one memory word, the time overhead for locking and unlocking is only a few machine cycles.

A `simple_lock` sequentializes access to a critical section but it is not fair, that is, it does not guarantee that processors get access in the order of their arrival at the `simple_lockup` operation.

5.1.10 Implementation of the Fair Lock

A *fair lock* sequentializes accesses while maintaining the order of arrival times of the processors at the `lockup` operation. Our implementation of fair locks adapted from Röhrig's thesis [Röh96] works similar to an automatic booking-office management system (see Figure 5.7). Each customer who wants to access the booking office must first get a *waiting ticket* from a (shared) ticket automaton. After having served a customer, the booking clerk increments a globally visible counter indicating the *currently active ticket index*. Only the customer whose waiting ticket index equals this counter value will be served next. The others have to wait until their ticket index becomes active.

In Fork we model this booking-office mechanism by two shared counters: one for the ticket automaton, and one for the ready-to-serve counter.

```
typedef struct {
  unsigned int ticket;   /* the ticket automaton */
  unsigned int active;   /* the currently active ticket */
} fair_lock, *FairLock;
```

[3]See the assembler source file `forklib2.asm`.

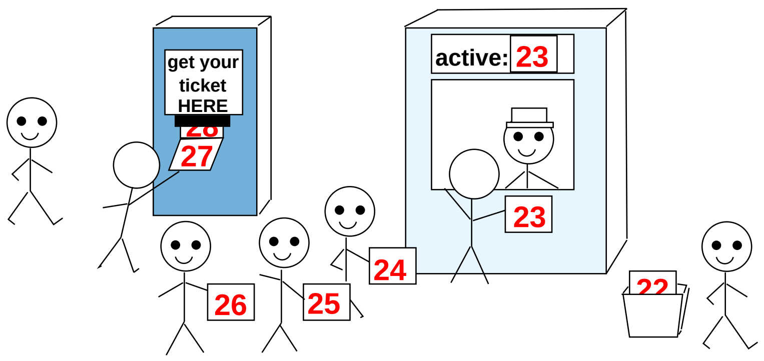


FIGURE 5.7: The analogy of a booking-office management for the fair lock implementation.

Both counters have to be accessed by atomic *fetch&increment* operations. They are initialized to zero. For an allocated fair lock, this is done by the following routine:

```
void fair_lock_init( fair_lock *pfl )
{
  pfl->ticket = pfl->active = 0;
}
```

A new `FairLock` instance can be created and initialized by the following constructor function:

```
FairLock new_FairLock( void )
{
  pfl = (FairLock) shmalloc( sizeof( fair_lock ));
  pfl->ticket = pfl->active = 0;
  return pfl;
}
```

The `fair_lockup` operation gets a ticket `t` and waits until `t` becomes active:

```
void fair_lockup( fair_lock *pfl )
{
  pr int t = mpadd( &(pfl->ticket), 1 );    // get a ticket t
  while (t != pfl->active)  ;  // wait until t becomes active
}
```

The `fair_unlock` operation increments the `active` counter:

```
void fair_unlock( fair_lock *pfl )
{
  mpadd( &(pfl->active), 1 );  // atomically increment counter
}
```

The Fork library contains assembler implementations[4] of these Fork routines. The assembler implementation guarantees that read and `mpadd` accesses to the active counter are scheduled to different cycles by alignment to different values of the `MODULO` flag. The space required to store a fair lock is 2 words, twice the space of a simple lock. The runtime required by the `fair_lockup` and the `fair_unlock` operation is practically the same as the time taken by `simple_lockup` resp. `simple_unlock`.

Finally, we should think about handling overflow of the counters. After 2^{32} calls to `fair_lockup`, the size of the unsigned integer `ticket` is exceeded. On the SB-PRAM, this is not a problem, because `mpadd` computes correctly modulo 2^{32} [Röh96]. Only if the total number of processes exceeded 2^{32}, explicit range checking would be required, but such a large PRAM machine is very unlikely to exist in the foreseeable future.

5.1.11 Implementation of the Reader–Writer Lock

For the *reader–writer lock* (see Section 4.2.8), access to a critical section is classified into read and write access. While a *writer* may modify a shared resource, a *reader* inspects it but leaves it unchanged. The implementation of a reader–writer lock must guarantee that either at most one writer and no reader, or no writer and any number of readers are within the critical section.

The implementation of the reader–writer lock in Fork follows the description by Wilson and Röhrig [Wil88, Röh96] and the implementation by Röhrig for the SB-PRAM [Röh96].

The reader–writer lock data structure contains an integer variable `readercounter` that acts more or less as a simple lock, and a fair lock `writerlock` that coordinates the writers. This `rw_lock` data type is declared in `fork.h`.

```
typedef struct {             //reader-writer-locks
  unsigned int readercounter;//bits 0-29 used as reader counter
                             //bit 30 is used as writer flag
  fair_lock writerlock;      //current and next number
} rw_lock, *RWLock;
```

The `readercounter` holds the number $r \in \{0, ..., p\}$ of readers that are currently within the critical section, and the `writerlock` is a fair lock used to sequentialize the writers with respect to the readers, and the writers with respect of each other. Also, bit 30 of the `readercounter` is used as a semaphore $w \in \{0, 1\}$, in order to avoid that readers and a writer that find a zero value in `readercounter` simultaneously would both enter the obviously "empty" critical section. Storing the reader counter r and the writer counter w in the same memory word alleviates the implementation considerably, as they can be manipulated simultaneously and atomically by the available atomic operators of Fork. The constructor function

```
RWLock new_RWLock( void );
```

[4]See the library assembler source file `lib/forklib2.asm`.

allocates space for a reader–writer lock l by `shmalloc()` and initializes l by setting the `readercounter` to zero and initializing the fair lock by

`fair_lock_init(&(`l`->writerlock));`

For the `lockup` and `unlock` operations, the mode of access desired by a processor is passed in a flag valued 0 for read and 1 for write:

```
#define RW_READ 0
#define RW_WRITE 1
```

The operation `int rw_lockup( rw_lock *l, int mode )` works as follows. If the processor is a reader (i.e., `mode` is `RW_READ`), two cases may occur: (1) if there is a writer in the critical section (i.e., the writer flag is set), it must wait until the flag is cleared; or (2) if the critical section is free (i.e., the writer flag is cleared), it tries to increment the `readercounter` and checks at the same time whether, in the meantime, a “fast” writer happened to succeed in setting the writer flag again. If that happened, the `readercounter` must be decremented again, and then the processor tries again from the beginning, to leave the critical section and thus to unlock the `writerlock`, and then compete again for entering the critical section.

If the processor is a writer (i.e., `mode` is `RW_WRITE`), it must wait until all writers that arrived before it and all readers have left the critical section. Then it atomically sets the writer flag by a `syncor` operation. As `sync`*op* and `mp`*op* instructions are scheduled by the compiler automatically to different cycles (i.e., to different values of the `MODULO` flag), setting and clearing of the writer flag is separated properly.

Here is the Fork code:

```
#define __RW_FLAG__ 0x40000000  /* 2^30 */

void rw_lockup( RWLock l, int mode )
{
 int waiting;
 if(mode == RW_READ) {     /* I am a reader */
    waiting = 1;
    while(waiting) {
       waiting = 0;
       // wait for other writers to leave:
       while( mpadd(&(l->readercounter), 0) & __RW_FLAG__ ) ;
       // writers finished - now try to catch the lock:
       if( mpadd(&(l->readercounter), 1) & __RW_FLAG__ ) {
          // another writer was faster than me
          mpadd( &(l->readercounter), -1); /* undo */
          waiting = 1; /* not done, try again */
       }
    }
 }
 else {      /* I am a writer */
```

```
    // wait for other writers to leave:
    fair_lockup(&(l->writerlock));
    // set flag to tell readers that I want to have the lock
    syncor( (int *)&(l->readercounter), __RW_FLAG__ );
    // wait for readers to leave:
    while( mpadd( &(l->readercounter), 0) & (__RW_FLAG__-1)) ;
 }
}
```

This mechanism is fair with respect to the writers, but if there are many write accesses, the readers may suffer from starvation as the writers are prioritized by this implementation. For this reason, this reader–writer lock implementation is called a *priority reader–writer lock*.[5] In order to "rank down" the writers' advantage, the corresponding `rw_unlock` function offers a tuning parameter `wait` specifying a delay time:

```
int rw_unlock( rw_lock *l, int mode, int wait )
{
 if (mode == RW_READ) {    /* I am a reader */
    mpadd ( &(l->readercounter), -1);
 }
 else {     /* I am a writer */
    int i;
    /* free lock for readers: */
    mpadd ( &(l->readercounter), -__RW_FLAG__);
    /* delay loop, give readers a better chance to enter: */
    for(i = 0; i < wait; i++) ;
    /* free for writers: */
    fair_unlock(&(l->writerlock));
 }
}
```

Hence, the degree of fairness of the implementation is controlled by the writers.

5.1.12 Implementation of the Reader–Writer–Deletor Lock

The data type `RWDLock` is declared in `fork.h`, and the operations defined on it (see Section 4.2.8) are implemented in the Fork standard library (`lib/async.c`).

```
#define RW_DELETE 2
```

The most important extension to the reader–writer lock implementation is an additional semaphore, the delete flag, that indicates whether a deletor has access to the lock. Initially the delete flag is set to zero. The `rwd_lockup` operation checks this flag during the waiting

[5]Wilson sketches an alternative implementation of a reader–writer lock prioritizing the readers over the writers [Wil88]. Röhrig describes a more involved *completely fair* implementation of a reader–writer lock [Röh96].

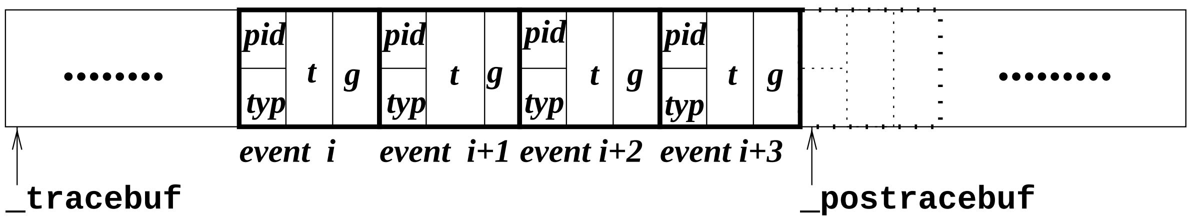


FIGURE 5.8: Organization of the event records and the trace buffer. Due to the constant-time atomic *fetch&add* operation `mpadd` of the SB-PRAM, an arbitrary number of subsequent event records can be allocated to the requesting processors in the same machine cycle.

loop. As soon as it is set to one, all `rwd_lockup` operations return zero to indicate failure. Otherwise, they return a nonzero value that indicates that the lock has been acquired normally.

The interested reader may have a look at `/lib/async.c` to see the complete implementation.

5.1.13 Implementation of the `trv` Trace Visualization Tool

Trace events that occur within a user-defined time interval are recorded in a central *trace buffer* that can later be written to a file and processed further. In contrast to almost any other parallel tracing tool, we do not use processor-individual trace buffers in the local memories. This design choice was motivated by three reasons: (1) On the SB-PRAM, storing and accessing trace event records consecutively in shared memory takes the same time as for a private memory area. (2) The output routine that flushes the trace buffer to a single file can be parallelized. (3) There is no need for a merge of up to 2048 local trace files afterwards.

An entry for a trace event in the trace buffer consists of an event type index, a (physical) processor ID, a time stamp, and a group identifier. A unique group identifier g is derived from the address of the group's synchronization cell in the shared stack. In the current implementation, each event entry requires 3 words in the trace buffer (see Figure 5.8), as the event type index and the processor ID can share one word without incurring additional time overhead. The total number of events usually depends linearly on the number of processors.

In order to activate the tracing feature, the user program must be compiled and linked with the `-T` option of the Fork compiler. If tracing is not desired, the program is thus linked with a variant of the standard library without tracing, hence there will be no loss of efficiency at all. Otherwise, the program is instrumented with additional code that writes event records into the trace buffer at the time t at which they occur (see Figure 5.8). The code is written in SB-PRAM assembler for performance reasons, because it is called quite frequently and must be executed very fast to reduce distortions of the behaviour of the computation to be traced.

`startTracing` starts the recording of events. A default trace buffer size is used unless the `startTracing` call is preceded by an `initTracing(`L`)` call that reallocates the central trace buffer on the permanent shared heap with L memory words.

The actual overhead of writing a trace event record is 17 machine cycles, as can be seen from Figure 5.9. Outside the dynamically defined time interval where tracing is enabled, the overhead is only 5 machine cycles. The code exploits the nonsequentializing multiprefix-

```
#ifdef PROFILING
// skip entries outside startTracing/stopTracing interval:
gethi    __tracingenabled,r31
ldgn     r31,(__tracingenabled)&0x1fff,r31
add      r31,0,r31     //compare
beq      forklib_barr_entry_trace_done
// issue a TRACE_BARRIER_ENTRY=3 trace entry
gethi    __postracebuf,r30
add      r30,(__postracebuf)&0x1fff,r30
getlo    3,r31         // size of an event record: 3 words
mpadd    r30,0,r31     // r31 <- mpadd( &_postracebuf, 3);
getlo    (3<<16),r30   // event type
getct    par1
stg      par1,r31,1    // r31[1] <- time
gethi    ___PROC_NR__,par1
ldg      par1,___PROC_NR__&0x1fff,par1
stg      gps,r31,2     // r31[2] <- gps
add      r30,par1,r30  // r30 <- (3<<16) + __PROC_NR__
stg      r30,r31,0     // r31[0] <- ""
forklib_barr_entry_trace_done:
#endif
```

FIGURE 5.9: SB-PRAM assembler code for writing an event record at the entry of the barrier routine. Excerpted from the Fork standard library. Note that addressing of the SB-PRAM memory is in terms of entire 32-bit words.

add instruction `mpadd` as an atomic *fetch&add* operator (see Figure 5.8); the shared pointer `_postracebuf` serves as a semaphore that indicates the position of the next free record in the trace buffer. Hence, events are stored in linear order of their occurrence; simultaneous events are recorded simultaneously in subsequent records in the trace buffer.

Eight event types (from 0 to 7) are predefined for entry and exit of subgroups, for entry to a group splitting operation, for entry and exit of barriers and lock acquire operations. User-defined events are recorded by calling the `traceEntry` routine with an integer parameter indicating a user-defined event type between 8 and 31. Phases started by such a user event will be drawn in the corresponding color taken from the standard color table of `xfig` unless the color is specified explicitly by the user.

After calling the synchronous `stopTracing` function, events are no longer recorded in the trace buffer. The events stored in the trace buffer can be written to a file (*trace file*) by calling the synchronous `writeTraceFile` function, which takes the trace file name and an optional title string as parameters. Formatting the data of the trace buffer is done fully in parallel; then, a single `write` call is sufficient to flush the formatted data to a single file in the host's file system, where it can be accessed by `trv`. It is important that the—albeit parallelized—somewhat time-consuming part of reformatting and file output is performed offline, that is, outside the `startTracing`/`stopTracing` interval.

Tracing shared memory accesses adds an overhead of about 8 clock cycles to shared memory accesses (load from and write to shared memory, `mpadd`/`syncadd`, `mpmax`/`syncmax`, `mpand`/`syncand`, and `mpor`/`syncor`). When writing the trace buffer, the values of these counters are also written to the trace file. Hence, the corresponding counters of all processors are later just added up to obtain the global access statistics. Tracing an access to shared memory means incrementing a local counter corresponding to that access type (read, write, variants of multiprefix and atomic update). In any case, the execution time of programs that are not compiled and linked with `-T` is not affected by the tracing feature.

`trv` converts the sequence of trace events in the trace file to a time-processor diagram in `xfig` format. `trv` is very fast. It uses a simple plane-sweep technique to process the trace events, which relies on the fact that the event records in the trace file are sorted by time. Then, it is sufficient to keep for each processor the type, group, and time of its event record processed last. As soon as `trv` encounters a new trace event for a processor, a rectangle for the now finished phase is added in the time bar for that processor. Finally, the last-activity rectangles of all processors are closed up to the right hand side border of the time-space diagram. The overall processing time is thus linear in the length of the trace file. 10000 events are processed in less than 1.5 seconds on a SUN SPARC10 workstation.

Current and future work on `trv` When tracing a long-running program or a large number of processors, the screen resolution is not sufficient to display all details. Nevertheless, the generated FIG file contains them (vector graphics). One possibility is to use a display tool with scroll bars, maybe with an explicit time axis to be added for better orientation. Nevertheless this results in a limited overview of the entire parallel trace.

An alternative is to apply a fish-eye view transformation with a rectangular focus polygon [FK95]. Inside the focus polygon, the diagram is shown in full detail for a small time and processor axis interval. Outside the focus polygon, the scaling of the axes continuously decreases, ending up the smallest resolution at the image boundaries.

Moreover, it is not always necessary that all involved processors participate in tracing during the `startTracing`...`stopTracing` interval. Instead, a subset of the processors to participate could be selected, for instance, by specifying a bit mask parameter to the `startTracing` call that is or-ed with the processor ID. When recording an event, the nonparticipating processors will then be delayed by the same number of machine cycles that the participating processors need for the recording, in order to maintain synchronicity where necessary and to reduce distortion effects on race conditions caused by the program's instrumentation.

The verbose display of the shared memory access statistics could be changed into a graphical representation (with bar or pie diagrams).

A recent extension of `trv` in a student project at the University of Trier added events for message passing and drawing of arcs for messages sent by the core routines of the MPI message passing library, which have been implemented in Fork in another student project.

The development of an interactive programmer interface for `trv`, integrated in a GUI-based compilation and program execution manager, has recently been started as a student project at the University of Trier.

5.2 Compiling Fork for Other Parallel Architectures

It appears that the compilation of Fork to efficient parallel code depends on several preconditions:

- *The target machine is a MIMD architecture.* Compiling Fork to SIMD architectures would lead to sequentialization[6] of concurrent groups that could operate in parallel on a MIMD architecture. The compilation of asynchronous program regions to a SIMD architecture implies simulation of a virtual MIMD machine on a SIMD machine, which would result in complete sequentialization and prohibitively high overhead.

- *User-coded barrier synchronization for a subset of the processors is not expensive.* Most commercial parallel platforms offer a system-specific routine for global barrier synchronization that is often (but not always) considerably faster than a user-coded variant with semaphores, such as by providing a separate network that is dedicated to synchronization. In the presence of group splitting, global barriers can be used only at the top level of the group hierarchy. Also, the implementation of a barrier on current massively parallel machines usually takes time at least logarithmic in the number of synchronizing processors. while it takes constant time on the SB-PRAM. Furthermore, we will see in Section 5.2.2 that even for a machine with powerful, nonsequentializing `atomic_add` or `fetch&add` instructions, like `syncadd` or `mpadd` on the SB-PRAM, barrier synchronization takes more time and space if exact synchronicity of the parallel machine at the instruction level is not supported. Also, the replacement of barriers by suitable padding with `nop`s, as discussed above for the SB-PRAM, is not possible when compiling for asynchronous architectures.

- *The language supports just as many processors as are available by the target architecture.* A general emulation of additional processors in PRAM mode by the compiler and its runtime system involves high overhead on any type of architecture, as we will see in Section 5.2.1, and hence is not supported in Fork. In contrast, this task is much simpler for SIMD or dataparallel languages, where the emulation of additional processing elements can be done statically by the compiler. For most MIMD languages where processors follow individual control flow paths, an emulation of virtual processors in software requires another layer of interpretation. Even if the multithreading capabilities of modern operating systems are exploited, the need for a sequentially consistent shared memory and groupwide barriers requires many context switches; the support of a synchronous execution mode requires many group locks. We will further elaborate on this issue in Section 5.2.1.

- *The target architecture offers a sequentially consistent shared memory.* An emulation of a shared memory on top of a distributed memory in software (i.e., a virtual or distributed shared memory implementation by the compiler's runtime system) is often inefficient, especially for irregular applications, if the memory consistency model is not relaxed

[6]In some cases, the concept of the maximal synchronous ancestor group introduced in Section 5.2.1 may help to exploit SIMD parallelism beyond the current leaf group.

to a weaker form (see *Practical PRAM Programming* [B1, Chap. 4] for a survey of hardware-based DSM systems and, e.g., the article by Protic et al. [PTM96] for a survey of software DSM systems). An emulation of a shared memory on a distributed memory multiprocessor by the compiler only is again possible only for SIMD or dataparallel languages like HPF, which we will discuss further in Section 5.2.3.

- *Private memory subspaces are embedded into the shared memory.* This feature of Fork enables keeping declaration and usage of pointers simple: dereferencing of any pointer is done by a `ldg` instruction. On the SB-PRAM, this simplification comes for free and is even enforced by the architecture, as there are no private processor memories of sufficient size available at all. Even for the small local memory modules of the SB-PRAM processors, which are used by the operating system only (mainly as hard disk buffers), the memory access time is in the same order of magnitude (one machine cycle) as for an access to the global, shared memory (two machine cycles). On other parallel architectures, exploiting locality of memory accesses is much more critical for the overall performance. If the private memory subspace is stored in a separate, processor-local memory module, different memory access instructions must be used depending on whether the address is private or shared. This can either be determined statically (by another sharity declaration of the pointee of a pointer and appropriate type checking rules for the compiler, as applied, e.g., in C^*), or dynamically (by inspecting at runtime the value of a pointer before dereferencing it, as applied in ForkLight).

- *Synchronous execution mode guarantees exact synchronicity and* (*in principle*) *the separation of reading and writing memory accesses.* The entire SB-PRAM architecture is synchronous at the instruction level. Hence processors will remain exactly synchronous once they have been synchronized, as long as they make the same branch decisions at conditional jumps (apart from asynchronous interrupts, which are switched off by the SB-PRAM operating system when executing a Fork user program). Also, on the SB-PRAM, the effect of memory accesses is deterministic in synchronous computations, as synchronous execution implies a well-defined total order among the memory accesses. This feature is unique to the SB-PRAM; exactly synchronous execution is not supported by any other MIMD machine. Obviously, exact synchronicity is necessary only at shared memory accesses; the local computation parts between shared memory accesses need not run synchronously at all. More specifically, synchronous execution is necessary only among accesses to the *same* shared memory location. Efficient compilation thus requires that the instruction/processor pairs accessing the same location can be statically identified at sufficient precision by data dependence analysis. Hence, when compiling Fork to an asynchronous shared memory architecture, additional synchronization will be required to put the accesses to shared memory locations into the right order. Even if exact synchronicity at the memory access level is somehow established, read and write accesses must be properly separated to guarantee the correctness and determinism of the computation. On the SB-PRAM this is achieved by alignment to the global `modulo` flag. On asynchronous architectures, this separation must be simulated by additional synchronization. Note that this problem also appears in asynchronous regions.

- *Concurrent write is allowed and deterministic.* This is another feature inherited from the SB-PRAM. Obviously, a deterministic concurrent write resolution policy like Priority CRCW makes sense only in a synchronous framework. Hence, no other MIMD architecture directly supports a comparable concurrent write conflict resolution mechanism, simply because there *is* no other synchronous MIMD architecture. Instead, accesses to the same memory location would have to be sequentialized on an asynchronous parallel machine, and then the values read and stored would generally depend on the relative speed of the processors and of the corresponding memory access request messages in the network. Hence, in addition to the sequentialization of concurrent write accesses, further synchronization code would be necessary to guarantee the deterministic order implied by the Priority CRCW policy. This problem also includes multiprefix operators in synchronous regions.

We see that the compilation of the Fork language in its present form will produce inefficient code for any other parallel hardware, if static program analysis does not succeed in providing sufficient information for optimizing the additional synchronization code being inserted.

On the other hand, if we relax some of the points listed above in the language definition, we will generally improve the efficiency of code compiled for other parallel architectures, but the language will no longer be a PRAM language. By such relaxations, the two non-PRAM variants of Fork, namely ForkLight for the Asynchronous PRAM model and NestStep for the BSP model, have been created.

5.2.1 Emulating Additional PRAM Processors in Software

The SB-PRAM does not support an efficient emulation of additional Fork processors (apart from its unused logical processor emulation feature that is quite limited in power), for good reasons, as we will see in this section: In the general case, emulating more SB-PRAM processors applied to the synchronous execution mode of Fork would either compromise the sequential memory consistency or cause tremendous inefficiency. Many of the problems and solution strategies discussed here will occur again in a similar form when considering the compilation of Fork for asynchronous shared memory machines in Section 5.2.2. First, we introduce some terminology.

A ***cross-processor data dependence*** denotes a relationship between a reading and a writing memory access instruction or two writing memory access instructions that are executed on different processors, such that these (may) access the same shared memory location. The cross-processor data dependence is directed from either (1) a write access to a read access that possibly reads the value written by the former, or (2) a read access to a write access that possibly overwrites the value read by the former, or (3) a write access to a write access that possibly overwrites the value written by the former. For instance, in

```
start {  sh int x;
         int y = $;
 S1:     x = y;
 S2:     y = x + 2;
```

```
    }
```

the read access to x in S2 is cross-processor data dependent on the write access in S1, and the write access to x in S1 is cross-processor-dependent on itself. For an introduction to the general concept of data dependency we refer to our summary in [B1, Sect. 7.1] and to the standard literature on parallelizing compilers [Ban93, Ban94, Ban97, Wol94, ZC90]. For now it is sufficient to see that the original execution order of cross-processor-dependent instructions must be preserved when emulating additional processors, in order to obtain the same program behavior; otherwise, there would be race conditions among different processors for accesses to the same shared memory location.

By ***virtualization*** we denote the process of emulating a set of PRAM processors, so-called *logical* processors, on a smaller PRAM, in software. Note that this includes the emulation of a PRAM by a single processor machine as a special case of an emulating PRAM. In an asynchronous environment, the logical processors are often referred to as *threads* or *lightweight processes*; in the synchronous PRAM environment we prefer to call them *processors*, as their number will be a runtime constant and the set of all logical processors should behave like a monolithic large PRAM for the programmer.

In simple cases, the virtualization may be done as a program transformation by the compiler only. This requires that the control flow for each processor be statically known, as for SIMD code, and that sufficient information about cross-processor data dependencies in the parallel program be available.

Another possibility for virtualization is to use the operating system or the runtime system of the compiler for simulating several logical processors on each of the *available processors* in a round-robbin manner, where a compile-time or runtime scheduler decides about how to distribute the CPU time of the emulating processor across the emulated processors (***task scheduling***). Scheduling can be ***nonpreemptive***, which means that a logical processor keeps the CPU until it releases the CPU explicitly by a cswitch operation that causes a context switch to the logical processor to be emulated next. Or it can be ***preemptive***, which means that a *runtime scheduler* (also called *dispatcher*) assigns CPU time to the emulated processors in a time-slicing manner. If a time slice is over, the scheduler interrupts execution of the current logical processor, issues a context switch, and the next logical processor is emulated during the following time slice. A ***context switch*** consists in saving the current status of the current logical processor (program counter, status register, register contents) in the main memory and restoring the latest status of the logical processor to be simulated next. In order to access the proper set of private variables, the emulating processor's BASE register must be set accordingly to the start address of the next logical processor's private memory subspace. This is (more or less) the method used in pramsim, where a context switch is performed after every assembler instruction.

Let us now consider some examples for the case that a larger number of Fork processors is to be emulated on top of a smaller SB-PRAM machine. First we look at the simpler cases: *SIMD code*, which denotes synchronous program regions executed by the root group only, without and with cross-processor data dependencies. These cases could be handled quite efficiently by the compiler. Then, we consider asynchronous program regions, and finally the general form of synchronous execution by multiple groups that may access group-globally declared shared variables. In these cases, simulation must be done by using explicit context switches, which requires support by the runtime system of the compiler and incurs considerable overhead if the semantics of the original Fork program is to be preserved.

Virtualization for SIMD code

For the following Fork program fragment

```
start   // $ numbered from 0 to __STARTED_PROCS__-1
   a[$] = 2 * b[$-1];
```

where we assume that the shared arrays a and b are disjoint, an arbitrary number of __STARTED_PROCS__ logical SB-PRAM processors could be easily emulated by the compiler on top of a SB-PRAM with only __AVAIL_PROCS__ physically available processors, as control flow is the same for each processor (SIMD) and there are no cross-processor data dependencies here. If the physical processor IDs, called __PPID__ on the smaller machine, are numbered from 0 to __AVAIL_PROCS__ minus 1, the following loop, running on a SB-PRAM with __AVAIL_PROCS__ processors, produces the same result:

```
start
  for ($=__PPID__; $<__STARTED_PROCS__; $+=__AVAIL_PROCS__)
    a[$] = 2 * b[$-1];
```

Moreover, each occurrence of a private variable, such as i, is simply replaced by a reference, say, i[__PROC_NR__], to a suitably defined shared array on the smaller SB-PRAM machine.

Virtualization for irregular SIMD code

Emulation by the compiler becomes more difficult if there are cross-processor data dependencies, in particular if these depend on runtime data. For instance, in

```
start
  a[x[$]] = a[$];
```

with a shared array a and a shared integer array x, a cross-processor data dependence among the executions of the assignment must be assumed that require that no element of a is overwritten before it has been read by all processors. Note that the compiler operates on the intermediate representation of the program, where the assignment statement has already been split into an equivalent sequence of low-level instructions:

```
Reg1 ⟵ $;
Reg2 ⟵ x[Reg1];
Reg3 ⟵ a[Reg1];
a[Reg2] ⟵ Reg3;
```

As the contents of array x is not known at compile time, a barrier and an explicit temporary array is required to obtain the same behavior on the smaller SB-PRAM machine. However, the following straightforward solution is *not* correct

```
for ($=__PPID__; $<__STARTED_PROCS__; $+=__AVAIL_PROCS__) {
    Reg1 ⟵ $;
    Reg2 ⟵ x[Reg1];
    Reg3 ⟵ a[Reg1];
    call _barrier;
    a[Reg2] ⟵ Reg3;
}
```

as the barrier would separate only those read and write accesses to a that are executed for the first __AVAIL_PROCS__ logical processors. Already the logical processor with $=__AVAIL_PROC__ may access an overwritten element of a, which contradicts the semantics of the original Fork program.

Instead, the compiler must generate intermediate code for the smaller SB-PRAM machine as follows, using a temporary array temp:

```
start {
for ($=__PPID__; $<__STARTED_PROCS__; $+=__AVAIL_PROCS__) {
    Reg1 ⟵ $;
    Reg3 ⟵ a[Reg1];
    temp[$] ⟵ Reg3;
}  // implicit barrier
for ($=__PPID__; $<__STARTED_PROCS__; $+=__AVAIL_PROCS__) {
    Reg1 ⟵ $;
    Reg2 ⟵ x[Reg1];
    a[Reg2] ⟵ temp[$];
}
}
```

Virtualization for asynchronous regions

Larger problems with emulating more processors occur where control flow is individual (MIMD) and not statically known, which is the common case for Fork programs (except for parts of synchronous regions where processors are known to be working in the root group only). In such cases, the compiler cannot use loops for the virtualization.

For example, consider the following program fragment, which was taken from an implementation of the CRCW Quicksort algorithm [CV91] described in Chapter 1 of *Practical PRAM Programming* [B1]:

```
farm {
  if (IAmALeaf[$])
```

```
    done[$] = 1;
  else {
    if (lchild[$]<N)
      while (! done[lchild[$]]) ;
    if (rchild[$]<N)
      while (! done[rchild[$]]) ;
    a[$] = a[lchild[$]] + a[rchild[$]];
    done[$] = 1;
  }
}
```

For the compiler it is no longer possible to determine statically which processor will follow which control flow path, as the branch decisions depend on runtime data. Hence, it cannot know where to put loops or barriers to obtain the same behavior with virtualization.

One solution would be to use a nonpreemptive scheduler in the operating system or run-time system of the compiler. In this case, each logical processor is responsible itself for releasing the CPU by an explicit context switch, for example, if it must wait at a barrier. Note that, if a context switch is forgotten, it may happen that processors are waiting for an event that will be simulated later, and thus get caught in a deadlock (because they do not reach the next `cswitch` statement that would enable the event-triggering processor to be simulated). This problem would not be there without virtualization. The alternative solution is preemptive scheduling. This avoids the abovementioned problem, as the control over context switching is held by the operating system only. On the other hand, this may lead to more context switches than are really necessary. Both constructions incur considerable overhead due to context switching and thread management. On the SB-PRAM we can assume that a context switch always takes the same number of CPU cycles, and thus does not affect synchronous execution adversely.

The asynchronous mode of Fork makes no assumptions on the relative speed of processors, and hence cross-processor data dependence is not to be handled by the compiler but by the programmer. Nevertheless, with nonpreemptive scheduling it is not sufficient to have a single context switch just at the end of a `farm` body, for instance, as processors may wait for each other, as in the example above, and the other logical processors simulated in an earlier time slice must be given the chance to find the updated value in a written shared memory location. Hence, as a conservative solution, a context switch is to be executed after each write access and before each read access to the shared memory. Note that the implementation of a barrier or of a lock acquire operation will accordingly contain context switches as well, such that all logical processors have the chance to see the zero in the synchronization cell before it is reincremented again.

Virtualization for synchronous MIMD code

In synchronous mode, however, we are faced with the problem of maintaining (at least) memory-level synchronicity. For synchronous code where control flow is irregular or cross-processor data dependencies cannot be resolved at compile time, because of problems such as aliasing by pointers, the compiler must, in the worst case, conservatively assume that each pair of shared memory access instructions where at least one write access is involved may cause a cross-processor data dependency, which must be protected by synchronization and a context switch.

For example, consider the following synchronous program fragment:

```
start {
 sh int x = 3, z = 4;
 pr int y;
 ...
 if ($<2)   z = x;
 else       y = z;
}
```

Note that we have here a cross-processor data dependence, caused by the store to the shared variable `z` and the load from `z`, which are executed by different processors in different groups that are asynchronous with respect to each other. A read or write access to a shared variable that is declared global to the group performing the access is called an ***asynchronous access*** to that variable [Käp92, Wel92, Lil93]. In this example, the two branches of the `if` statement contain asynchronous occurrences of `z`, while the access in the previous initialization is synchronous.

By the semantics of the synchronous mode in Fork, all emulated logical processors executing the `else` part must read the *same* value of `z`. Because of the simulation on a smaller machine, not all emulated processors can execute the load of `x` resp. `z` or the store to `z` simultaneously. Rather, some of these that evaluate the condition to zero might complete the `else` branch before the `if` branch has been executed by any processor, and some might arrive after. These processors would store a different value for `y`, thus compromising the semantics of the synchronous mode of execution.

In order to maintain the semantics of the synchronous mode in Fork, it would be required to keep a group lock for each shared variable that may eventually be accessed by a proper subgroup of the group declaring it. Because of the presence of pointers and weak typing in C, it is hardly possible to determine these variables statically. Hence, a lock would be required for each shared memory cell! Clearly, this waste of shared memory space can be considerably reduced by keeping only a small number n of such locks, stored in an array `SMlock` of simple locks, and hashing the memory accesses across the n locks, where n should be a power of 2 for fast modulo computation. But this is to be paid by sequentialization of independent accesses to different shared memory locations that happen to be hashed to the same lock. Also, the overhead of locking and unlocking (see Section 4.2.8) is not negligible even if efficient hardware support such as the powerful `mpadd` instruction is used extensively. In the example above, we would obtain the following pseudocode:

```
start  // compiler initializes logical processor
       // emulation by the runtime system
  if ($<10) {
     seq {
        simple_lockup( SMlock+z%n );
     } // implicit barrier, contains context switch
```

```
      z = x;
      seq {
         simple_unlock( SMlock+z%n );
      } // implicit barrier, contains context switch
   }
   else {
      seq {
         simple_lockup( SMlock+z%n );
      } // implicit barrier, contains context switch
      y = z;
      seq {
         simple_unlock( SMlock+z%n );
      } // implicit barrier, contains context switch
   }
   // implicit barrier - contains context switch
```

This means that execution for the subgroup that acquires the lock first will be simulated completely, while the other processors wait and just repeatedly switch context in their `simple_lockup` call. Only when the first subgroup is finished and waits at the implicit barrier after the `if` statement (which also contains context switches) does the second subgroup succeed in acquiring the lock. The overhead caused by the locking, unlocking, the context switches, and the sequentialization of the two subgroups would lead to considerable inefficiency.

As we would like to avoid this inefficiency, Fork renounces an option of emulating additional processors; instead the programmer is responsible for writing more flexible parallel code that works with any number of available processors.

An optimization for the locking of asynchronous accesses to shared variables on PRAM architectures has been proposed for the old FORK standard [HSS92]. At any time t of the execution of a Fork program, the ***maximal synchronous ancestor group*** $ms(g,t)$ of the current group g denotes the least-depth ancestor group $\bar{g}$ of g in the group hierarchy tree for which all groups in the subtree rooted at $\bar{g}$ still work synchronously at time t. For instance, the `fork` instruction does not immediately lead to asynchrony of its subgroups; rather, these proceed synchronously up to the execution of the first branch instruction where the control flow of the different subgroups may diverge. A combination of static and runtime program analysis [Lil93] can compute for each instruction a safe underestimation ms of the possible values for $ms(g,t)$ for all its executions. The value of ms may change at branching points of control flow. For an asynchronous access to a shared variable `z` defined by an ancestor group g' of the current group, group locking of the access to `z` is not necessary if all groups that may access `z` concurrently (in the worst case, g' and all its successor groups) are located within the group hierarchy subtree rooted at $ms(g,t)$, that is, if $ms(g,t)$ is equal to g' or to an ancestor group of g'.

Note that the concept of the maximal synchronous ancestor group may, at least in principle, also be used to save synchronization cells, as all successor groups g' of $ms(g,t)$ could use the same synchronization cell. Nevertheless, this hardly improves the performance of barrier synchronization on asynchronous MIMD architectures; rather, on the contrary, the barrier overhead will generally grow with an increasing number of processors on non-SB-PRAM architectures. On the other hand, analysis of the maximal synchronous ancestor group may lead to more efficient code when compiling Fork programs to SIMD architectures (like vector processors, SIMD processor arrays, VLIW architectures, or EPIC processors), as several active groups could share a SIMD instruction and thus reduce the effect of serialization of concurrent groups on SIMD machines. On the other hand, when compiling for asynchronous (and non-combining) MIMD architectures, the concept of the maximal synchronous ancestor group is not helpful at all; on the contrary, the artificially enlarged groups would incur more overhead of barrier synchronization than barrier-synchronizing each leaf group independently, because, on such architectures, barrier synchronization does no longer perform in constant time. On the SB-PRAM, where barrier synchronization takes constant time, the concept of the maximal synchronous ancestor group implies no substantial advantage (beyond optimizing asynchronous accesses), but determining the maximal synchronous ancestor group requires some space and runtime overhead. Hence, it is not used in the prototype compiler for the SB-PRAM.

5.2.2 Compiling for Asynchronous Shared Memory Architectures

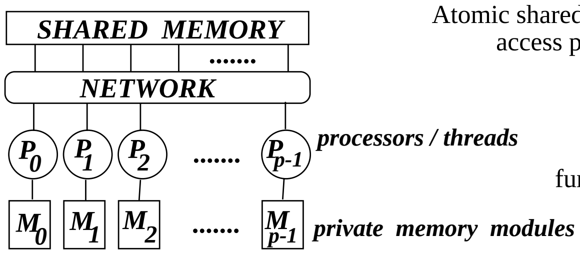
Now we leave the SB-PRAM as a compilation target and consider instead an asynchronous parallel machine with a sequentially consistent shared memory, also known as an *Asynchronous PRAM* [Gib89, CZ89]. Such a machine with an abstract shared memory access interface is summarized in Figure 5.10. An implementation of the necessary functions for controlling concurrent execution and accessing the shared memory, as defined in Figure 5.10, for an existing shared memory machine or a portable interface like P4 or OpenMP is straightforward [C16].

Hatcher and Quinn provided a detailed presentation of the issues in compiling dataparallel programs for such an asynchronous shared memory machine [HQ91, Chap. 4], describing a compiler for Dataparallel C to a Sequent multiprocessor. Note that in Fork the activity region of each group defines a SIMD computation, except for asynchronous accesses to shared variables. Hence, the code generation scheme for SIMD or dataparallel code must be extended for Fork by groupwide barriers and groupwide sharing of variables. This implementation of the group concept is nearly identical to the mechanism described in Section 5.1, with one exception: The groupwide barrier requires major modifications if the target architecture is asynchronous, which we will describe next. Then, we describe the emulation of a synchronous computation on an asynchronous machine. Finally, we discuss the necessary extensions for a target machine supporting only a weaker form of memory consistency.

Implementation of a groupwide barrier

As in the compilation for the SB-PRAM described in Section 5.1, a compiler for an Asynchronous PRAM, such as the ForkLight compiler, keeps for each group a shared group frame and, on each processor of the group, a private group frames.

The *shared group frame* (see Figure 5.11) is allocated in the group's shared memory. The shared group frame contains the shared variables local to this group and (in contrast to a single cell as before) *three synchronization cells* $sc[0]$, $sc[1]$, $sc[2]$. Again, each processor holds a



Atomic shared memory access primitives:

`SMwrite`	(write a value into a shared memory cell)
`SMread`	(read a value from a shared memory cell)
`atomic_add`	(add an integer to a cell)
`fetch_add`	(add an integer to a cell and return its previous value)

furthermore:

`shmalloc`	(shared memory allocation)
`beginparallelsection`	(spawn p threads)
`endparallelsection`	(kill spawned threads)

FIGURE 5.10: Asynchronous PRAM model with shared memory access operations. The seven operations on the right side are sufficient to handle shared memory parallelism as generated by the compiler. The processors' native load and store operations are used for accessing private memory. Other functions, such as inspecting the processor ID or the number of processors, are implemented by the compiler's runtime library.

register `gps` pointing to its current group's shared group frame, and additionally a private counter `csc` indexing the *current synchronization cell*, hence it holds one of the indices 0, 1, or 2. `csc` may be stored in the private group frame. When a new group is created, `csc` is initialized to 0, $sc[0]$ is initialized to the total number of processors in the new group, and $sc[1]$ and $sc[2]$ are initialized to 0. If no processor of the group is currently at a barrier synchronization point, the current synchronization cell $sc[\texttt{csc}]$ contains just the number of processors in this group.

At a groupwide barrier synchronization, each processor atomically increments the next synchronization cell by 1, then atomically decrements the current synchronization cell by 1, and waits until it sees a zero in the current synchronization cell; see Figure 5.11. The algorithm guarantees that all processors have reached the barrier when a zero appears in the current synchronization cell. Only then they are allowed to proceed. At this point of time, though, the next current synchronization cell, $sc[R_{\text{next}}]$, already contains the total number of processors, i.e. is properly initialized for the following barrier synchronization. Once $sc[R_{\text{csc}}]$ is 0, all processors of the group are guaranteed to see this, as this value remains unchanged at least until after the following synchronization point.

The execution time of a barrier synchronization is, for most shared memory systems, dom-

```
_async_barrier:
$R_{\text{csc}} \leftarrow$csc;
$R_{\text{next}} \leftarrow R_{\text{csc}} + 1$;
**if** $(R_{\text{next}} > 2)$  $R_{\text{next}} \leftarrow 0$   // *wrap-around*
atomic_add( gps+$R_{\text{next}}$, 1);
atomic_add( gps+$R_{\text{csc}}$, -1);
**while** (SMread($sc[R_{\text{csc}}]$)$\neq 0$) ;   // *wait*
csc$\leftarrow R_{\text{next}}$;
```

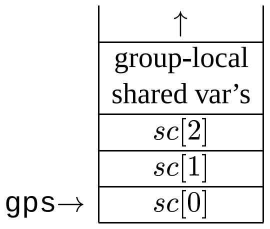


FIGURE 5.11: Barrier synchronization pseudocode and shared group frame for asynchronous shared memory architectures.

inated by the groupwide `atomic_add` and `SMread` accesses to shared memory, while all other operations are local. If the hardware does not support combining of `atomic_add` or `fetch_add` shared memory accesses, execution of these will be sequentialized, and hence a barrier will take time linear in the number of processors in the group. For that case, there are other barrier algorithms than this one based on shared counters. For instance, tree-based algorithms using a tree of semaphores or message passing along a spanning tree of the processors of the group lead (in principle) to logarithmic time complexity of a barrier execution. Furthermore, various methods exploiting special hardware features of particular parallel machines have been proposed. A survey and practical evaluation of barrier algorithms for shared and distributed memory architectures is available [HS96].

Emulating synchronous execution

In the asynchronous program regions of a Fork program, all potentially present cross-processor data dependencies must be handled explicitly by the programmer. Hence, nothing has to be changed there when compiling for an Asynchronous PRAM.

If a synchronous region is to be compiled for an Asynchronous PRAM, additional synchronization must be inserted. In principle, synchronization is required between any two subsequent assignment statements. In cases where a memory location addressed by the left side of an assignment may be accessed concurrently also on its right side, a temporary variable must be used, and a synchronization must occur even within the assignment statement. For instance, in

```
start {
     sh int x[N], y[N];
S1:  x[$+1] = 2 * x[$];
S2:  y[$] = x[$+1] + 1;
}
```

there are cross-processor data dependences from `S1` to `S1` and from `S1` to `S2`, due to accesses to the same elements of `x`. Hence, barrier synchronizations must be placed as follows:

```
start {
      sh int x[N], y[N], temp[N];
S1a:  temp[$] = 2 * x[$];
S1b:  _barrier;
S1c:  x[$+1] = 2 * temp[$+1];
S1d:  _barrier;
S2:   y[$] = x[$+1] + 1;
}
```

As in the case of logical processor emulation, there are several methods that differ in their efficiency and in their applicability, depending on how precise static information on the cross-processor data dependencies is available.

If the control flow is statically known, as in SIMD or dataparallel code, a simple graph-oriented method [SO97] can be used to cut all cross-processor dependences in the data dependency graph with a minimum number of barriers. Related methods have been described

[PDB93, PH95]. The ***cross-processor data dependence graph*** G_c is a directed graph that consists of the (parallel) assignment statements as nodes, and the cross-processor data dependences as edges. Although polynomial in time [Gav72], generating parallel target code from a dataparallel program with a *minimal* number of barriers is an expensive computation, as a minimum cut of G_c must be determined. Linear-time heuristics for dataparallel code have been proposed [QHS91, SO97]. The greedy algorithm used by Quinn et al. [QHS91] applies a sequence of iterative steps to the cross-processor data dependence graph G_c. In each step, the algorithm determines the dependency edge with earliest target node. Before this target node, a barrier is inserted, and all edges cut by this barrier are removed from G_c before applying the next step to the reduced G_c. Stöhr and O'Boyle [SO97] propose a similar heuristic called "fast first sink" and prove that it generates a minimum number of barriers for straight-line code. Obviously, the more precise the data dependency analysis is, the less edges will be in G_c, which usually results in less barriers being generated. In some cases, even runtime analysis of the dependency structure [MSS$^+$88, KMR90] may be profitable, namely, if this runtime analysis can be reused several times, such as for iterative computations in a dataparallel program. Generally, restructuring of the program may help to reduce the number of barriers needed [PH95]. For instance, where the data dependencies permit, statements may be reordered to allow more dependence edges to share barriers. Conversely, Jeremiassen and Eggers [JE94] describe a static program analysis that detects sharing of barriers by independent threads of computation in a coarse-grained parallel program. An improvement of the Hatcher–Quinn barrier generation mechanism [HQ91] has been proposed for loops [UH95]. An optimization of the Hatcher–Quinn code generation technique [HQ91] for cache-based DSM systems has also been given [Kla94].

If the processors at the source and the sink of a cross-processor data dependency can be statically determined, the global barrier can be replaced by simpler ***bilateral synchronization***, with semaphores or message passing [PDB93, Tse95, KHJ98]. Bilateral synchronization may be more efficient than a global barrier if the memory access patterns involved can be statically analyzed and match one of a few common situations like nearest-neighbor communication, reduction, or broadcast. Also, fuzzy barriers [Gup89] can help to avoid idle times at barriers.

Shared memory consistency issues

The PRAM model, and thus also Fork, assume a sequentially consistent shared memory. In the Fork compiler for the SB-PRAM, this was guaranteed because (1) the SB-PRAM has no caches, hence there is only one copy of each shared memory location in the main memory; and (2) the compiler does not hold the value of a shared variable in registers after a store instruction; hence the copies in the processor-local registers will always have the same value as the memory location itself.

Some commercial shared memory parallel machines like Tera MTA inherently support sequential memory consistency by hardware mechanisms. On the other hand, many distributed-shared memory systems are cache-based and rely on a weaker form of shared memory consistency. Nevertheless, these architectures also provide some additional means for establishing sequential memory consistency.

One such mechanism is explicitly enforcing sequential consistency for a given program point by a memory barrier instruction, such as the `flush` directive in OpenMP [Ope97], which flushes pending shared memory write requests, such that subsequent memory accesses will always find the most recent value. The scope of the `flush` directive may also be limited to a given list of special shared memory objects. Another possibility provided by some programming languages (like NestStep) or user interfaces (like OpenMP) is to declare sequential consistency by default for special shared program variables, such as for all lock objects (OpenMP) or for all shared variables declared as `volatile` (OpenMP, NestStep).

Hence, when compiling Fork to a nonsequentially consistent parallel machine, either a directive `flush(`a`)` must be inserted after each write access to a shared memory location a, or the entire shared memory must be declared as a `volatile` shared array and addressed by suitable array accesses, if this is supported by the target environment.

Again, in special situations it is possible to eliminate some of the `flush` directives if the existence of corresponding cross-processor data dependences can be disproved by static program analysis. For instance, if the value written is guaranteed to be read only by the writing processor itself, the corresponding memory barrier is not necessary.

5.2.3 Compiling for Distributed Memory Architectures

There are several ways to compile Fork to a distributed memory architecture. The simplest one is to use a software DSM emulation, which reduces the problem to compiling for an asynchronous shared memory architecture.

Another possibility is to use a message passing interface that supports ***one-sided communication***, also known as ***direct remote memory access*** (DRMA), as in MPI-2 [MPI97] or BSPlib [HMS$^+$98]. One-sided communication means that the recipient of a message needs not to execute an explicit receive instruction in order to write the arrived data to a well-defined memory location. Instead, the sender decides where the data will be stored in the receiver's local memory, and the receiver is not explicitly informed[7] about this. In that case, the shared memory can be hashed over the local memories of the processors, and a write access results in a one-sided send command to the processor owning the shared variable. Read accesses request the data from the owning processor and block until the requested data have arrived. This scheme guarantees sequential memory consistency, as there is only one version of each shared memory cell. Note that this constitutes a very simple implementation of a distributed shared memory, and thus again reduces the problem to compiling for asynchronous shared memory architectures. Processor idle times due to waiting for the result of a read request can be avoided by multithreading, where several logical processors share an available processor and are controlled by a scheduler (see Section 5.2.1). After issuing a read request, a context switch is performed. When the requesting processor is executed again, the requested value may already have arrived.

For SIMD or dataparallel code, in particular if generated by parallelizing compilers, the

[7]A possible implementation of one-sided communication may spawn a second thread on each processor that listens for incoming messages all the time. Whenever a message arrives, it receives it and (depending on the desired action stated in the message) writes the data to an address specified in the message or sends the requested contents of a memory location back to the sender. Of course, also `fetch_add` or `atomic_add` requests can be served in this way. A generalization of this method is known as *Active Messages* [CGSvE92].

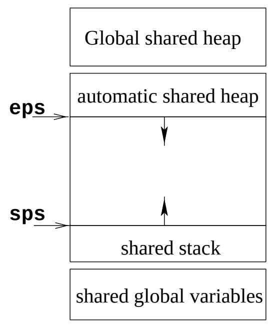


FIGURE 5.12: Organization of the shared memory.

compiler can directly generate two-sided communication [CK88, ZBG88, RP89, KMR90, LC90, ZC90, HKT91b, HKT91a]. Data, in particular large arrays, are distributed across the processors. The choice of this data distribution is critical for the performance of the generated code and is hence left to the programmer, such as in the form of HPF directives. Usually, the data distribution implies the distribution of computation; the owner of the memory location addressed by the left side of an assignment is responsible for evaluating the right-side expression (owner-computes rule). If the processors involved in a cross-processor data dependence are statically known, as is the case for simple array indexing schemes, the corresponding send and receive instructions are generated directly. A receive instruction waits until the corresponding message has arrived; hence the message passing mechanism can be used at the same time for bilateral synchronization. If the source and the sink of a cross-processor data dependence cannot be determined statically, runtime techniques like the inspector–executor technique [MSS$^+$88, KMR90] may be used.

Prefetching is a compile-time technique to partially hide the latency of a read access to a remote memory location. The read access is split into two phases. In the first phase, a load request is sent to the remote processor owning the desired value. This send operation must be scheduled to the earliest point where it is still guaranteed that the most recent value will be returned, namely, immediately after that preceding barrier or interprocessor communication that guarantees the consistency of the requested value. Furthermore, a receive buffer must be available from that point on. The second phase is a receive operation that waits until the remote processor has sent the requested value. If data dependency analysis unveils that enough statements can be scheduled between the first and the second phase without compromising the semantics of the program, the latency of the access can thus be padded with useful work.

5.3 Compiling ForkLight for an Asynchronous PRAM

The ForkLight implementation uses its own shared memory management to implement the hierarchical group concept, in a very similar way as described in Section 5.1.2 for the compilation of Fork for the SB-PRAM, using a sufficiently large slice of shared memory. To the bottom of this shared memory slice we map the shared global initialized resp. noninitialized variables. In the remainder of this shared memory part we arrange a shared stack and an automatic shared heap, again pointed to by the shared stack pointer `sps` and the automatic shared heap pointer `eps`, respectively (see Figure 5.12). Group splitting operations cause splitting of the remaining shared stack space, creating an own shared stack and automatic heap for each subgroup, resulting in a "cactus" stack and heap). Another shared memory slice is allocated to install the global shared heap. A private stack and heap are maintained in each processor's private memory by the native C compiler.

Initially, the processor on which the user has started the program executes the startup code, initializes the shared memory, and activates the other processors as requested by the user. All these processors start execution of the program in asynchronous mode by calling `main()`.

As for the compilation of Fork, the ForkLight compiler keeps for each group a shared and a private group frame, where the shared group frame (see Figure 5.11), pointed to by the pointer variable `gps`, is allocated on the group's shared stack. As usual, the shared group frame contains the shared variables local to this group. In contrast to Fork, there are *three synchronization cells* $sc[0]$, $sc[1]$, $sc[2]$ instead of a single one, as discussed in Section 5.2.2, where we described the algorithm for a groupwide barrier synchronization on an asynchronous PRAM. Also, for ForkLight, the pointer to the parent group's shared group frame is stored on the private group frame: shared memory accesses are much more expensive than private memory accesses, and thus all information that needs not necessarily be stored in a shared memory location should be kept in the local memory. Moreover, the private group frame holds a private counter `csc` indexing the *current synchronization cell*. When a new group is created, `csc` is initialized to 0, $sc[0]$ is initialized to the total number of processors in the new group, and $sc[1]$ and $sc[2]$ are initialized to 0. If no processor of the group is currently at a barrier synchronization point, the current synchronization cell $sc[\texttt{csc}]$ contains just the number of processors in this group. Furthermore, the private group frame, pointed to by a pointer `gpp`, contains a reference to that group's shared group frame and to the parent group's private group frame. It also contains fields for the group index `@`, the processor ID `$`, and the processor rank `$$`. `@` needs not be stored on the shared group frame since it is read-only. Finally, the private group frame also holds the pointers `eps` and `sps`.

Note that many parallel machines offer hardware support for *global* barrier synchronization, which is usually more efficient than our software mechanism with explicit handling of semaphores. Thus, where the group is statically known to contain all started processors, the global barrier routine can be called instead.

Note that in the Fork compiler, some constructs (the loops) do not require a private group frame as there is only one child group and thus no need for redefining `$`. Nevertheless, code generation becomes simpler if all subgroup-creating constructs build private group frames in the same way. In particular, this simplifies the generation of code for `return`, `break`, and `continue`. In the ForkLight compiler, the additional overhead of constructing private group frames also for loops is marginal, as it involves only local operations.

5.3.1 Translation of a Function Call

Asynchronous functions are just compiled as known from sequential programming, as no care has to be taken for synchronicity.

A control-synchronous function with shared local variables needs to allocate a shared group frame. As these variables should be accessed only after all processors have entered the function, there is an implicit group-wide barrier at entry to a control-synchronous function.

5.3.2 Translation of the `fork` Statement

A straightforward implementation of the `fork` statement in ForkLight assumes that all k subgroups will exist and distributes shared stack space equally[8] among these. For

```
fork ( $k$; @=$e$ ) <stmt>
```

the following code is generated:

```
(1) $R_k \leftarrow eval(k)$;  $R_@ \leftarrow eval(e)$;  $slice \leftarrow \lfloor$(eps-sps)$/R_k\rfloor$;
(2) if $(0 \le R_@ < R_k)$
      { sc $\leftarrow$ sps$+R_@ * slice$;
       SMwrite( sc, 0 ); }
(3) barrier local to the (parent) group                  // necessary to guarantee a zero in sc[0]
(4) if $(0 \le R_@ < R_k)$ {
       $R_\$ \leftarrow$ fetch_add(sc,1);
       allocate a private group frame pfr
       and store there gps, eps, sps, gpp;
       initialize the new csc field to 0, @ field to $R_@$, $ field to $R_\$$
       if $(R_\$ = 0)$
       { sc[1]$\leftarrow$ 0; sc[2]$\leftarrow$ 0; }
     }
(5) barrier local to the (parent) group                  // guarantees final subgroup sizes in sc[0]
(6) if $(0 \le R_@ < R_k)$                  // enter subgroup $R_@$                otherwise skip
      { gps$\leftarrow$ sc; gpp$\leftarrow$ pfr; sps$\leftarrow$ gps+3+#sh.locals; eps$\leftarrow$ gps$+slice$; }   else goto (9)
(7) code for <stmt>
(8) atomic_add( gps+csc, -1);                    // cancel membership in the subgroup
     leave the subgroup by restoring gps, sps, eps, gpp from the private group frame
(9) (next statement)
```

The overhead of the above implementation mainly consists of the parallel time for two groupwide barriers, one subgroupwide concurrent `SMwrite`, and one subgroupwide `fetch_add` operation. Also, there are two exclusive `SMwrite` accesses to shared memory locations. The few private operations can be ignored, since their cost is usually much lower than shared memory accesses.

A first optimization addresses the fact that group splitting and barriers can be skipped if the current group consists of only one processor. In that case, also group-wide barrier synchronizations can be skipped. A private status flag `ONE` (stored in the private group frame) keeps track of this property; it is set to 1 when at entry to a subgroup the group size reaches 1, and 0 otherwise. The number of skipped subgroup constructions may just be handled in a

[8]The shared stack memory may be distributed among the groups also in the ratio of subgroup sizes. While this proportional splitting method seems, on the average, to be the better choice with respect to memory fragmentation compared to the uniform splitting of the shared stack, it may nevertheless be disadvantageous if the subgroups turn out to have equal memory requirements independent of the number of processors executing them. A possible solution may be to install a compiler option that causes the compiler to generate code for proportional splitting if desired by the user.

counter `ignframes` stored in the private group frame. Initially set to zero when allocating that frame, the counter is incremented when a new subgroup should have been entered, and decremented when it is left. [9]

A second optimization exploits the observation that some of the subgroups may be empty. In that case, space fragmentation can be reduced by splitting the parent group's stack space in only that many parts as there are different values of $R_{@}$. The additional cost is an `atomic_incr` operation to a zero-initialized scratch cell executed by the leader ($R_{\$} == 0$) of each new subgroup, another barrier on the parent group, and a `SMread` operation. Static program analysis may help to avoid this dynamic test in some situations. For instance, if the exact space requirements of one subgroup were statically known, all remaining space could be left to the other subgroups. Unfortunately, the presence of nonexact synchronicity, pointers and weak typing makes such analysis quite difficult.

Third, not all group splitting operations require the full generality of the `fork` construct. Splitting into equally (or weighted) sized subgroups, as provided, for instance, in PCP [BGW92] and NestStep, can be implemented with only one barrier and without the `fetch_add` call, as the new subgroup sizes and ranks can be computed directly from locally available information. The extension of ForkLight by corresponding customized variants of the `fork` statement to ForkLight is straightforward.

5.3.3 Accessing Local Shared Variables

For ForkLight, we slightly modify the method used in the Fork compiler for accessing group-local shared variables: The necessary pointer chasing operations to access a shared group frame in the statically visible part of the group hierarchy tree are now performed as local memory accesses, as these pointers are stored in the private group frames. Hence, there is only one, expensive shared memory access involved.

5.3.4 Optimization of Barriers

Implicit barriers are generated by the compiler before shared data is allocated, for instance when entering a `csync` function with shared local variables. This guarantees that space for them on the shared stack can be safely reused. In addition, the programmer may add explicit barrier statements where considered necessary.

In order to incur minimum overhead due to synchronization, the total number of `barriers` generated must be minimized. Minimization of group-wide barriers in the same group activity region can be done using the framework discussed in Section 5.2.2. Moreover, if two `fork` statements are immediately nested (i.e. there is no branch of control flow between their headers) such that there is no shared memory access between their exit points

```
fork( ... ) {
   ...  /*no branch of control flow*/
   fork(...) {
      ...
```

[9]Note that the value of the group index must be stacked if it changes for a subgroup.

```
      }
      ... /*no shared memory access*/
   }
```

then the barrier at the end of the inner `fork` can be eliminated, since the barrier for the parent group is already sufficient to avoid reuse of the shared group frames.

5.3.5 Virtual Processing

Up to now we required the number of processors (or threads) p executing the program to be a runtime constant limited to the hardware resources, that is, each of the P hardware processors executes exactly one process. Now we discuss what has to be done if $p > P$ threads should be generated. In this case, each of the p physical processors could simulate $k = \lceil p/P \rceil$ threads, in a round-robbin fashion.

We have seen in Section 5.2.1 that a virtual processor emulation in software is impossible to realize for the *fully* synchronous language Fork without incurring prohibitive overhead in time and space. But the relaxed synchronicity of ForkLight permits a straightforward implementation.

Context switching is often supported by the native language programming environment and/or the processor hardware. The remaining question is where to insert context switches in the generated C program.

If no detailed program dependence analysis is made, it is conservatively safe if a context switch is inserted

- before each read access to shared memory (i.e., `SMread` and `fetch_add`).
- after each write access to shared memory (i.e., `SMwrite`, `atomic_add` and `fetch-_add`)

because each emulated thread must have the possibility to see (and react to) the modification of a memory location by another thread. Context switching before read accesses can be combined with prefetching to hide the delay of a remote read access.

This implies that for a barrier statement at least three context switches would be executed. An optimization is also possible here, similar to the optimizations of barriers discussed in the previous subsection. For instance, if there occurs no access to shared memory between two subsequent context switches, one of these switches can be eliminated. This means for the barrier implementation that the context switch to be inserted immediately after the `atomic-_incr` can be omitted.

More optimizations are possible if cross-processor data dependencies are computed. Context switches between subsequent read accesses to the same shared memory location are not necessary if no write access to that location by another processor may occur in between, and subsequent accesses that are guaranteed to access different shared memory locations need no context switch in between.

5.3.6 Translation to OpenMP

OpenMP [Ope97] is a shared memory parallel application programming interface for Fortran77 and C/C++, consisting of a set of compiler directives and several run time library functions. The C/C++ API had just been released after the prototype implementation of the ForkLight compiler was finished (end of 1998), although no implementation of the C/C++ API was available even at that time (only implementations of the Fortran77 API).

A transcription of the existing ForkLight back-end to OpenMP is straightforward: `beginparallelsection()` and `endparallelsection()` correspond to a `omp parallel` directive at the top level of the program. The shared stack and heap are simulated by two large arrays declared `shared volatile` at this directive, `SMread` and `SMwrite` become accesses of these arrays. Explicit `flush` directives after `SMwrite`s are not necessary for `volatile` shared variables. `omp barrier` and other synchronization primitives of OpenMP cannot be used for ForkLight because they are not applicable to nested SPMD computations; thus we will use our own implementation for the synchronization routines. OpenMP also offers support for hardware-supplied `atomic_inc` and `atomic_dec` instructions by the `atomic` directive which is applicable to increment and decrement operators, but nevertheless `fetch_add` has to be expressed using the sequentializing `critical` directive. We propose the following simple optimization to increase scalability:

Hashing of critical section addresses The `omp critical` directive optionally takes a compile-time constant name (string) as a parameter. `critical` sections with different names may be executed concurrently, while entry to all `critical` sections with the same name is guarded by the same mutual exclusion lock. For our software implementation of a specific atomic memory operation (e.g., `fetch_add`) we generate c copies, where $c \geq 1$ is a power of 2. Each copy t, $0 \leq t < c$, is guarded by a `critical` directive parameterized by the binary representation of t. Furthermore we select $l = \log c$ bit positions $i_1, ..., i_l$ and define a hash function h for addresses x by $h(x) = x_{i_1} x_{i_2} ... x_{i_l}$, where x_j denotes the jth bit of address x. Consequently, `fetch_add` accesses to addresses x_1 and x_2 are not sequentialized if $h(x_1) \neq h(x_2)$. The choice of c and of i_1,...,i_l is a performance tuning operator.

5.3.7 Performance Results

We have implemented the compiler for two parallel platforms that have been supported by P4: multiprocessor Solaris workstations and the SB-PRAM. As we shall see, these two completely different types of architecture represent two quite extremal points in the spectrum of shared memory architectures regarding execution of P4 / ForkLight programs.

On a loaded four-processor SUN workstation running Solaris 2.5.1, where atomic memory access is sequentialized, we observed good speedup for well-parallelizable problems like pi calculation or matrix multiplication but only modest or no speedup for problems that require frequent synchronization:

Pi-calculation (stochastic method, $N = 5000000$):

#processors	1	2	3	4
time [s]	22.61	14.55	11.37	7.86

matrix–matrix multiplication, 200×200 integers:

#processors	1	2	3	4
time [s]	41.08	24.16	19.22	14.49

parallel mergesort for 48000 integers using a handwritten routine for sequential sorting:

#processors	1	2	4
time [ms]	4390	3581	2325

parallel quicksort for 120000 integers using the host's optimized `qsort()` routine for sequential sorting:

#processorss	1	2	4
time [ms]	4447	4656	4417

parallel quicksort for 120000 integers using a handwritten quicksort routine for sequential sorting:

#processorss	1	2	4
time [ms]	14498	9834	6350

On the SB-PRAM we obtained an important improvement by exploiting its native fetch-add (`mpadd`) and atomic-add (`syncadd`) operators which do not lead to sequentialization, in comparison to standard P4 which does not efficiently support atomic fetch-add or atomic-add.

For the 128 PE prototype[10] of the SB-PRAM (1998) at Saarbrücken running the SB-PRAM operating system PRAMOS, we obtained the following performance results:

parallel mergesort on 1000 integers using the optimized sequential `qsort()` function:

#processors	1	2	4	8	16	32	64
time [ms]	573	373	232	142	88	57	39

parallel mergesort on 10000 integers using a hand-written sequential quicksort routine:

#processors	1	2	4	8	16	32	64
time [ms]	2892	4693	3865	1571	896	509	290

parallel quicksort on 1000 integers using a handwritten sequential quicksort routine:

#processors	1	2	4	8	16	32	64
time [ms]	1519	807	454	259	172	145	130

These figures allow the following interpretations:

- Efficient support for nonsequentializing atomic `fetch_add` and `atomic_add`, as in the SB-PRAM or Tera MTA, is essential when running ForkLight programs with large numbers of processors. (ForkLight) executables relying only on pure P4 suffer from serialization and locking/unlocking overhead and are thus not scalable to large numbers of processors.

- On a nondedicated, loaded multiuser / multitasking machine like our Solaris multiprocessor workstation, parallel speedup suffers from poor load balancing due to stochastic

[10] Due to some operating system restrictions, only up to 124 PE's can be used for the application program, the other ones (one PE per processor board) are reserved as I/O servers by PRAMOS.

delaying effects: the processors are unsymmetrically delayed by other users' processes, and at barriers these delays accumulate.

- Even when running several P4 processes on a single processor, performance could be much better for $p > 1$ if the ForkLight run time system had complete control over context switching for its own processors. Otherwise, much time is lost spinning on barriers to fill the time slice assigned by an OS scheduler that is unaware of the parallel application. This is an obvious weakness of P4.
- Explicit load balancing in an SPMD application may be problematic in particular for small machine sizes (quicksort), or when the hardware scheduler does not follow the intentions of the user, for instance when the scheduler maps several P4 processes to a processor where only one process was intended for.
- Where these requirements are met, our prototype implementation achieves acceptable speedups and performance scales quite well even for rather small problem sizes.

5.4 Compiling NestStep to a Distributed Memory System

In this section we describe an inexpensive implementation of NestStep on top of a workstation cluster with a uniform message passing interface like MPI or Java Sockets. NestStep programs are translated by a pre-compiler to ordinary source programs in the basis language. These, in turn, are compiled as usual by the basis language compiler and linked with the NestStep runtime library.

5.4.1 Group Objects and Group Splitting

The runtime system keeps on each processor a pointer `thisgroup` to a `Group` object that describes the current group of that processor. The `Group` object contains (see also Table 4.6) the size of the group, its initial size at group creation, the rank of that processor within the group, its initial rank at group creation, the group index, the depth in the group hierarchy tree, a superstep counter, and a pointer to the `Group` object for the parent group. Furthermore, there are fields that are used internally by the runtime system when splitting the group, such as counter arrays. There are also pointers to a list of shared variables declared by that group, to a list of references to variables that are to be combined in the next combine phase, and to a list of references to variables that are to be combined in the final combine phase of that group. Finally, there is a list of hidden organizational shared variables that are used for communicating distributed shared array sections, which we will discuss in Section 5.4.9.

Dynamic group splitting, in its full generality of data-dependent subgroup selection, requires in principle a prefix computation to determine the future rank of each processor in its desired subgroup, and the future subgroup sizes. This prefix computation may be done either in parallel or in sequential. The necessary communication required by a parallel prefix computation can, especially for small and medium group sizes, easily outweigh the advantage of a parallel prefix computation. Instead, the implementation applies, by default, a replicated *sequential* prefix computation, where each processor evaluates the subgroup selector expression of *all* processors in the group. Provided that these expressions are not too complicated, the sequential version is usually faster because it is communication-free. Another advantage of this scheme is that each processor can, at practically no additional expense, store the absolute IDs of all processors belonging to its group in its current `Group` object; this information is needed anyway for point-to-point communication of distributed array sections.[11] Depending on the underlying network and the size of the group being split, the runtime system can select the parallel or the sequential variant.

As in the sequential variant, each processor evaluates the subgroup index of every processor in the group. NestStep-C requires in the second parameter of the most general `neststep` statement not just an integer-valued expression to determine the subgroup index, but a pointer to a function that takes an integer parameter (the rank) and returns an integer, namely the new subgroup index. This function should behave in the same way on each processor. For instance, it should call the pseudorandom number generator only if that is guaranteed to produce the same sequence of pseudorandom numbers on each participating processor. Also, the function should not access variables that may contain different values on different processors.

The static group splitting that is applied for the simpler `neststep` variants, such as `neststep(`k`)`, is communication-free anyway.

5.4.2 Naming Schemes for Addressing Shared Variables

If the different processors, such as the Java Virtual Machines (JVMs) for NestStep-Java, may use different object file formats or load classes in different order, the same (shared) class or object variable could have different relative addresses on different processors. In that case, a system-wide unique, symbolic reference must be passed with the update value in the combine or access messages.

In the NestStep-Java prototype implementation we used the full name string of the shared variable, prefixed by the declaring group's `path()` string, and applied hashing to avoid excessive string handling. This scheme also allows to distinguish between different instances of a local shared variable in the different calls of a recursive function executed by different groups, as in the `quicksort` example in Figure 4.42.

In the NestStep-C prototype implementation, unique names were coded as a `Name` structure consisting of four integer components:

- the *procedure name code*, which is 0 for global variables and a positive value for procedures, computed by the frontend. The procedure code must be globally unique, account-

[11] Indeed, this is just what is done at subgroup creation by the communicator and processor group concept of MPI. Nevertheless, for the NestStep-C runtime system, the communicator concept of MPI is *not* used, in order to avoid excessive overhead of subgroup creation. Instead, all communication is relative to the global communicator `MPI_COMM_WORLD`; each `Group` object holds a dynamically allocated array `pids` that contains the global indices of the processors belonging to that group, indexed by their (initial) group-local rank `rankinit`. As these arrays have to be computed anyway in the prefix computation discussed above, there is no need to recompute these in the corresponding MPI group functions. In particular, the customized routines implementing the variants of the `neststep` statement maintain the following invariant

```
thisgroup->pids[ thisgroup->rankinit ] == PID
```

where `PID` holds the global processor rank (within the MPI system) of the current processor.

ing for the case of separate compilation of multiple source files. Negative procedure names are used for internal shared variables of the runtime system.

- the *group name code*. It must be dynamically unique with respect to all predecessors and all siblings of the declaring group in the group hierarchy tree. Such a naming scheme can be obtained by the following straightforward enumeration approach: The group code for the root group is 0; the ith child of a group receives the parent's group code plus $i + 1$, where children are counted starting at 0.
- a *relative name*. This is a positive integer which is given by the frontend to distinguish between different shared variables in the same activity region. Negative relative names are used internally by the runtime system.
- an *offset value*, which is used to address individual array elements and structure fields. It is -1 for scalar variables and positive for array elements and structure fields. For multidimensional arrays, the index space is flattened according to the relevant basis language policy.

Hence, equality of variables can be checked fast and easily by three or at most four integer comparisons.

5.4.3 Values and Array Objects

Values are internally represented by `Value` objects. A `Value` can be an integer or floating-point value, or a pointer to an `Array` or `DArray` object.

`Array` objects represent replicated shared arrays. An `Array` object contains the number of elements, the element `Type`, and a pointer to the first array element in the processor's local copy.

`DArray` objects represent distributed arrays. A `DArray` object contains the number of elements, the element `Type`, the distribution type, the size of the owned array section, the global index of the first owned array element, and an `Array` object containing the owned array elements.

5.4.4 Shared Variables

Shared variables can be declared and allocated for each group activity region. They are accessed via `ShVar` objects, which are wrapper data structures for the local copies of program variables declared as shared. A `ShVar` object contains the `Name` entry identifying the variable, its `Type`, and its `Value`. The `Group` object associated with each group holds a list of `ShVar` objects containing the shared variables it has declared. Global shared variables are stored in a separate global list.

Lists of shared variables are represented by `ShVars` objects. There are methods for inserting, retrieving, and deleting `ShVar` objects from such lists. The implementation uses unsorted, dynamically allocated arrays, because in most programs the lists of shared variables are usually quite short.

Searching for a shared variable starts in the `ShVars` list of shared variables local to the current group. If the group name code in the `Name` being searched for does not match the current group, the search method recurses to the parent group, following the path upward in the group hierarchy tree until the static scope of visibility of local variables is left (this scope is computed by the frontend). Finally, the `ShVars` list of global shared variables is searched.

5.4.5 Combine Items

A `CombineItem` object is a wrapper data structure that represents either a combine request with a value contributed to the groupwide combine phase at the end of a superstep or a commit request from another processor returning a combined value; additionally, combine items may also contain requests for remote distributed array sections and replies to such requests. These will be discussed in Section 5.4.9.

Combine items are designed for traveling across processor boundaries. Hence, a combine item must not contain processor-local pointers to values but the values themselves. A `CombineItem` object thus consists of the `Name` identifying the shared variable to be combined, the `Type`, a `Value` (which may include entire `Array` objects), the name of the contributing processor, and an integer characterizing the binary combine function that determines the combine policy. Combine items have an optional second `Name` entry that refers to the identifier of a private variable k, which is, for technical reasons, nevertheless registered as a `ShVar` object as well. This entry indicates the private target variable of a prefix computation for prefix combine items, and the private target array section for combine items that represent read requests to remote distributed array sections.

Combine items participating in the combine phase for the same superstep are collected in a combine list that is pointed to from the current `Group` object. A `CombineList` object has a similar structure as a `ShVars` variable list, that is, it supports dynamic insertion, retrieval, and deletion of `CombineItem`s, with the exception that the combine items in a combine list are sorted by `Name` in lexicographic order.

5.4.6 Serialization of Combine Lists

Combine lists can be posted to other processors in the groupwide combine phase. For this purpose, all relevant objects (i.e., `Value`, `Array`, `DArray`, `Name`, `CombineItem` and `CombineList` objects) have serialization routines that allow to pack them into a dynamically allocated byte array. They also offer `size` routines that allow to estimate the prospective space requirement for the byte array. In Java, such routines come automatically with each `Serializable` object; in C they have been written by hand and are considerably faster.

A message is represented by a `Msg` object, which is just a dynamically allocated buffer typed `void *`. It contains the abovementioned byte array, prefixed by the byte array length, the message tag (i.e., whether it is a combine/commit message or a distributed array read/update request or reply) and the number of serialized combine items. `Msg` objects can be easily shipped across processor boundaries, for instance by using MPI routines or Java socket communication, and are deserialized to `CombineList`s on the receiver side. For this purpose, all serializable objects provide a deserialization routine as well.

5.4.7 Combine Trees

For each group g, a static *combine tree* is embedded into the given processor network. The necessary entries are collected in a `Tree` object that is pointed to by the `Group` object for the current group on each processor. The combine tree may be any kind of spanning tree, such as a d-ary tree of height $\lfloor \log_d g.\texttt{size()} \rfloor + 1$ or a binomial tree of height $\lceil \log_d g.\texttt{size()} \rceil$. The d-ary tree is better suited where processing and committing of the combine lists in the nodes is the performance bottleneck, as the length of the longest communication path is minimized. In contrast, the binomial tree is better suited where the (sequentialized) receiving of combine lists (for reduction and gather communications along the tree) or forwarding (for broadcast and scatter communications) is the time-critical part. The degree d may be chosen individually for each group depending on the group size and expected average message length [12]. In both cases the processors are linked such that the initial group-relative processor ranks g`.rankinit` correspond to a preorder traversal of the tree, see Figure 5.13. Hence, a combine tree fulfills (1) the heap property (i.e., increasing ranks along any path from the root to a leaf) and (2) that for any inner node, all ranks in its left subtree are smaller than those in its right subtree (if existing). These properties are exploited for parallel prefix computation in the downwards part of the combine phase.

In the `Tree` object, each processor stores the processor IDs of its parent node and of its child nodes in the combine tree.

Hardware support for treelike communication structures is clearly preferable to our software combine trees, at least for the root group. However, in the presence of runtime-data-dependent group splitting, a static preallocation of hardware links for the subgroup combine trees is no longer possible.

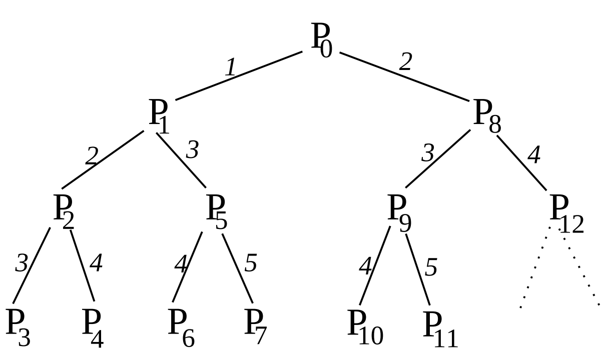
5.4.8 Combining

The combine phase at the end of a `step` consists of an upwards wave of messages going from the leaves towards the root of the tree, followed by a downwards sweep. It is guaranteed that each processor participates (if necessary, by an empty contribution) in each combine phase of a group. Hence, groupwide barrier synchronization is covered by the messages of the combine phase and needs not be carried out separately.

Upwards combining

Now consider a processor p_i that has reached the end of a `step` statement and wants to combine its modified shared variable copies, collected in a list m of combine items sorted by the variables' names, with array indices considered as a part of the variable name. For brevity, we denote combine items by 4-tuples: an ordinary combine item for a variable named `x` and a contributed value v as a 4-tuple $(\texttt{x}, v, ., .)$, and a corresponding prefix item with prefix destination variable `q` and accumulated prefix value k as a 4-tuple $(\texttt{x}, v, \texttt{q}, k)$. If p_i has nothing to contribute, the list m is empty. In that case, p_i is only interested in the implicit barrier associated with this combine operation.

[12]In our prototype implementations we used, for simplicity, a flat tree with $d = p - 1$. This is acceptable for small numbers of processors p.



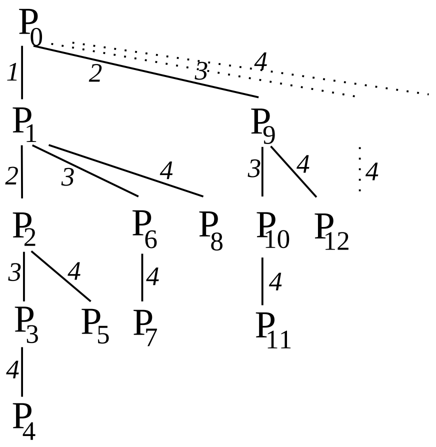


FIGURE 5.13: Two possible combine trees for a group of 13 processors: (left hand side:) a d-ary tree with maximum degree $d = 2$, (right hand side:) a binomial tree. The numbers annotated with the edges indicate the broadcast time distance from the root (assuming sufficient bandwidth), where the time to compose and forward a message to a node is counted as one time unit.

If p_i is a leaf in the combine tree, it just sends m as a *combine* request message to its parent. Otherwise, p_i waits for the *combine* requests m_0,...,m_{nc-1} from its nc children. A shared variable `x` may thus occur in a combine item $(\texttt{x}, v, ., .)$ contributed by p_i itself and/or in combine items $(\texttt{x}, v_i, ., .)$ in some of the messages m_i, $i = 0, ..., nc - 1$. If the combine method declared for `x` is `sh<?>` (arbitrary), it is sufficient to pick one of these combine items and append it to the resulting message $\bar{m}$. Otherwise, p_i accumulatively applies the specified combine function f_x ($f_x \in \{+, *, \&, |\}$ or user-defined) to all these contributed values. The accumulated value $\bar{v}$ is then added as a combine item $(\texttt{x}, \bar{v}, ., .)$ to the resulting message $\bar{m}$. Note that this list processing corresponds to a kind of merge operation if the lists m, m_0,...,m_{nc-1} are sorted. The resulting list m_i is then sorted as well. If the lists are long, this merging could, in the sorted case, also be pipelined by splitting the lists into several packets. Once all variables occuring in m, m_0,...,m_{nc-1} have been processed, the resulting message $\bar{m}$ is sent as a *combine* request to the tree parent of p_i, if p_i is not the root of the tree. The root node adds its contributions for modified `sh<0>` variables. This finishes the upwards combine phase.

Downwards combining

After this first phase, the main result of combining, namely the future values v_x of the combined shared variables x, is available as list $\bar{m}$ at the root processor p_0 of the tree. p_0 creates *commit* messages consisting of commit items from $\bar{m}$ and sends them downwards the tree to its children. All other processors p_i wait for a *commit* message M from their parent. The commit items contain the shared variable's name, its new value, and an accumulated prefix value for prefix commit items. The local copies of shared variables are updated as requested by M. For prefix commit items, the corresponding private variables `q` are updated, and where necessary, modified prefix commit items are forwarded to the children. Finally, the commit items are forwarded to the children (if there are any). Note that, if no prefix computation is involved, the downwards combining phase corresponds just to a groupwide broadcast of $\bar{m}$.

The pseudocode of the downwards combine phase is given in Figure 5.14.

Detection of group termination

Termination of a group is detected by keeping track of the group size. Each combine list has a flag *decrSizeFlag* that can be used to signal the other processors that a processor has terminated its work for the current group (see Figure 5.15). By a prefix-sum combining of all *decrSizeFlag* contributions [13], the current rank and size fields are updated in each combine phase. In this way, the group remains in a consistent state as long as some of its processors may decide to continue and perform further supersteps. The leaving processors must stay with the group as *shadow* members, in order to serve requests to owned distributed shared array elements and to forward messages in the combine tree[14]. The group has completely finished its work as soon as its size field becomes zero.

Inter-subgroup combining for nested supersteps

Consider a group g with $d \geq 2$ subgroups $g_0, ..., g_{d-1}$ executing a `neststep` statement. As illustrated in Figure 4.38, an *inter-subgroup combining phase* takes place immediately before the subgroups g_i are deleted and the parent group g is reactivated.

Each subgroup g_i, $i = 0, ..., d-1$, has committed groupwide some changes to shared variables. For shared variables declared globally to g_i (i.e. declared by g or one of its ancestor groups in the group hierarchy tree) the update requests are collected in a list L_i and must now be combined with the changes potentially requested by other subgroups g_j, $j \neq i$. The list L_i of pending global update requests is maintained by the root processor of the group hierarchy tree for each group; it is initialized to the empty list when the group is created.

As soon as g_i has finished its work, the root node r_i in the combine tree of g_i sends its list L_i of collected global update requests as a *finalCombine* request message to the combine tree root node r of g. If L_i is empty, r_i sends an empty *finalCombine* message to indicate termination of g_i.

Once r has received the update requests from all r_i, $i = 0, ..., d-1$, it computes the g-wide combined value for each variable to be updated, using its respective *combine* function. These values are then broadcast downwards to the r_i as a *finalCommit* message. Each r_i commits these values for its subgroup g_i by propagating the *finalCommit* message downwards its subgroup combine tree. Note that the *finalCommit* message may be empty if there is nothing to commit. Each processor of g_i receiving the *finalCommit* message from its parent in the combine tree forwards it to its children (if there are any), locally performs the requested updates and then returns to execution of the parent group g.

Also, r compiles a list L' of update requests in $\cup_{i=0}^{d-1} L_i$ for the variables that are declared global to g, and appends it to its own list L of global updates.

[13]This is easily integrated into the combining mechanism discussed above by introducing an artificial `ShVar` wrapper object for the size field.

[14]The combine tree is, up to now, not dynamically adjusted when processors leave the group.

```
protected void commit()
{
 int[] tchild ← myGroup.tree.child;
 int $nc$ ← tchild.length;   // number of children in combine tree
 $m$ is the list that I contributed to the upwards combine phase;
 if (rankinit==0) then  // I am the root
   $M \leftarrow \bar{m}$, the list that I compiled at the end of the upwards combine phase,
    with combine items converted to commit items, plus updates of sh<0> variables
 else   receive commit message $M$ from tparent;
 if $(nc > 0)$  then  // I am an inner node:
 {  let $m_i$ be the combine request message
    that I got from my child $i$, $0 \leq i < nc$
   initialize empty downwards messages $M_0, ..., M_{nc-1}$;
   for  all commit items $(\texttt{x}, v, \texttt{q}, p)$ in $M$
   {  if  $(\texttt{x}, v, ., .)$ is not a prefix commit item then {
        append $(\texttt{x}, v, ., .)$ to all $M_i$, $i = 0, ..., nc - 1$;
        locally update x  ←  $v$; }
      else  // $(\texttt{x}, v, \texttt{q}, p)$ is a prefix commit item
      {  x is declared sh<$f_x$:q> with a private variable q
        and combine method $f_x \in \{+, *, \&, |\}$ or user-defined;
        $p$ is the accumulated prefix sum for my subtree.
        if $\exists\ (\texttt{x}, v') \in m$ then {
          locally update q  $leftarrow$  $p$;   $p \leftarrow f_x(p, v')$; }
        for  (int $i = 0$;  $i < nc$;  $i++$)
          if  $\exists\ (\texttt{x}, v'_i) \in m_i$ then  {
            add a prefix commit item (x,$v$,q,$p$) to $M_i$;   $p \leftarrow f_x(p, v'_i)$; }
          else  add a commit item (x,$v$,q,$0$) to $M_i$;
   } }
   for  (int $i = 0$;  $i < nc$;  $i++$)
      send commit message containing $M_i$ to tchild[$i$];
}
 else  // I am a leaf in the combine tree:
   for  all commit items $(\texttt{x}, v, \texttt{q}, p)$ in $M$  {
      locally update x  ←  $v$;
      if  $(\texttt{x}, v, \texttt{q}, p)$ is a prefix item then  locally update q  ←  $p$;  }
 update the size field if requested in $M$;
 if (rankinit==0) then  // I am the root:
   for all updates $u$ in $\bar{m}$ to shared variables declared global to this group:
       append $u$ to list $L$, to be used later in the final combine phase of this group
}
end commit
```

FIGURE 5.14: The *commit* method implements the downwards part of the combine phase for replicated shared variables.

```
protected void leave()
{
 Group mg = Group.myGroup;
 mg.decrSizeFlag = true;   // signal the others that I will leave the group
 while (mg.size > 0)   // more steps: run in shadow mode, participate in combining only
 {  mg.combine();   // upwards
    mg.commit();   // downwards
    serve requests for owned distributed array elements;
    mg.decrSizeFlag = false;
 }
 Group.finalCombine();
 Group.finalCommit();
 Group.myGroup = mg.parent;
}
```

FIGURE 5.15: Group.*leave*() is executed when a processor has finished its work for this group.

Note that with this scheme, concurrent prefix computations performed in different subgroups do not work when addressing the same shared variable declared global to these subgroups, since the subgroupwide prefix results have already gone to some private variables or expressions and thus cannot be updated a posteriori. Hence, the compiler must warn if the programmer applies prefix combine operations to group-global shared variables.

5.4.9 Bulk Mirroring and Updating of Distributed Arrays

According to the language definition in Section 4.5.6, an update to a remote array element becomes effective at the end of the current superstep, while reading a remote array element yields the value it had at the beginning of the superstep. This implies communication of values of array elements at the boundaries of supersteps. For efficiency reasons, updates and reads to entire sections of remote array elements should be communicated together in a bulk way; the routine `mirror` for explicit registering of a bulk array read and `update` for explicit registering of a bulk array update are available for this purpose.

We propose a new method that overlaps the requests for reading and updating remote sections of distributed arrays with the standard combining mechanism for the replicated shared variables (as described above) and the synchronization at the end of a superstep. We show how the combining mechanism can be utilized to avoid the need of a thread-based system for one-sided communication that would otherwise be required for serving these requests.

The basic idea is to compute at runtime, by standard combining, the number of requests that each processor will receive. Then, each processor performs that many blocking receive operations as necessary to receive and serve the requests. This partially decouples the communication of distributed array sections and of the standard combine communication, such that requests for distributed array sections can be already silently sent during the computation part of the superstep. The runtime system distinguishes between these two types of messages by inspecting the message tag in the `MPI_Recv` calls, such that always the right type of message can be received.

Each processor keeps in its `Group` object a hidden, replicated shared integer array, called `N_DAmsgs[]`, whose (dynamically allocated) length is equal to the group size p. Entry `N_DAmsgs[`i`]` is intended to hold the number of messages with requests for reading or updating distributed array sections owned by processor i, $0 \leq i < p$, that are to be sent at the end of the current superstep. Initially, all entries in this array are set to zero. Whenever a processor j sends, during a superstep computation, a message tagged `DA_REQ` with a distributed array request to a remote processor i, it increments the corresponding counter `N_DAmsgs[`i`]` in its local copy. For each processor j itself, its entry `N_DAmsgs[`j`]` in its local copy of that array remains always zero. If any entry has been incremented in a superstep, the array `N_DAmsgs` is appended to the list of combine items for the next combine phase, with combine method `Combine_IADD` (integer addition). Hence, after the combine phase, array `N_DAmsgs` holds the global numbers of messages sent to each processor, as desired.

Now, each processor i executes `N_DAmsgs[`i`]` times `MPI_Recv()` with message tag `DA_REQ`. In the case of a read request, the processor sends the requested value back to the sender with message tag `DA_REP`. In the case of a write request, it performs the desired update computation with its owned array section, following the precedence rules implied by the combine method[15]. Finally, each processor executes that many times `MPI_Recv` as it had previously issued read requests, and stores the received array elements in the destination variable indicated in the combine item holding their values. Although this variable is, in principle, a private array, it is nevertheless registered as a `ShVar` object, such that it can be retrieved at this point. Finally, the counter array `N_DAmsgs` is zeroed again to be ready for the following superstep.

5.4.10 Optimizations

For `step`s, combining and detection of group termination can be skipped if the executing group consists of only one processor. Correspondingly, at subgroup-creating `neststep` statements also the construction and destruction of the `thisgroup` objects for the subgroups is not necessary for one-processor groups. Only the subgroup ID may change; in that case the current group ID is temporarily saved on a hidden stack in the `thisgroup` object.

The tree-based combining mechanism is a general strategy that supports programmable, deterministic concurrent write conflict resolution, programmable concurrent reductions, and programmable parallel prefix computations on shared variables. Nevertheless, for supersteps where the combining phase requires no prefix computations, the downwards combining phase may be replaced by a simple native broadcast operation, which also includes the desired barrier synchronization effect. Furthermore, where it is statically known for a superstep that combining is only needed for a single shared variable which is written by just one processor, the combining phase could be replaced by a (blocking) broadcast from that processor to the group, thus bypassing the combine tree structure. Generally, static analysis of cross-processor dependences may help to replace some supersteps with point-to-point communication, which

[15]Note that prefix combining for elements of distributed arrays is not supported. Hence, potential combining for concurrent write updates to the same element of a distributed array can be done just in the order of arriving messages.

includes replacing groupwide barrier synchronization with bilateral synchronization, without compromising the superstep consistency from the programmer's point of view.

5.4.11 Prototype Implementation

Implementation of NestStep-Java in Java

Our prototype implementation for a subset of *NestStep* (without volatile and distributed arrays) is based on Java as basis language and implementation language. A precompiler translates NestStep-Java source code to ordinary Java source code. As back end we use the SUN Java compiler `javac` to produce Java bytecode that is executed by a set of JVM's running on the participating processors. The NestStep-Java run time system encapsulates message passing, updating of shared variables, and group management. The necessary communication between processors (i.e., JVMs residing on different physical processors) is based on Java object serialization and the Java API for TCP/IP socket communication[16].

A NestStep program is started by a shell script invoking JVM instances on the machines involved. Each processor gets a copy of a *host file* containing the parallel system configuration. From this host file each processor creates a table containing the IP addresses and port numbers of all processors.

The NestStep-Java runtime system is operational for replicated shared variables, objects, and arrays[17], and the driver is finished. Hence, simple hand-translated NestStep-Java programs can be executed.

For the NestStep-Java version of the `parprefix` example (cf. Section 4.5.6) we obtain the following measurements for our NestStep-Java prototype implementation.

`parprefix`	seq	$p = 2$	$p = 4$	$p = 8$	$p = 16$
$N = 1000000$	5.4 s	7.8 s	5.6 s	3.7 s	2.2 s

These times are maximized across the processors involved (JVMs running on loaded SUN SPARC 5/10 machines and Linux PCs), averaged over several runs. We used SUN's `javac` compiler and `java` interpreter from JDK 1.1.5 (Solaris 2.4).

The implementation is optimized to avoid the overhead of parallelism if only one processor is involved. Note that the efficiency of the parallel prefix algorithm is bounded by 50 % since it must perform two sweeps over the array while one sweep is sufficient for the sequential algorithm.

Implementation of NestStep-C in C

We used the free `MPICH` implementation [ANL99] of MPI as the communication layer of the NestStep-C runtime system.

[16]As an alternative, one could use an MPI binding for Java, as described e.g. for HPJava [CZF$^+$98].

[17]Distributed arrays are only available in the C version.

NestStep-C **BSP parameters** In our current prototype implementation of NestStep-C, we found that the time for an empty superstep (the l parameter of the BSP model) is approximately

$$l(p) = 1.3 + 0.1 \cdot p \text{ milliseconds}$$

where the factor p is due to the fact that the network is a bus network (Ethernet), which sequentializes all point-to-point communications, even where these may, in principle, be performed concurrently.

In contrast, on an architecture based on a hypercube network, for instance, a combine tree could—at least for the root group—be embedded immediately into the host network, such that communication within different subtrees can proceed concurrently and each tree edge coincides one-to-one with a physical hypercube link [BT89, Sect. 1.3]. In that case, the factor p could be replaced by a term linear in the maximum node degree, $\log p$, times the tree depth, which is $\Theta(\log p)$. The node degree factor disappears if transmission along all the $\log p$ links of a node can proceed concurrently.

The inverse-bandwidth parameter g is also a function of p. We found that for large array updates, g is dominated by the network speed:

$$g(p) = 1.5 + 0.8 \cdot p \text{ microseconds per array element}$$

For updates of only a few elements of an array, g is at least the inverse bandwidth for scalar updates:

$$g(p) = 80 + 8 \cdot p \text{ microseconds per scalar variable updated}$$

which is dominated by the software overhead of processing the combine items. The above discussion of the factor p holds here accordingly.

As the value h for the h-relation, we use the number of (replicated) shared variables updated (for scalar variables) or the number of replicated shared array elements updated (for array section updates), respectively. Then we obtain the following time formula for a NestStep-C superstep:

$$t(p) = \max_{0 \le i < p} w(i,p) + \max_{0 \le i < p} h(i,p) \cdot g(p) + l(p)$$

where $w(i,p)$ denotes the local computation work performed by the processor ranked i in a group of p processors.

Measurements for example programs For the NestStep-C version of the `parprefix` example as shown in Section 4.5.6 we obtain the following measurements for our NestStep-C prototype implementation.

`parprefix`	seq	$p=2$	$p=3$	$p=4$	$p=5$
N=1000000	0.790 s	0.397 s	0.268 s	0.205 s	0.165 s
N=10000000	7.90 s	4.00 s	2.70 s	1.98 s	1.59 s

For the integration-based PI calculation example in Figure 4.39, we obtain the following figures:

picalc	seq	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$
N=1000000	0.838 s	0.466 s	0.337 s	0.277 s	0.255 s	0.230 s
N=10000000	8.354 s	4.224 s	2.844 s	2.158 s	1.758 s	1.495 s

For the randomized BSP quicksort program in Appendix C.1 the parallel speedup is less impressive because large array sections must be shipped around in the bulk remote read and write operations, and all this communication is sequentialized on the bus network. We obtain the following figures:

quicksort	seq	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$
N=80000	0.641 s	0.393 s	0.428 s	0.437 s	0.391 s	0.375 s
N=120000	0.991 s	0.622 s	0.582 s	0.564 s	0.502 s	0.485 s

All measurements are wall clock times that were taken on a network of PCs running Linux, connected by Ethernet. The NestStep application had no exclusive use of the network.

Summary and Future Work

For the prototype implementations of NestStep-Java and NestStep-C, we observe a similar effect as in the implementation of ForkLight: A substantial amount of performance is lost in programs with frequent synchronization because the runtime system working on top of MPI or on top of JVM processes, which are scheduled by the local system scheduler on their machines, has no control about the scheduling mechanism. Hence, delays incurred by a processor when working on a critical path in the communication tree accumulate during the computation. We expect that this effect will be less dramatic on homogenous multiprocessors where NestStep can use the processors and the network exclusively.

Further work on the present two implementations of NestStep could unfortunately not be done at the University of Trier due to manpower limitations. The author, who has now too many other commitments to continue working on these implementations alone, hopes to be able to continue this work in the future when the current shortage of master students is overcome. In particular, further development and more experiments with the NestStep-C implementation are deferred to future work. For instance, we plan, time permitting, to run NestStep-C applications on real, massively parallel supercomputers. For now, the current status of implementation is sufficient—from a research point of view—to demonstrate the feasibility of our approach.

5.5 Summary

We have discussed the compilation aspects for synchronous, control-synchronous and super-step-synchronous languages that support static and dynamic nesting of parallelism. We have also considered compilation for various types of target machines.

For all three languages, Fork, ForkLight and NestStep, the group concept is implemented by maintaining some group frame data structure that holds all relevant data for a group. This data structure is split into a part residing in shared memory (if there is one) and another part that resides in the local processor memory. Only when compiling for the SB-PRAM where shared memory access takes the same time as a private memory access, it makes sense to hold as much information as possible in the shared memory part, in order to save overall memory space (which is a scarce resource on the SB-PRAM). Otherwise, it is advisable to hold as much group information locally as possible to take advantage of the faster local memory access. For ForkLight, only the synchronization cells and shared variables must be kept in shared memory, and for NestStep, all group information is replicated across the processors and kept consistent at superstep boundaries.

Unfortunately, the SB-PRAM is just a research prototype and not generally accessible. Moreover, the SB-PRAM is optimized for irregular applications and is, at least in terms of peak performance, not competitive—even if the same chip technology would be applied—with massively parallel supercomputers that are mainly designed to solve regular, LINPACK-like problems fast. Also, Fork can, in general, not be compiled to efficient code for any other parallel architecture than the SB-PRAM. Hence, Fork is, based on the present compiler and the SB-PRAM simulator, limited in its applicability to teaching purposes and to gaining more insights into the practical behaviour of PRAM algorithms devised by the parallel theory community [B1].

ForkLight is some kind of compromise between the high programming comfort known from Fork and the features offered by modern, asynchronous MIMD shared memory machines. Hence, it has much better chances to be compiled to efficient code for sequentially consistent shared memory architectures such as the Tera MTA. Retargeting the ForkLight implementation to further shared memory platforms based on the recent Open-MP interface is an issue of future work that could unfortunately not be done at the University of Trier due to manpower limitations.

NestStep has the largest potential for practical use, as it is compiled to workstation and PC clusters that are now ubiquitous, and that may even be heterogeneous if the underlying message passing interface supports heterogeneity. Future work should investigate the potential of the new implementation of remote access to distributed shared array sections in NestStep-C. Further research and implementation work on NestStep may consider static and dynamic load balancing and other static optimizations such as automatic prefetching or fuzzy barriers.

When considering the sequence of implementations as language–platform pairs Fork–SB-PRAM, ForkLight–P4, and NestStep–MPI, we observe that

- the implementation platform becomes more and more realistic and available,
- the features of the source language are more and more relaxed to allow compact and efficient implementations for the corresponding implementation platform,
- the software part of the implementation, in particular the runtime system, becomes more and more dominant,
- virtualization becomes simpler,
- the relative cost of group management and of groupwide barriers increases, and

- thus the support of customized variants of the group-splitting operations gains importance.

Acknowledgements

The material presented in Sections 5.1 and 5.2 largely follows the presentation in Chapter 6 of *Practical PRAM Programming* [B1].

The main strategy for the implementation of the group concept of Fork for the SB-PRAM, such as the frame concept and the translation scheme for the `if` statement and the loops in synchronous regions, has been described in a diploma thesis by Joachim Lillig [Lil93] supervised by Helmut Seidl and Reinhard Wilhelm at the University of Saarbrücken. This work, which was based on the old `FORK` standard of 1989, described also the principles of simulating larger PRAMs on a smaller one, including the self-restoring barrier with three cells that we used with the asynchronous PRAM, and the problem of asynchronously accessed shared variables in Fork. The ideas for the implementation of the simple lock, the fair lock and the reader–writer lock have been taken from the diploma thesis by Jochen Röhrig [Röh96] at the University of Saarbrücken, who in turn was inspired by the work of Wilson [Wil88] for the NYU Ultracomputer.

The author thanks the current and former members of the SB-PRAM group around Wolfgang Paul, in particular, Michael Bosch, Stefan Franziskus, Jochen Röhrig, and Dieter Scheerer, for their help with system-specific questions and for the support of the SB-PRAM system software.

The author thanks also all the users of the Fork compiler for many bug reports and comments that helped to improve the compiler.

Appendix A

Supplementary Material to Chapter 2

A.1 Proof of the optimality of *labelfs2*

In this appendix we show that a call *labelfs2*(v) (see Section 2.3.5) generates an optimal contiguous schedule of a tree with import and export nodes rooted at v, which uses *label*(v) registers. We prove this by two lemmata:

Lemma A.1 *Let $T = (V, E)$ be a tree and $v \in V$ be an arbitrary inner node of T. labelfs2 generates a schedule for cone(v) that uses label(v) registers.*

Proof: By induction. Let v be an inner node with two children v_1 and v_2. S_1 is the subtree with root v_1, S_2 is the subtree with root v_2. We suppose that the algorithm generates schedules for *coneDAG*(T, v_1) and *coneDAG*(T, v_2) that use label(v_1) and label(v_2) registers. To evaluate v, we have two possibilities if we use contiguous schedules: If we evaluate v_1 before v_2, we use

$$m_1 = \max(\mathit{label}(v_1), \mathit{label}(v_2) + \mathit{occ}(v_1) + 1 - \mathit{freed}(v_1))$$

registers. We need *occ*(v_1) registers to hold the export nodes of T_1 and one register to hold v_1. On the other hand, we free *freed*(v_1) registers when evaluating S_1. If we evaluate v_2 before v_1, we use

$$m_2 = \max(\mathit{label}(v_2), \mathit{label}(v_1) + \mathit{occ}(v_2) + 1 - \mathit{freed}(v_2))$$

registers. The algorithm evaluates v_1 before v_2 iff

$$\mathit{label}(v_1) + \mathit{occ}(v_2) - \mathit{freed}(v_2) \geq \mathit{label}(v_2) + \mathit{occ}(v_1) - \mathit{freed}(v_1)) \qquad \text{(A.1)}$$

If condition (A.1) is true, then $m_1 \leq m_2$ and the algorithm uses the best schedule. If condition (A.1) is not true, then the algorithm evaluates v_2 before v_1. In this case, we have $m_2 \leq m_1$ and the algorithm again uses the best schedule. □

Lemma A.2 *Let $T = (V, E)$ be a tree and $v \in V$ be an arbitrary inner node of T. label(v) is a lower bound for the minimal number of registers needed by a contiguous schedule for v.*

Proof: Let S be the smallest subtree of T for which a violation of the lemma occurs. Let v be the root of S with children v_1 and v_2. Let S_1 denote the subtree of S rooted at v_1, and S_2 the subtree rooted at v_2.

We consider the case that

$$\mathit{label}(v_1) + \mathit{occ}(v_2) - \mathit{freed}(v_2) \geq \mathit{label}(v_2) + \mathit{occ}(v_1) - \mathit{freed}(v_1) \tag{A.2}$$

(the case $<$ is symmetric). Then

$$\mathit{label}(v) = \max(\mathit{label}(v_1), \mathit{label}(v_2) + \mathit{occ}(v_1) + 1 - \mathit{freed}(v_1))$$

The lemma holds for S_1 and S_2, otherwise we have found a smaller violation tree. There are two possibilities to evaluate S: Evaluating S_1 before S_2, uses at least $\mathit{label}(v_1)$ registers for S_1. $\mathit{occ}(v_1)$ registers are required to hold the export nodes of S_1 and one register is needed to hold v_1. $\mathit{freed}(v_1)$ registers are freed. We need at least $\mathit{label}(v_2)$ registers for S_2. So we need at least $\mathit{label}(v)$ registers for S.

Evaluating S_2 before S_1, uses at least $\mathit{label}(v_2)$ registers for S_2. $\mathit{occ}(v_2)$ registers are required to hold the export nodes of S_2 and one register is needed to hold v_2. $\mathit{freed}(v_2)$ registers are free. For v_1 we need at least $\mathit{label}(v_1)$ registers, thus we need at least $m = \max(\mathit{label}(v_2), v_1) + \mathit{occ}(v_2) + 1 - \mathit{freed}(v_2))$ registers for S. Because of equation (A.2) we have

$$\mathit{label}(v) = \max(\mathit{label}(v_1), \mathit{label}(v_2) + \mathit{occ}(v_1) + 1 - \mathit{freed}(v_1)) \leq m$$

and hence we need at least $\mathit{label}(v)$ registers to evaluate S. □

Until now, we have assumed that two different import nodes of a tree T_i have different corresponding export nodes. We now explain what has to be done if this is not true. Let $A = \{w_1, \ldots, w_n\} \subseteq V_i$ be a set of import nodes of T_i with the same corresponding export node that is stored in a register r. As described above we have set

$$imp_p(w_1) = \ldots = imp_p(w_n) = 1 \text{ and } imp_{np}(w_1) = \ldots = imp_{np}(w_n) = 0$$

But r can be freed, after the last node of A is evaluated. By choosing an appropriate node $w \in A$ to be evaluated last, T_i eventually can be evaluated with one register less than the label of the root specifies. We determine w by a top–down traversal of T_i. Let v be an inner node of T_i with children v_1 and v_2. Let S_j be the subtree with root v_j, $j = 1, 2$. If only one of S_1 and S_2 contains nodes of A, we descend to the root of this tree. If both S_1 and S_2 contain nodes of A, we examine, whether we can decrease the label value of v by choosing S_1 or S_2. Let be $a = \mathit{label}(v_1) + \mathit{occ}(v_2) - \mathit{freed}(v_2)$ and $b = \mathit{label}(v_2) + \mathit{occ}(v_1) - \mathit{freed}(v_1)$ If $a > b$, this can only be achieved by searching w in S_1. If $a < b$, this can only be achieved by searching w in S_2. If $a = b$, we cannot decrease the register need and can search in S_1 or S_2.

We repeat this process until we reach a leaf $w \in A$. We set $imp_p(w) = 0$, $imp_{np}(w) = 1$.

Appendix B

Supplementary Material to Chapter 3

B.1 An Example for SPARAMAT Analysis

As an example, consider the following Fortran implementation of a Hopfield neural network simulation based on a sparse matrix describing the synapse interconnections of the neurons.

```
   program main
   integer wfirst(21), wcol(20), i, j, k, n, nz
   real wdata(100), xst(20), stimul(20), val(20),
   real alpha, beta, tinv, mexp, pexp, accum, tanhval

c  read test matrix in csr format:
   wfirst(1)=1
   read(*,*) n
   do i = 1, n
     read(*,*) k
     wfirst(i+1) = wfirst(i)+k
     do j = wfirst(i),wfirst(i+1)-1
       read(*,*) wdata(j)
       read(*,*) wcol(j)
     enddo
   enddo
   nz = wfirst(n+1)-1

c  simulate a hopfield network (learning online)
c  with interconnection matrix (wdata,wcol,wfirst):
   niter=100
   alpha=0.2
   beta=0.7
   tinv=1.0
   do i = 1, n
     stimul(i) = -1.0
     xst(i) = 0.0
   enddo
   do k = 1, niter
     do i = 1, n
       accum = 0.0
       do j = wfirst(i), wfirst(i+1)-1
         accum = accum + wdata(j)*xst(wcol(j))
       enddo
       val(i) = beta*accum + alpha*stimul(i)
     enddo
     do i = 1, n
       pexp = exp(val(i))
       mexp = exp(-val(i))
       tanhval = (pexp-mexp) / (pexp+mexp)
       xst(i) = tanhval
     enddo
     do i = 1, n
       do j = wfirst(i), wfirst(i+1)-1
         wdata(j)=wdata(j)+tinv*(xst(i)*xst(wcol(j)))
       enddo
     enddo
     tinv = tinv * 0.9
   enddo
   do i = 1, n
```

```
     write (*,*) xst(i)
   enddo
   end
```

After applying concept matching and optimizing the format property conditions, the unparsed program looks as follows:

```
program main

integer wfirst(21), wcol(20), k, n, nz
real wdata(100), xst(20), stimul(20), val(20), tinv
real mexp(20), pexp(20), accum(20)

MREAD( CSR(V(wdata,1,nz,1),IV(wfirst,1,n+1,1),
           IV(col,1,n,1),n,nz, stdin, _simplehb) )

<assume monotonicity of wfirst(1:n+1)>
<assume injectivity of wcol(wfirst(i):wfirst(i+1)) forall i in 1:n>

SINIT(tinv,1.0)
VINIT( V(stimul,1,n,1), -1.0)
VINIT( V(xst,1,n,1), 0.0)
do k = 1, 100
  VMATVECMV( V(accum,1,n,1),
             CSR( V(wdata,1,nz,1),IV(wfirst,1,n+1,1), IV(col,1,n,1),n,nz),
             V(xst,1,n,1), VCON(0.0,n)
  VMAPVV( V(val,1,n,1), ADD, VMAPVS(MUL, V(accum,1,n,1), 0.7),
                             VMAPVS(MUL, V(stimul,1,n,1), 0.2))
  VMAPV( V(pexp,1,n,1), EXP, V(val,1,n,1))
  VMAPV( V(mexp,1,n,1), EXP, VMAPV(NEG, V(val,1,n,1)))
  VMAPVV( V(xst,1,n,1), DIV,
          VMAPVV(ADD, V(pexp,1,n,1), VMAPV(NEG, V(mexp,1,n,1))),
          VMAPVV(ADD, V(pexp,1,n,1), V(mexp,1,n,1)))
  MOUTERVV( CSR( V(wdata,1,nz,1),IV(wfirst,1,n+1,1), IV(col,1,n,1),n,nz),
            VMAPVS( MUL, V(xst,1,n,1), tinv ), V(xst,1,n,1) )
  SCAL(tinv, 0.9)
enddo
VWRITE( V(xst,1,n,1), stdout )
end
```

B.2 Example for the STRIP Concept

While scanning FORTRAN source codes for concepts and templates of sparse matrix computations, we discovered a new type of memory access concept that is encountered in practically all occurrences of CSR sparse matrix computations. We refer to this memory access concept as the STRIP concept (see also Section 3.3.1).

We illustrate the STRIP concept at a typical occurrence in a particular SPARSKIT implementation of matrix–matrix multiplication for the CSR format. Instead of building the values of the elements in the output matrix consecutively, this implementation creates partial sums in a temporary vector corresponding to elements for a particular row in the output matrix. For each nonzero element $A_{i,k}$ in the current row of A the column index k is used to select the corresponding row in B, and that row k is multiplied with $A_{i,k}$. Each product is added to a zero-initialized temporary array that is indexed by the column index k. Then, the same process is repeated with the next nonzero value in the current row i of A and the corresponding row k' of B. Once the iteration over row i of A is finished, the values in the temporary array are copied into the output matrix C, and the temporary array entries are restored to zero. This process continues until all rows of A are processed. Figure B.1 shows the situation where the values of the first row of the input matrix A are selected, together with the selected corresponding column indices in B.

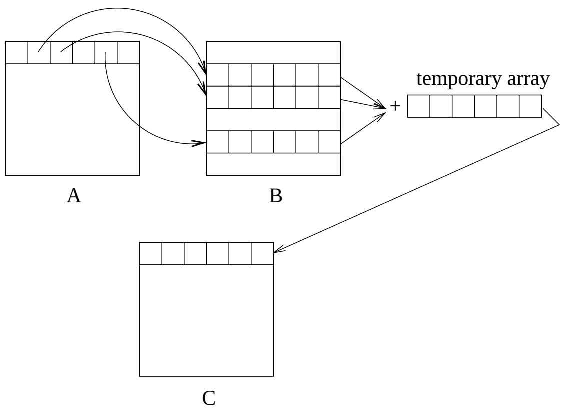


FIGURE B.1: Matrix–Matrix Multiplication for CSR.

This algorithm is driven by the column indices k for the current row i of A. The first-in-row array of A yields, indexed by i, the starting position in the array of the column values k of A. As, after matching the inner loops, all these k values are considered simultaneously as a single vector access, they select a set of entire rows (i.e., a set of vector accesses to the same array) in B, as the elements selected in B's first-in-row array indicate the beginning and end positions of the selected rows in B. This multiple-indirect memory access pattern is referred to as the `STRIP` concept. `STRIP` earned its name by how it selects a set of contiguous regions, i.e. strips, of an array (here, B). See Figure B.2 for the `STRIP` concept instance representing the situation shown in Figure B.3.

```
STRIP(B, {IVX(FirB, {IVX(ColA, {IV(FirA, {RANGE(1,2,1)})})})})
```

FIGURE B.2: A `STRIP` instance.

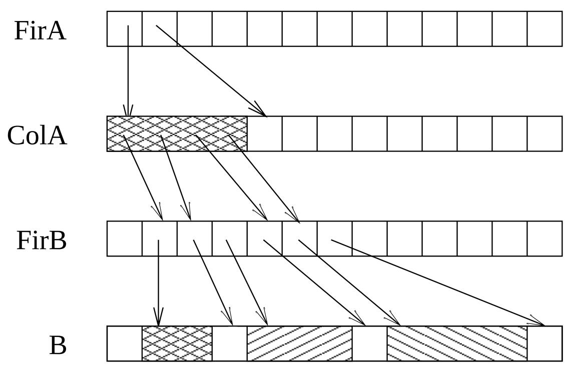


FIGURE B.3: Visualization of the `STRIP` concept.

Appendix C

Supplementary Material to Chapters 4 and 5

C.1 NestStep Implementation of BSP p-way Quicksort

The following code implements a simplified version of a randomized BSP algorithm by Gerbessiotis and Valiant [GV94] for p-way Quicksort.

We are given a distributed shared array A that holds N numbers to be sorted. The p processors sample their data and agree on $p - 1$ pivots that are stored in a replicated shared array pivs. Now processor i, $1 < i < p - 1$, becomes responsible for sorting all array elements whose value is between pivs[$i - 1$] and pivs[i]; processor 0, for all elements smaller than pivs[0], and processor $p - 1$, for all elements larger than pivs[$p - 2$], respectively. A bad load distribution can be avoided with high probability if oversampling is applied, so that the pivots define subarrays of approximately equal size. The partitioned version of array A is temporarily stored in the distributed target array B, by a remote write operation. For this operation, the processors determine the sizes and prefix sums of global sizes of the subarrays they will become responsible for. Then, they perform a remote read from B to local subarrays T. These are sorted in parallel, and then the sorted sections are copied back to B by a remote write operation.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "NestStep.h"

/**
 * p-way randomized Combine-CRCW BSP-Quicksort implementation
 * variant by Gerbessiotis/Valiant JPDC 22(1994)
 * implemented in NestStep-C.
 */
int N=10;               // default value

#define debmain 1       // switch off to debug only the run time system
#define MAXNDISPLAY 20  // max. size of local partition to be printed

/** findloc():
 *  find largest index j in [0..n-1] with a[j] <= key < a[j+1]
 *  (where a[n] = +infty).
 *  C’s bsearch() cannot be used, as this requires a[j]==key.
 */
int findloc( void *key, void *a, int n, int size,
             int (*cmp)(const void *, const void *) )
{
 int j;
 // for the first, use a naive linear search:
 for (j=0; j<n; j++)
   if ( cmp( key, a+j*size ) <= 0)  return j;
 return n-1;
}

int flcmp( const void *a, const void *b )   // compare two floats
{
 if (*(float *)a < *(float *)b) return -1;
 if (*(float *)a > *(float *)b) return 1;
 return 0;
}

int main( int argc, char *argv[] )
{
  // shared variables:
  sh float *A</>, *B</>;   // block-wise distributed arrays
  sh float *size, *pivs;   // replicated shared arrays
  // private variables such as T used as destination of remote read
  // are automatically treated in a special way by the compiler:
  float *T;
  int *mysize, *Ssize, *prefsize;
  float **myslice;
  // local variables:
  int i, j, ndp, nmp, myndp, pivi, psumsize, ctr;
  double startwtime, endwtime;

  NestStep_init( &argc, &argv );   // sets NUM and PID

  if (argc > 1)    // N is passed as a parameter:
     sscanf( argv[1], "%d", &N );

  ndp = N / #;
  nmp = N % #;
  if (PID < nmp) myndp = ndp+1;
  else           myndp = ndp;
  A = new_DArray( N, Type_float ); // dynamic allocation of distr. A[1:N]
  B = new_DArray( N, Type_float ); // dynamic allocation of distr. B[1:N]
  mysize = new_Array( #, Type_int ); // dynamic allocation of private array
  Ssize = new_Array( #, Type_int );
  size = new_Array( #, Type_int );
  prefsize = new_Array( #, Type_int );
  pivs = new_Array( #, Type_int );

  forall (i, A)                    // useful macro supplied by NestStep
     A[i] = (double)(rand()%1000000); // create some values to be sorted

  // print the unsorted distributed input array:
  myndp = DArray_Length(A);       // the size of my owned local partition
  if (myndp < MAXNDISPLAY)
    forall (i, A)
```

```
      fprintf(stderr,"P%d: A[i=%d] = %2.2f\n", $$, i, A );

  startwtime = NestStep_time();  // here starts the algorithm

  // STAGE 1: each processor randomly selects a pivot element
  //          and writes (EREW) it to pivs[PID].
  // Pivot intervals are defined as follows:
  // I(P_0)     = ] -\infty, pivs[0] ]
  // I(P_i)     = ] pivs[i-1], pivs[i] ],  0<i<NUM-2
  // I(P_NUM-1) = ] pivs[NUM-2], \infty [

  step { /*1*/
    int pivi = DArray_index( rand()%myndp ); // random index in my partition
    pivs[PID] = A[pivi];                     // write (now only locally visible)
  } // here the writes to A in this step are combined and globally committed
  // now the pivs are in pivs[0..NUM-1]

  // STAGES 2, 3, 4 form a single superstep:

  step { /*2*/
   // STAGE 2: locally sort pivots. This computation is replicated
   // (which is not work-optimal, but faster than performing the computation
   //  on one processor only and communicating the result to the others.)

   qsort( pivs, NUM, sizeof(float), flcmp );

   // STAGE 3:  local p-way partitioning in my local array A->array
   // I have no in-place algorithm, so I overallocate O(p*(ndp-p)) space,
   // only npd elements of which will be actually filled by the partitioning.
   myslice = (float **)malloc( NUM * sizeof(float *));
   myslice[0] = (float *)malloc( NUM * ndp * sizeof(float));
   for (i=1; i<NUM; i++) myslice[i] = myslice[0] + i*ndp;
   for (i=0; i<NUM; i++) size[i] = 0;
   forall (i, A) {  // insert my owned elements into slices:
     j = findloc( &(A[i]), pivs, NUM, sizeof(float), flcmp );
     myslice[j][size[j]++] = A[i];
   }
   for (i=0; i<NUM; i++) mysize[i] = size[i]; // keep the local sizes for later

   // Now write the partitions back to the contiguous local array partition:
   ctr=0;
   for (i=0; i<NUM; i++)
     for (j=0; j<mysize[i]; j++)
       A[ctr++] = myslice[i][j];

   // STAGE 4: Globally compute the array size of global slice sizes
   //          and the prefixes of the individual processor contributions.
   //          This is just done as a side-effect of combining:

  } combine ( size[:]<+:prefsize[:]> );  // end of superstep 2

  // This array ^^^^ notation in the optional combine annotation
  // is a hint for the compiler that the shared array "size"
  // needs not be packed elementwise, thus creating smaller messages.
  // Here, the effect is marginal, as "size" is usually small.

  // After combining, size[i] holds the global size of pivot interval I(i),
  // and prefsize[i] holds the accumulated length of contributions of the
  // processors P0..P{PID-1} from pivot interval I(i).
  // now prefsize[i] holds the prefix sum of all size[i] contributions
  // made by processors 0...PID-1

  // STAGE 5: Write the local slices to the corresponding parts in array B.

  step { /*3*/
   psumsize = 0;
   for (i=0; i<NUM; i++) {
    // remote write to distributed array B:
    B[ psumsize+prefsize[i], psumsize+prefsize[i] + mysize[i]-1 ]
      = myslice[ 0: mysize[i]-1 ];
    Ssize[i] = psumsize;  // prefix sum size[0]+...+size[i-1]
    psumsize += size[i];  // is computed by replicated computation
   }
  }
  // this combining includes the updates to B

  // STAGE 6:
  // Allocate space for all elements in my pivot interval I(PID),
  // and bulk-read the myslice slices from the partitioned array A.

  step { /*4*/
    T = NestStep_new_Array( size[PID], Type_float ); // dynamic allocation
    // remote read from distributed array B:
    T[ 0 : size[PID]-1 ]
      = B [ Ssize[PID] : Ssize[PID]+size[PID]-1 ];
  }

  // Stages 7 and 8 can be combined into a single superstep,
  // because Stage 7 contains only local computation.

  step { /*5*/

   // STAGE 7: Now each processor sorts its array T.

   qsort( T, /* this is recognized as a NestStep array by the compiler */
          size[PID], sizeof(float), flcmp );

   // STAGE 8: Now write T back to B.

   // bulk remote write to B:
   B [ Ssize[PID] : Ssize[PID]+size[PID]-1 ]
     = T [ 0 : size[PID] - 1 ]
  }

  endwtime = NestStep_Wtime();

  // print distributed result array:
  if (myndp < MAXNDISPLAY)
   forall (i, B)
    fprintf(stderr, "P%d: B[%d]=%2.2f\n", PID, i, B[i] );

  fprintf(stderr,"P%d: wall clock time = %f\n", PID, endwtime-startwtime);
  fprintf(stderr,"P%d: Number of supersteps = %d\n", PID,
          thisgroup->stepcounter  /*which is 5 here*/ );

  NestStep_finalize();
} // main
```

Bibliography

[ACC$^+$90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. 4th ACM Int. Conf. Supercomputing*, pp. 1–6. ACM Press, 1990.

[ACD$^+$97] K. T. Au, M. M. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, M. Köhler, G. Keller, W. Pfannenstiel, and M. Simons. Enlarging the Scope of Vector-Based Computations: Extending Fortran 90 by Nested Data Parallelism. In *Proc. APDC'97 Int. Conf. Advances in Parallel and Distributed Computing*, pp. 66–73. IEEE Computer Society Press, 1997.

[ADK$^+$93] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer J.*, 36(8):756–762, Dec. 1993.

[AEBK94] W. Ambrosch, M. Ertl, F. Beer, and A. Krall. Dependence–conscious global register allocation. In *Proc. Conf. on Programming Languages and System Architectures*, pp. 125–136. Springer LNCS 782, Mar. 1994.

[AG85] A. V. Aho and M. Ganapathi. Efficient Tree Pattern Matching: an Aid to Code Generation. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 334–340, 1985.

[AGT89] A. V. Aho, M. Ganapathi, and S. W. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.*, 11(4):491–516, Oct. 1989.

[AH90] Z. Ammarguellat and W. L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 283–295. ACM Press, June 20–22 1990.

[AI91] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 39–50, 1991.

[AJ76] A. Aho and S. Johnson. Optimal Code Generation for Expression Trees. *J. ACM*, 23(3):488–501, July 1976.

[AJU77] A. Aho, S. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1), Jan. 1977.

[AKLS88] E. Albert, K. Knobe, J. D. Lukas, and G. L. Steele Jr. Compiling Fortran 8x Array Features for the Connection Machine Computer Systems. In *ACM SIGPLAN Symp. Parallel Programming: Experiences with Applications, Languages and Systems*, pp. 42–56, 1988.

[AKP91] F. Abolhassan, J. Keller, and W. J. Paul. On the cost–effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. Parallel and Distributed Processing*, pp. 2–9. IEEE Computer Society Press, Dec. 1991.

[AN88] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 308–317. ACM Press, 1988.

[ANL99] ANL Argonne National Laboratories. MPICH - a portable implementation of MPI, version 1.2. www-unix.mcs.anl.gov/mpi/mpich/, 1999.

[ANN95] A. Aiken, A. Nicolau, and S. Novack. Resource-Constrained Software Pipelining. *IEEE Trans. Parallel and Distrib. Syst.*, 6(12):1248–1270, Dec. 1995.

[ANS90] ANSI American National Standard Institute, Inc., New York. American National Standards for Information Systems, Programming Language C. ANSI X3.159–1989, 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *COMPILERS: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[BAFR96] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph. ParC — An Extension of C for Shared Memory Parallel Processing. *Software Pract. Exp.*, 26(5):581–612, May 1996.

[Bal90] V. Balasundaram. A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor. *J. Parallel and Distrib. Comput.*, 9:154–170, 1990.

[Ban93] U. Banerjee. *Loop Transformations for Restructuring Compilers:* Vol. I*: The Foundations*. Kluwer Academic Publishers, 1993.

[Ban94] U. Banerjee. *Loop Transformations for Restructuring Compilers:* Vol. II*: Loop Parallelization*. Kluwer Academic Publishers, 1994.

[Ban97] U. Banerjee. *Loop Transformations for Restructuring Compilers:* Vol. III*: Dependence Analysis*. Kluwer Academic Publishers, 1997.

[BAW96] Y. Ben-Asher and R. Wilhelm. Compilation Techniques for Fair Execution of Shared Memory Parallel Programs over a Network of Workstations. http://cs.haifa.ac.il/YOSI/PAPERS/pcomp.ps, 1996.

[BBC$^+$94] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[BBJ98] S. Ben Hassen, H. E. Bal, and C. Jacobs. A Task and Data Parallel Programming Language based on Shared Objects. *ACM Trans. Program. Lang. Syst.*, 1998.

[BCKT89] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 275–284, 1989.

[BCT92] P. Briggs, K. Cooper, and L. Torczon. Rematerialization. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 311–321, 1992.

[BDH$^+$91] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved Deterministic Parallel Integer Sorting. *Information and Computation*, 94, 1991.

[BDO$^+$95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A structured high level programming language and its structured support. *Concurrency – Pract. Exp.*, 7(3):225–255, 1995.

[BDP94] B. Bacci, M. Danelutto, and S. Pelagatti. Resource Optimisation via Structured Parallel Programming. In [DR94], pp. 1–12, Apr. 1994.

[BE91] J. Boyland and H. Emmelmann. Discussion: Code Generator Specification Techniques (summary). In *Code Generation: Concepts, Tools, Techniques [GG91]*, pp. 66–69, 1991.

[BEH91] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 122–131, Apr. 1991.

[BFJ$^+$96] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proc. 8th Annual ACM Symp. Parallel Algorithms and Architectures*, pp. 297–308, 1996.

[BG89] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay on one cycle. *ACM Trans. Program. Lang. Syst.*, 11(1):57–67, Jan. 1989.

[BGM$^+$89] D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 258–263, 1989.

[BGS95] D. A. Berson, R. Gupta, and M. L. Soffa. HARE: A Hierarchical Allocator for Registers in Multiple Issue Architectures. Technical report, Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260, Feb. 1995.

[BGW92] E. D. Brooks III, B. C. Gorda, and K. H. Warren. The Parallel C Preprocessor. *Sci. Progr.*, 1(1):79–89, 1992.

[BH98] H. E. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurr.*, 6(3):74–84, 1998.

[BHRS94] S. Bhansali, J. R. Hagemeister, C. S. Raghavendra, and H. Sivaraman. Parallelizing sequential programs by algorithm-level transformations. In V. Rajlich and A. Cimitile, eds., *Proc. 3rd IEEE Workshop on Program Comprehension*, pp. 100–107. IEEE Computer Society Press, Nov. 1994.

[BHS$^+$94] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel and Distrib. Comput.*, 21:4–14, 1994.

[Bik96] A. J. C. Bik. *Compiler support for sparse matrix computations*. PhD thesis, Leiden University, 1996.

[BJK$^+$95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multi-threaded run-time system. In *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 207–216, 1995.

[BJR89] D. Bernstein, J. M. Jaffe, and M. Rodeh. Scheduling arithmetic and load operations in parallel with no spilling. *SIAM J. Comput.*, 18:1098–1127, 1989.

[BJvR98] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library: Design, Implementation and Performance. Technical Report tr-rsfb-98-063, Heinz Nixdorf Institute, Dept. of Computer Science, University of Paderborn, Germany, Nov. 1998.

[BJvR99] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library: Design, Implementation and Performance. In *Proc. IPPSSPDP'99 IEEE Int. Parallel Processing Symp. and Symp. Parallel and Distributed Processing*, 1999.

[BK96] G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proc. 5th Int. Symp. High Performance Distributed Computing* (*HPDC*), pp. 243–252. IEEE Computer Society Press, 1996.

[BKK93] R. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1993.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a Language for Parallel Programming of Distributed Systems. *IEEE Trans. on Software Engineering*, 18(3):190–205, Mar. 1992.

[BL92] R. Butler and E. L. Lusk. User's Guide to the P4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.

[BL94] R. Butler and E. L. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20(4):547–564, Apr. 1994.

[Ble96] G. Blelloch. Programming Parallel Algorithms. *Comm. ACM*, 39(3):85–97, Mar. 1996.

[Blu94] W. Blume et al. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, Aug. 1994.

[Bos88] P. Bose. Interactive Program Improvement via EAVE: An Expert Adviser for Vectorization. In *Proc. ACM Int. Conf. Supercomputing*, pp. 119–130, July 1988.

[BPRD97] R. Boisvert, R. Pozo, K. Remington, and J. Dongarra. Matrix-market: a web resource for test matrix collections. In R. B. et al., ed., *The Quality of Numerical Software: Assessment and Enhancement*, pp. 125–137. Chapman and Hall, 1997.

[Bra99] T. Brandes. Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library. In Amestoy, P. and Berger, P. and Dayde, M. and Duff, I. and Giraud, L. and Frayssé, V. and Ruiz, D., ed., *Proc. Int. Euro-Par Conf.*, pp. 833–844. Springer LNCS 1685, Sept. 1999.

[BRG89] D. Bernstein, M. Rodeh, and I. Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.*, 38(9):1308–1314, Sept. 1989.

[BS76] J. Bruno and R. Sethi. Code generation for a one–register machine. *J. ACM*, 23(3):502–510, July 1976.

[BS87] T. Brandes and M. Sommer. A Knowledge-Based Parallelization Tool in a Programming Environment. In *Proc. 16th Int. Conf. Parallel Processing*, pp. 446–448, 1987.

[BSB96] M. J. Bourke, P. H. Sweany, and S. J. Beaty. Extended List Scheduling to Consider Execution Frequency. In *Proc. 29th Annual Hawaii Int. Conf. System Sciences*, Jan. 1996.

[BSBC95] T. S. Brasier, P. H. Sweany, S. J. Beaty, and S. Carr. Craig: A practical framework for combining instruction scheduling and register assignment. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 1995.

[BSC96] R. M. Badia, F. Sanchez, and J. Cortadella. OSP: Optimal Software Pipelining with Minimum Register Pressure. Technical Report UPC-DAC-1996-25, DAC Dept. d'arquitectura de Computadors, Universitat Polytecnica de Catalunya, Barcelona, Campus Nord. Modul D6, E-08071 Barcelona, Spain, June 1996.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods*. Prentice-Hall, 1989.

[BW96] A. J. C. Bik and H. A. G. Wijshoff. Automatic Data Structure Selection and Transformation for Sparse Matrix Computations. *IEEE Trans. Parallel and Distrib. Syst.*, 7(2):109–126, Feb. 1996.

[CAC$^+$81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[Cal91] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proc. 4th Annual Workshop on Languages and Compilers for Parallel Computing*, 1991.

[CBS95] J. Cortadella, R. M. Badia, and F. Sanchez. A Mathematical Formulation of the Loop Pipelining Problem. Technical Report UPC-DAC-1995-36, Department of Computer Architecture, Universitat Politecnica de Catalunya, Campus Nord. Modul D6, E-08071 Barcelona, Spain, Oct. 1995.

[CC92] I. A. Carmichael and J. R. Cordy. *TXL - Tree Transformational Language Syntax and Informal Semantics*. Dept. of Computing and Information Science, Queen's University at Kingston, Canada, Feb. 1992.

[CC93] I. A. Carmichael and J. R. Cordy. *The TXL Programming Language Syntax and Informal Semantics Version 7*. Dept. of Computing and Information Science, Queen's University at Kingston, Canada, June 1993.

[CC94] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dartmouth College, Computer Science, Hanover, NH, 1994.

[CC95] H.-C. Chou and C.-P. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Trans. Parallel and Distrib. Syst.*, 6(3):303–313, 1995.

[CCK91] D. Callahan, S. Carr, and K. Kennedy. Register Allocation via Hierarchical Graph Coloring. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 192–203, June 1991.

[CD95] W. W. Carlson and J. M. Draper. Distributed Data Access in AC. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 39–47. ACM Press, 1995.

[CDG$^+$93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing'93*, Nov. 1993.

[CG72] E. Coffman and R. Graham. Optimal scheduling for two–processor systems. *Acta Informatica*, 1:200–213, 1972.

[CG89] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989.

[CGMS94] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–656, Apr. 1994.

[CGSvE92] D. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. 19th Int. Symp. Computer Architecture*, May 1992.

[CH84] F. C. Chow and J. L. Hennessy. Register allocation by priority-based coloring. *ACM SIGPLAN Notices*, 19(6):222–232, 1984.

[Cha82] G. J. Chaitin. Register allocation & spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):201–207, 1982.

[Cha87] D. Chase. An improvement to bottom-up tree pattern matching. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 168–177, 1987.

[CK87] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. In *First Int. Conf. on Supercomputing, Athens (Greece)*. Springer LNCS 297, June 1987.

[CK88] D. Callahan and K. Kennedy. Compiling Programs for Distributed Memory Multiprocessors. *J. Supercomputing*, 2:151–169, 1988.

[CM98] B. Chapman and P. Mehrotra. OpenMP and HPF: Integrating Two Paradigms. In *Proc. Int. Euro-Par Conf.*, 1998.

[CMRZ94] B. Chapman, P. Mehrotra, J. V. Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. In *Proc. CONPAR*, Sept. 1994.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Sci. Progr.*, 1(1):31–50, 1992.

[Cof76] E. Coffman Jr. (Ed.). *Computer and Job/Shop Scheduling Theory*. John Wiley & Sons, 1976.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman and MIT Press, 1989.

[Coo73] S. A. Cook. An observation on time–storage trade–off. In *Proc. 5th Annual Symposium on Theory of Computing*, pp. 29–33, 1973.

[CS90] D. Callahan and B. Smith. A Future-based Parallel Language for a General-Purpose Highly-parallel Computer. Report, Tera Computer Company, Seattle, WA, `http://www.tera.com`, 1990.

[CSG99] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[CSS95] M. M. T. Chakravarty, F. W. Schröer, and M. Simons. V - Nested Parallelism in C. In W. Giloi, S. Jähnichen, and B. Shriver, eds., *Proc. 2nd Int. Conf. Massively Parallel Programming Models*, pp. 167–174. IEEE Computer Society Press, 1995.

[CV91] B. S. Chlebus and I. Vrto. Parallel Quicksort. *J. Parallel and Distrib. Comput.*, 11:332–337, 1991.

[CZ89] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM model. In *Proc. 1st Annual ACM Symp. Parallel Algorithms and Architectures*, pp. 169–178, 1989.

[CZ95] R. Cole and O. Zajicek. The Expected Advantage of Asynchrony. *J. Comput. Syst. Sciences*, 51:286–300, 1995.

[CZF$^+$98] B. Carpenter, G. Zhang, G. Fox, X. Li, X. Li, and Y. Wen. Towards a Java Environment for SPMD Programming. In *Proc. 4th Int. Euro-Par Conf.*, pp. 659–668. Springer LNCS 1470, 1998.

[DF84] J. W. Davidson and C. W. Fraser. Code Selection through Object Code Optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, Oct. 1984.

[DFH$^+$93] J. Darlington, A. J. Field, P. G. Harrison, P. H. B. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *Proc. Conf. Parallel Architectures and Languages Europe*, pp. 146–160. Springer LNCS 694, 1993.

[DGNP88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[DGTY95] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. ACM Press, July 1995. *SIGPLAN Notices* 30(8), pp. 19–28.

[Dha88] D. M. Dhamdhere. A fast algorithm for code movement optimization. *SIGPLAN Notices*, 23(10):172–180, 1988.

[Dha91] D. M. Dhamdhere. A fast practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Trans. Program. Lang. Syst.*, 13(2):291–294, 1991.

[DI94] B. DiMartino and G. Ianello. Towards Automated Code Parallelization through Program Comprehension. In V. Rajlich and A. Cimitile, eds., *Proc. 3rd IEEE Workshop on Program Comprehension*, pp. 108–115. IEEE Computer Society Press, Nov. 1994.

[DJ79] J. Dongarra and A. Jinds. Unrolling Loops in Fortran. *Software Pract. Exp.*, 9(3):219–226, 1979.

[dlTK92] P. de la Torre and C. P. Kruskal. Towards a Single Model of Efficient Computation in Real Parallel Machines. *Future Generation Computer Systems*, 8:395–408, 1992.

[DR94] K. M. Decker and R. M. Rehmann, eds. *Programming Environments for Massively Parallel Distributed Systems*. Birkhäuser, Basel (Switzerland), 1994. Proc. IFIP WG 10.3 Working Conf. at Monte Verita, Ascona (Switzerland), Apr. 1994.

[Duf77] I. S. Duff. `MA28` – a set of Fortran subroutines for sparse unsymmetric linear equations. Tech. rept. AERE R8730, HMSO, London. Source code available via netlib [NET], 1977.

[EK91] M. A. Ertl and A. Krall. Optimal Instruction Scheduling using Constraint Logic Programming. In *Proc. 3rd Int. Symp. Programming Language Implementation and Logic Programming* (*PLILP*), pp. 75–86. Springer LNCS 528, Aug. 1991.

[EK92] M. A. Ertl and A. Krall. Instruction scheduling for complex pipelines. In *Compiler Construction (CC'92)*, pp. 207–218, Paderborn, 1992. Springer LNCS 641.

[Ell85] J. R. Ellis. *Bulldog: A Compiler for VLIW Architechtures*. PhD thesis, Yale University, 1985.

[ELM95] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proc. 5th Workshop on Compilers for Parallel Computers*, pp. 503–516. Dept. of Computer Architecture, University of Malaga, Spain. Report No. UMA-DAC-95/09, June 28–30 1995.

[EN89] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *Proc. 3rd ACM Int. Conf. Supercomputing*. ACM Press, 1989.

[Ers71] A. P. Ershov. *The Alpha Programming System*. Academic Press, London, 1971.

[Ert99] M. A. Ertl. Optimal Code Selection in DAGs. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, 1999.

[Far98] J. Farley. *JAVA Distributed Computing*. O'Reilly, 1998.

[Fea91] P. Feautrier. Dataflow Analysis of Array and Scalar References. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991.

[Fer90] C. Ferdinand. Pattern Matching in TRAFOLA. Diplomarbeit, Universität des Saarlandes, Saarbrücken, Germany, 1990.

[FH91a] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software Pract. Exp.*, 21(9):963–988, Sept. 1991.

[FH91b] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

[FH95] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings Publishing Co., 1995.

[FHP92a] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, Sept. 1992.

[FHP92b] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — Fast Optimal Instruction Selection and Tree Parsing. *ACM SIGPLAN Notices*, 27(4):68–76, Apr. 1992.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C–30(7):478–490, July 1981.

[FK95] A. Formella and J. Keller. Generalized Fisheye Views of Graphs. In *Proc. Graph Drawing '95*, pp. 242–253. Springer LNCS 1027, 1995.

[FOP$^+$92] A. Formella, A. Obe, W. Paul, T. Rauber, and D. Schmidt. The SPARK 2.0 system – a special purpose vector processor with a VectorPASCAL compiler. In *Proc. 25th Annual Hawaii Int. Conf. System Sciences*, volume 1, pp. 547–558, 1992.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[FR92] S. M. Freudenberger and J. C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation: Concepts, Tools, Techniques [GG91]*, pp. 146–170, 1992.

[Fre74] R. Freiburghouse. Register allocation via usage counts. *Comm. ACM*, 17(11), 1974.

[FRR99] U. Fissgus, T. Rauber, and G. Rünger. A framework for generating task parallel programs. In *Proc. 7th Symp. Frontiers of Massively Parallel Computation*, pp. 72–80, Annapolis, Maryland, 1999.

[FSW92] C. Ferdinand, H. Seidl, and R. Wilhelm. Tree Automata for Code Selection. In *Code Generation: Concepts, Tools, Techniques* [GG91], pp. 30–50, 1992.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th Annual ACM Symp. Theory of Computing*, pp. 114–118, 1978.

[FWH$^+$94] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The Polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, Oct. 1994.

[GA96] L. George and A. W. Appel. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, May 1996.

[Gav72] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1(2):180–187, 1972.

[GBD$^+$94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Sept. 1994.

[GE92] C. H. Gebotys and M. I. Elmasry. *Optimal VLSI Architectural Synthesis*. Kluwer, 1992.

[GE93] C. H. Gebotys and M. I. Elmasry. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, CAD-12(9):1266–1278, Sept. 1993.

[GG78] R. Glanville and S. Graham. A New Method for Compiler Code Generation. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 231–240, Jan. 1978.

[GG91] R. Giegerich and S. L. Graham, eds. *Code Generation - Concepts, Tools, Techniques*. Springer Workshops in Computing, 1991.

[GG96] W. Gellerich and M. M. Gutzmann. Massively Parallel Programming Languages – a Classification of Design Approaches. In *Proc. Int. Symp. Computer Architecture*, volume I, pp. 110–118. ISCA, 1996.

[GH88] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. ACM Int. Conf. Supercomputing*, pp. 442–452. ACM Press, July 1988.

[Gib89] P. B. Gibbons. A More Practical PRAM Model. In *Proc. 1st Annual ACM Symp. Parallel Algorithms and Architectures*, pp. 158–168, 1989.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. With an addendum, 1991.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GLR83] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Large Numbers of Cooperating Sequential Processes. *ACM Trans. Program. Lang. Syst.*, 5(2):164–189, Apr. 1983. (see also: Ultracomputer Note No. 16, Dec. 1980, New York University).

[GM86] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. ACM SIGPLAN Symp. Compiler Construction*, pp. 11–16, July 1986.

[GP99] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim et al., ed., *IPPS/SPDP'99 Workshops Proceedings, IEEE Int. Parallel Processing Symp. and Symp. Parallel and Distributed Processing*, pp. 123–137. Springer LNCS 1586, 1999.

[GR90] M. Golumbic and V. Rainish. Instruction scheduling beyond basic blocks. *IBM J. Res. Develop.*, 34(1):94–97, Jan. 1990.

[Gri84] R. G. Grimes. `SPARSE-BLAS` basic linear algebra subroutines for sparse matrices, written in Fortran77. Source code available via netlib [NET], 1984.

[Gro92] J. Grosch. Transformation of Attributed Trees using Pattern Matching. In U. Kastens and P. Pfahler, eds., *Fourth Int. Conf. on Compiler Construction (CC'92), Springer LNCS vol. 641*, pp. 1–15, Oct. 1992.

[GS90] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, Apr. 1990.

[GSW95] M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, Jan. 1995.

[Gup89] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proc. 3rd Int. Conf. Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-III*), pp. 54–63. ACM Press, 1989.

[Güt81] R. Güttler. Erzeugung optimalen Codes für series–parallel graphs. In *Sprinter LNCS 104*, pp. 109–122, 1981.

[Güt82] R. Güttler. *Die Erzeugung optimalen Codes für series–parallel-graphs in polynomieller Zeit*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1982.

[GV94] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel and Distrib. Comput.*, 22:251–267, 1994.

[GYZ$^+$99] R. Govindarajan, H. Yang, C. Zhang, J. N. Amaral, and G. R. Gao. Minimum register instruction sequence problem: Revisiting optimal code generation for dags. CAPSL Technical Memo 36, Computer Architecture and Parallel Systems Laboratory, University of Delaware, Newark, Nov. 1999.

[GZG99] R. Govindarajan, C. Zhang, and G. R. Gao. Minimum Register Instruction Scheduling: a New Approach for Dynamic Instruction Issue Processors. In *Proc. Annual Workshop on Languages and Compilers for Parallel Computing*, 1999.

[HB94] D. Hearn and M. P. Baker. *Computer Graphics, Second Edition*. Prentice-Hall, 1994.

[HE91] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[HFG89] W.-C. Hsu, C. N. Fischer, and J. R. Goodman. On the minimization of loads/stores in local register allocation. *IEEE Trans. on Software Engineering*, 15(10):1252–1262, Oct. 1989.

[HG83] J. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, July 1983.

[HHG$^+$95] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler Technology for Future Microprocessors. *Proc. of the IEEE*, 83(12):1625–, Dec. 1995.

[HHL$^+$93] S. U. Hänßgen, E. A. Heinz, P. Lukowicz, M. Philippsen, and W. F. Tichy. The Modula-2* Environment for Parallel Programming. In *Proc. 1st Int. Conf. Massively Parallel Programming Models*. IEEE Computer Society Press, 1993.

[Hig93] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Sci. Progr.*, 2, 1993.

[HK91] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Trans. Parallel and Distrib. Syst.*, 2(3):350–359, July 1991.

[HKT91a] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed Memory Machines. In *Proc. 5th ACM Int. Conf. Supercomputing*, pp. 86–100, Nov. 1991.

[HKT91b] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler-Support for Machine-Independent Parallel Programming in Fortran-D. Technical Report Rice COMP TR91-149, Rice University, Mar. 1991.

[HL91] V. Herrarte and E. Lusk. Studying parallel program behaviour with `upshot`. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1991.

[HLG$^+$99] C. Herrmann, C. Lengauer, R. Günz, J. Laitenberger, and C. Schaller. A Compiler for $\mathcal{HDC}$. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau, Germany, May 1999.

[HLQA92] P. J. Hatcher, A. J. Lapadula, M. J. Quinn, and R. J. Anderson. Compiling Data-Parallel Programs for MIMD Architectures. In H. Zima, ed., *Proc. 3rd Workshop on Compilers for Parallel Computers*, pp. 347–358. Technical report ACPC/TR 92-8 of the Austrian Center of Parallel Computation, July 1992.

[HMS$^+$98] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: the BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.

[HN90] M. Harandi and J. Ning. Knowledge-based program analysis. *IEEE Software*, pp. 74–81, Jan. 1990.

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern Matching in Trees. *J. ACM*, 29(1):68–95, Jan. 1982.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer J.*, 5(4):10–15, 1962.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):667–677, Aug. 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition edition, 1996.

[HQ91] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, 1991.

[HR92a] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. I: The Model. *J. Parallel and Distrib. Comput.*, 16:212–232, 1992.

[HR92b] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. II: Binary Tree and FFT Algorithms. *J. Parallel and Distrib. Comput.*, 16:233–249, 1992.

[HS93] R. Heckmann and G. Sander. TrafoLa-H Reference Manual. In B. Hoffmann and B. Krieg-Brückner, eds., *Program Development by Specification and Transformation: The PROSPECTRA Methodology, Language Family, and System*, pp. 275–313. Springer LNCS Vol. 680, 1993.

[HS96] J. M. D. Hill and D. B. Skillicorn. Practical Barrier Synchronisation. Technical Report PRG-TR-16-96, Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1996.

[HSS92] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A High-Level Language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[Hu61] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(11), 1961.

[IBM] IBM. Visualization Tool (VT): Execution Analysis on the IBM SP. http://www.tc.cornell.edu/UserDoc/Software/PTools/vt/.

[Inm84] Inmos Ltd. *OCCAM Programming Manual*. Prentice-Hall, New Jersey, 1984.

[Ive62] K. E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[JE94] T. E. Jeremiassen and S. J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques* (*PACT*), pp. 171–180, Aug. 1994.

[JG88] G. Jones and M. Goldsmith. *Programming in Occam 2*. Prentice-Hall, 1988.

[Jor86] H. F. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3:93–110, 1986.

[JR96] K. Jansen and J. Reiter. Approximation Algorithms for Register Allocation. Technical Report 96-13, Universität Trier, FB 4 - Mathematik/Informatik, D-54286 Trier, Germany, 1996.

[Juv92a] S. Juvaste. An Implementation of the Programming Language pm2 for PRAM. Technical Report A-1992-1, Dept. of Computer Science, University of Joensuu, Finland, 1992. `ftp://cs.joensuu.fi`.

[Juv92b] S. Juvaste. The Programming Language pm2 for PRAM. Technical Report B-1992-1, Dept. of Computer Science, University of Joensuu, Finland, 1992. `ftp://cs.joensuu.fi`.

[Käp92] K. Käppner. Analysen zur Übersetzung von FORK, Teil 1. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[Käs97] D. Kästner. Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung f"ur den digitalen Signalprozessor adsp-2106x. Diplomarbeit, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[KE93] D. R. Kerns and S. J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. ACM Press, June 1993.

[Keß94] C. W. Keßler, ed. *Automatic Parallelization—New Approaches to Code Generation, Data Distribution and Performance Prediction*. Vieweg, Wiesbaden (Germany), 1994.

[KG92] T. Kiyohara and J. C. Gyllenhaal. Code Scheduling for VLIW/Superscalar Processors with Limited Register Files. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*. IEEE Computer Society Press, 1992.

[KH93] P. Kolte and M. J. Harrold. Load/Store Range Analysis for Global Register Allocation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 268–277, June 1993.

[KHJ98] J.-S. Kim, S. Ha, and C. S. Jhon. Relaxed barrier synchronization for the BSP model of computation on message-passing architectures. *Inform. Process. Lett.*, 66:247–253, 1998.

[Kla94] A. C. Klaiber. *Architectural Support for Compiler-Generated Data-Parallel Programs*. PhD thesis, University of Washington, 1994.

[KMR90] C. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting shared data structures on distributed memory architectures. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 177–186, 1990.

[KNE93] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Trans. on Software Engineering*, 18(12):1065–1075, Dec. 1993.

[KNS92] W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. In *Proc. 7th Knowledge-Based Software Engineering Conf. (KBSE'92)*, pp. 216–225, 1992.

[KP95] S. Kannan and T. Proebsting. Register Allocation in Structured Programs. In *Proc. 6th Annual ACM/SIAM Symp. Discrete Algorithms*, 1995.

[KPF95] S. M. Kurlander, T. A. Proebsting, and C. N. Fisher. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Trans. Program. Lang. Syst.*, 17(5):740–776, Sept. 1995.

[KPS94] J. Keller, W. J. Paul, and D. Scheerer. Realization of PRAMs: Processor Design. In *Proc. WDAG'94 8th Int. Workshop on Distributed Algorithms*, pp. 17–27. Springer Lecture Notes in Computer Science 857, 1994.

[Kri90] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *ACM SIGPLAN Notices*, 25(7):97–106, 1990.

[Kro75] H. Kron. *Tree Templates and Subtree Transformational Grammars*. PhD thesis, UC Santa Cruz, Dec. 1975.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1992. ACM SIGPLAN Notices **27**(7):224–234.

[KRS94] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994.

[KRS98] J. Knoop, O. Rüthing, and B. Steffen. Code motion and code placement: Just synonyms? In *Proc. European Symp. Programming*. Springer LNCS, 1998.

[Kun88] K. S. Kundert. `SPARSE 1.3` package of routines for sparse matrix LU factorization, written in C. Source code available via netlib [NET], 1988.

[Lam88] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN Symp. Compiler Construction*, pp. 318–328, July 1988.

[Lan97] M. Langenbach. Instruktionsanordnung unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), Oct. 1997.

[LC90] J. Li and M. Chen. Synthesis of Explicit Communication from Shared Memory Program References. In *Proc. Supercomputing '90*, Nov. 1990.

[LE95] J. L. Lo and S. J. Eggers. Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. ACM Press, June 1995.

[Lei97] C. E. Leiserson. Programming Irregular Parallel Applications in Cilk. In *Proc. IRREGULAR'97 Int. Symp. Solving Irregularly Structured Problems in Parallel*, pp. 61–71. Springer LNCS 1253, June 1997.

[Li77] H. Li. Scheduling trees in parallel / pipelined processing environments. *IEEE Trans. Comput.*, C-26(11):1101–1112, Nov. 1977.

[Lil93] J. Lillig. Ein Compiler für die Programmiersprache `FORK`. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 1993.

[LLG$^+$92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH multiprocessor. *IEEE Comput.*, 25(3):63–79, 1992.

[LLM$^+$87] E. Lawler, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer. Pipeline Scheduling: A Survey. Technical Report Computer Science Research Report, IBM Research Division, San Jose, CA, 1987.

[LM97] R. Leupers and P. Marwedel. Time-constrained code compaction for DSPs. *IEEE Transactions on VLSI Systems*, 5(1), 1997.

[LMN$^+$00] M. Leair, J. Merlin, S. Nakamoto, V. Schuster, and M. Wolfe. Distributed OMP — A Programming Model for SMP Clusters. In *Proc. 8th Workshop on Compilers for Parallel Computers*, pp. 229–238, Jan. 2000.

[LMW88] P. Lipps, U. Möncke, and R. Wilhelm. Optran: A Language/System for the Specification of Program Transformations: System Overview and Experiments. In D. Hammer, ed., *Compiler Compilers and High–Speed Compilation*, pp. 52–65. Springer LNCS vol. 371, 1988.

[LS85] K.-C. Li and H. Schwetman. Vector C: A Vector Processing Language. *J. Parallel and Distrib. Comput.*, 2:132–169, 1985.

[LSRG95] C. León, F. Sande, C. Rodríguez, and F. García. A PRAM Oriented Language. In *Proc. EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing*, pp. 182–191. IEEE Computer Society Press, Jan. 1995.

[LVA95] J. Llosa, M. Valero, and E. Ayguade. Bidirectional scheduling to minimize register requirements. In *Proc. 5th Workshop on Compilers for Parallel Computers*, pp. 534–554. Dept. of Computer Architecture, University of Malaga, Spain. Report No. UMA-DAC-95/09, June 28–30 1995.

[LVW84] J. Leung, O. Vornberger, and J. Witthoff. On some variants of the bandwidth minimization problem. *SIAM J. Comput.*, 13:650–667, 1984.

[MAL93] D. Maydan, S. Amarasinghe, and M. Lam. Array data-flow analysis and its use in array privatization. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 2–15. ACM Press, Jan. 1993.

[Man82] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, 1982.

[Mar97] M. I. Marr. *Descriptive Simplicity in Parallel Computing*. PhD thesis, University of Edinburgh, 1997.

[Mat90] N. Mathis. Weiterentwicklung eines Codeselektorgenerators und Anwendung auf den NSC32000. Diplomarbeit, Universität des Saarlandes, Saarbrücken, Germany, 1990.

[McC96] W. F. McColl. Universal computing. In *Proc. 2nd Int. Euro-Par Conf.*, volume 1, pp. 25–36. Springer LNCS 1123, 1996.

[MD94] W. M. Meleis and E. D. Davidson. Optimal Local Register Allocation for a Multiple-Issue Machine. In *Proc. ACM Int. Conf. Supercomputing*, pp. 107–116, 1994.

[Met95] R. Metzger. Automated Recognition of Parallel Algorithms in Scientific Applications. In *IJCAI-95 Workshop Program Working Notes: "The Next Generation of Plan Recognition Systems"*. sponsored jointly by IJCAII/AAAI/CSCSI, Aug. 1995.

[MI96] B. D. Martino and G. Iannello. Pap Recognizer: a Tool for Automatic Recognition of Parallelizable Patterns. In *Proc. 4th IEEE Workshop on Program Comprehension*. IEEE Computer Society Press, Mar. 1996.

[MPI97] MPI Forum. MPI-2: Extensions to the Message Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997. http://www.erc.msstate.edu/labs/hpcl/projects/mpi/mpi2.html.

[MPSR95] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining Register Allocation and Instruction Scheduling (Technical Summary). Technical Report TR 698, Courant Institute of Mathematical Sciences, New York, July 1995.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Comm. ACM*, 22(2):96–103, 1979.

[MR90] M. Metcalf and R. Reid. *Fortran90 Explained*. Oxford University Press, 1990.

[MS94] T. MacDonald and Z. Sekera. The Cray Research MPP Fortran Programming Model. In [DR94], pp. 1–12, Apr. 1994.

[MSS$^+$88] R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of run-time support for parallel processors. In *Proc. 2nd ACM Int. Conf. Supercomputing*, pp. 140–152. ACM Press, July 1988.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NET] NETLIB. Collection of free scientific software. Accessible by anonymous ftp to `netlib2.cs.utk.edu` or `netlib.no` or e-mail `"send index"` to `netlib@netlib.no`.

[NG93] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, 1993.

[Nic84] A. Nicolau. Percolation scheduling: A parallel compilation technique. Technical Report 85-678, Cornell University, 1984.

[NP90] A. Nicolau and R. Potasman. Incremental Tree Height Reduction for Code Compaction. Technical Report 90-12, Dept. of Information and Computer Science, University of California Irvine, Irvine, CA 92717, 1990.

[NP94] C. Norris and L. L. Pollock. Register Allocation over the Program Dependence Graph. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 266–277. ACM Press, June 1994.

[NP98] C. Norris and L. L. Pollock. The Design and Implementation of RAP: A PDG-based Register Allocator. *Software Pract. Exp.*, 28(4):401–424, 1998.

[NS95] B. Natarajan and M. Schlansker. Spill-Free Parallel Scheduling of Basic Blocks. In *Proc. 28th Annual IEEE/ACM Int. Symp. Microarchitecture*. IEEE Computer Society Press, 1995.

[OP99] S. Orlando and R. Perego. $COLT_{\mathtt{HPF}}$, a Run-Time Support for the High-Level Co-ordination of HPF Tasks. *Concurrency – Pract. Exp.*, 11(8):407–434, 1999.

[Ope97] OpenMP Architecture Review Board. OpenMP: a Proposed Industry Standard API for Shared Memory Programming. White Paper, `http://www.openmp.org/`, Oct. 1997.

[Pal96] Pallas GmbH. *VAMPIR User's Manual*. Pallas GmbH, Brühl, Germany, 1996. http://www.pallas.de/pages/vampir.htm.

[PDB93] S. Prakash, M. Dhagat, and R. Bagrodia. Synchronization issues in data-parallel languages. In *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, pp. 76–95. Springer LNCS 768, Aug. 1993.

[PE95] B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proc. 9th ACM Int. Conf. Supercomputing*, pp. 444–448, July 1995.

[Pel98] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor&Francis, 1998.

[PF91] T. A. Proebsting and C. N. Fischer. Linear–time, optimal code scheduling for delayed–load architectures. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 256–267, June 1991.

[PF92] T. A. Proebsting and C. N. Fischer. Probabilistic Register Allocation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 300–310, June 1992.

[PF94] T. A. Proebsting and C. W. Fraser. Detecting Pipeline Structural Hazards Quickly. In *Proc. 21st ACM SIGPLAN Symp. Principles of Programming Languages*, 1994.

[PG88] E. Pelegri-Llopart and S. L. Graham. Optimal Code Generation for Expression Trees: An Application of BURS Theory. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 294–308, 1988.

[PH95] M. Philippsen and E. Heinz. Automatic Synchronization Elimination in Synchronous FORALLs. In *Proc. 5th Symp. Frontiers of Massively Parallel Computation*, 1995.

[Pin93] S. Pinter. Register allocation with instruction scheduling: a new approach. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*. ACM Press, 1993.

[PM94] M. Philippsen and M. U. Mock. Data and Process Alignment in Modula-2*. In *Proc. AP'93* [Keß94], pp. 177–191, 1994.

[PP91] S. S. Pinter and R. Y. Pinter. Program Optimization and Parallelization Using Idioms. In *Proc. ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 79–92, 1991.

[PP94] S. Paul and A. Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Trans. on Software Engineering*, 20(6):463–475, 1994.

[Pro98] T. Proebsting. Least-cost Instruction Selection for DAGs is NP-Complete. unpublished, http://www.research.microsoft.com/ toddpro/papers/proof.htm, 1998.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry — an Introduction*. Springer–Verlag, New York, 1985.

[PS90] K. V. Palem and B. B. Simons. Scheduling time–critical instructions on RISC machines. In *Proc. 17th ACM SIGPLAN Symp. Principles of Programming Languages*, pp. 270–280, 1990.

[PS93] K. V. Palem and B. B. Simons. Scheduling time–critical instructions on RISC machines. *ACM Trans. Program. Lang. Syst.*, 15(4), Sept. 1993.

[PT78] W. J. Paul and R. E. Tarjan. Time–space trade–offs in a pebble game. *Acta Informatica*, 10:111–115, 1978.

[PT92] M. Philippsen and W. F. Tichy. Compiling for Massively Parallel Machines. In *Code Generation: Concepts, Tools, Techniques* [GG91], pp. 92–111, 1992.

[PTC77] W. J. Paul, R. E. Tarjan, and J. Celoni. Space bounds for a game on graphs. *Math. Systems Theory*, 10:239–251, 1977.

[PTM96] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology*, 4(2):63–79, 1996.

[PTVF92] W. H. Press, S. A. Teukolski, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition*. Cambridge University Press, 1992.

[PW96] T. A. Proebsting and B. R. Whaley. One-pass, optimal tree parsing — with or without trees. In T. Gyimothy, ed., *Compiler Construction (CC'96)*, pp. 294–308. Springer LNCS 1060, 1996.

[QH90] M. J. Quinn and P. Hatcher. Data-Parallel Programming on Multicomputers. *IEEE Software*, pp. 124–131, Sept. 1990.

[QHS91] M. J. Quinn, P. J. Hatcher, and B. Seevers. Implementing a Data-Parallel Language on Tightly Coupled Multiprocessors. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds., *Advances in Languages and Compilers for Parallel Processing*, pp. 385–401. Pitman / MIT Press, London, 1991.

[RAA$^+$93] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callaghan, E. Richards, and W. Smith. The Paragon performance monitoring environment. In *Proc. Supercomputing '93*, pp. 850–859. IEEE Computer Society Press, 1993.

[Ran87] A. G. Ranade. How to emulate shared memory. In *Proc. 28th Annual IEEE Symp. Foundations of Computer Science*, pp. 185–194, 1987.

[RAN$^+$93] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pp. 104–113. IEEE Computer Society Press, 1993.

[RBJ88] A. G. Ranade, S. N. Bhatt, and S. L. Johnson. The Fluent Abstract Machine. In *Proc. 5th MIT Conference on Advanced Research in VLSI*, pp. 71–93, Cambridge, MA, 1988. MIT Press.

[RF93] X. Redon and P. Feautrier. Detection of Recurrences in Sequential Programs with Loops. In *Proc. Conf. Parallel Architectures and Languages Europe*, pp. 132–145. Springer LNCS 694, 1993.

[Röh96] J. Röhrig. Implementierung der P4-Laufzeitbibliothek auf der SB-PRAM. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 1996.

[RP89] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 69–89. ACM Press, 1989.

[RP94] L. Rauchwerger and D. Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. 8th ACM Int. Conf. Supercomputing*, pp. 33–43. ACM Press, July 1994.

[RR98] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *J. Systems Architecture*, 45:483–503, 1998.

[RR99] T. Rauber and G. Rünger. A coordination language for mixed task and data parallel programs. In *Proc. 13th Annual ACM Symp. Applied Computing*, pp. 146–155, San Antonio, Texas, 1999.

[RR00] T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Trans. on Software Engineering*, 26(4):315–339, 2000.

[RS87] J. Rose and G. Steele. C*: an Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Inc., Cambridge, MA, 1987.

[RS92] G. Rünger and K. Sieber. A Trace-Based Denotational Semantics for the PRAM-Language FORK. Technical Report C1-1/92, Sonderforschungsbereich 124 VLSI-Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[RW90] C. Rich and L. M. Wills. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software*, pp. 82–89, Jan. 1990.

[Saa94] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations, Version 2. Research report, University of Minnesota, Minneapolis, MN 55455, June 1994.

[Sch73] J. T. Schwartz. On Programming: An Interim Report on the SETL Project. Technical report, Courant Institute of Math. Sciences, New York University, 1973.

[Sch91] A. Schmitt. A Formal Semantics of FORK. Technical Report C1-11/91, Sonderforschungsbereich 124 VLSI-Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken, Germany, 1991.

[Sch92] A. Schmitt. *Semantische Grundlagen der PRAM-Sprache* FORK. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[Sch97] R. Schreiber. High Performance Fortran, Version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.

[SDB93] A. Skjellum, N. Dorr, and P. Bangalore. Writing Libraries in MPI. In *Scalable Parallel Libraries Conf.* IEEE Computer Society Press, 1993.

[Sei93] H. Seidl. Equality of Instances of Variables in FORK. Technical Report C1-6/93, Sonderforschungsbereich 124 VLSI-Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken, Germany, 1993.

[Sei00] H. Seidl. Personal communication, 2000.

[Set75] R. Sethi. Complete register allocation problems. *SIAM J. Comput.*, 4:226–248, 1975.

[Set76] R. Sethi. Scheduling graphs on two processors. *SIAM J. Comput.*, 5(1):73–82, 1976.

[SG89] M. K. Seager and A. Greenbaum. Slap: Sparse Linear Algebra Package, Version 2. Source code available via netlib [NET], 1989.

[SMS96] T. Sterling, P. Merkey, and D. Savarese. Improving Application Performance on the HP/Convex Exemplar. *IEEE Comput.*, 29(12):50–55, 1996.

[Sny82] L. Snyder. Recognition and Selection of Idioms for Code Optimization. *Acta Informatica*, 17:327–348, 1982.

[SO97] E. A. Stöhr and M. F. P. O'Boyle. Barrier Synchronisation Optimisation. In *Proc. HPCN Europe 1997*. Springer LNCS 1225, Apr. 1997.

[SOH$^+$96] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[SR00] M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Comput.*, 33(2):37–45, Feb. 2000.

[SS96] V. Sarkar and B. Simons. Anticipatory Instruction Scheduling. In *Proc. 8th Annual ACM Symp. Parallel Algorithms and Architectures*, pp. 119–130. ACM Press, June 24–26 1996.

[SSK95] L. Schäfers, C. Scheidler, and O. Krämer-Fuhrmann. Trapper: A graphical programming environment for parallel systems. *Future Generation Computer Systems*, 11(4-5):351–361, Aug. 1995.

[ST95] D. B. Skillicorn and D. Talia, eds. *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1995.

[ST98] D. B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, June 1998.

[SU70] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17:715–728, 1970.

[Sun90] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency – Pract. Exp.*, 2(4):315–339, 1990.

[SW93] G. Sabot and S. Wholey. Cmax: a Fortran Translator for the Connection Machine System. In *Proc. 7th ACM Int. Conf. Supercomputing*, pp. 147–156, 1993.

[SWGG97] R. Silvera, J. Wang, G. R. Gao, and R. Govindarajan. A Register Pressure Sensitive Instruction Scheduler for Dynamic Issue Processors. In *Proc. PACT'97 Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society Press, Nov. 1997.

[SY97] J. Subhlok and B. Yang. A New Model for Integrated Nested Task and Data Parallel Programming. In *Proc. 6th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. ACM Press, June 1997.

[Thi92] Thinking Machines Corp. Connection Machine Model CM-5. Technical Summary. TMC, Cambridge, MA, Nov. 1992.

[THS98] O. Traub, G. Holloway, and M. D. Smith. Quality and Speed in Linear-scan Register Allocation. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 142–151, 1998.

[Tie89] M. D. Tiemann. The GNU instruction scheduler. CS343 course report, Stanford University, http://www.cygnus.com/ tiemann/timeline.html, 1989.

[TPH92] W. F. Tichy, M. Philippsen, and P. Hatcher. A Critique of the Programming Language C*. *Comm. ACM*, 35(6):21–24, June 1992.

[Tse95] C.-W. Tseng. Compiler Optimizations for Eliminating Barrier Synchronization. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 144–155. ACM Press, 1995.

[UH95] G. Utard and G. Hains. Deadlock-free absorption of barrier synchronisations. *Inform. Process. Lett.*, 56:221–227, 1995.

[Ung95] T. Ungerer. *Mikroprozessortechnik*. Thomson, 1995.

[UZSS96] M. Ujaldon, E. L. Zapata, S. Sharma, and J. Saltz. Parallelization Techniques for Sparse Matrix Applications. *J. Parallel and Distrib. Comput.*, 38(2), Nov. 1996.

[Val90] L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8), Aug. 1990.

[Veg92] S. R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*, pp. 180–188. IEEE Computer Society Press, 1992.

[VS95] R. Venugopal and Y. Srikant. Scheduling expression trees with reusable registers on delayed-load architectures. *Computer Languages*, 21(1):49–65, 1995.

[Wan93] K.-Y. Wang. A Framework for Static, Precise Performance Prediction for Superscalar-Based Parallel Computers. In H. Sips, ed., *Proc. 4th Workshop on Compilers for Parallel Computers*, Dec. 1993.

[War90] H. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. Develop.*, 34(1):85–92, Jan. 1990.

[Wel92] M. Welter. Analysen zur Übersetzung von FORK, Teil 2. Diploma thesis, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[Wil79] R. Wilhelm. Computation and Use of Data Flow Information in Optimizing Compilers. *Acta Informatica*, 12:209–225, 1979.

[Wil88] J. M. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add*. PhD thesis, Dept. of Computer Science, New York University, June 1988.

[Wil90] L. M. Wills. Automated program recognition: a feasibility demonstration. *Artificial Intelligence*, 45, 1990.

[WKE95] J. Wang, A. Krall, and M. A. Ertl. Software Pipelining with Reduced Register Requirements. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques* (*PACT*), 1995.

[WLH00] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 121–133, 2000.

[Wol94] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1994.

[WW88] B. Weisgerber and R. Wilhelm. Two Tree Pattern Matchers for Code Generation. In *Springer LNCS vol. 371*, pp. 215–229, 1988.

[YHL93] J. Yan, P. Hontalas, and S. Listgarten et al. The Automated Instrumentation and Monitoring System (AIMS) reference manual. Technical Report Memorandum 108795, NASA Ames Research Center, Moffett Field, CA, 1993.

[YWL89] C.-I. Yang, J.-S. Wang, and R. C. T. Lee. A branch-and-bound algorithm to solve the equal-execution time job scheduling problem with precedence constraints and profile. *Computers Operations Research*, 16(3):257–269, 1989.

[ZBG88] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesley, 1990.

[Zha96] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Technische Fakultät der Universität des Saarlandes, Saarbrücken (Germany), 1996.

[Zla91] Z. Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publisher, 1991.

[ZWS81] Z. Zlatev, J. Wasniewsky, and K. Schaumburg. *Y12M - Solution of Large and Sparse Systems of Linear Algebraic Equations*. Springer LNCS vol. 121, 1981.

[ZYC96] X. Zhang, Y. Yan, and R. Castaneda. Evaluating and Designing Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors. *IEEE Parallel & Distributed Technology*, 4(1):25–42, 1996.

Appendix D

List of Publications

This is a list of my publications, classified as textbooks, refereed journal papers, refereed conference papers, invited conference and workshop papers, theses, and miscellaneous publications like technical reports. Within each of these categories, the publications are sorted in chronological order.

Textbook

[B1] Jörg Keller, Christoph W. Keßler, and Jesper L. Träff. *Practical PRAM Programming.* Textbook, 596 pages. Wiley, New York, 2000.

Refereed Journal Papers

[J1] Christoph W. Keßler and Thomas Rauber. *Generating Optimal Contiguous Evaluations for Expression DAGs.* Computer Languages **21**(2), 1995.

[J2] Christoph W. Keßler. *Pattern-Driven Automatic Parallelization.* Scientific Programming **5**, pp. 251-274, 1996.

[J3] Christoph W. Keßler and H. Seidl. *The Fork95 Parallel Programming Language: Design, Implementation, Application.* International Journal on Parallel Programming **25**(1), pp. 17-50, Plenum Press, Feb. 1997.

[J4] Christoph W. Keßler. *Scheduling Expression DAGs for Minimal Register Need.* Computer Languages **24**(1), pp. 33-53, Elsevier, 1998.

[J5] Christoph W. Keßler and Jesper L. Träff. *Language and Library Support for Practical PRAM Programming.* Parallel Computing **25**: 105–135, Elsevier, 1999.

[J6] Beniamino di Martino and Christoph W. Keßler. *Two Program Comprehension Tools for Automatic Parallelization.* IEEE Concurrency **8**(1), 37–47, 2000.

[J7] Christoph W. Keßler. *NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model.* The Journal of Supercomputing, to appear (2000).

Refereed International Conference Papers

[C1] Christoph W. Keßler, Wolfgang J. Paul, and Thomas Rauber. *A randomized heuristic approach to register allocation.* Proc. of 3rd Int. Symposium on Progr. Lang. Implementation and Logic Programming (PLILP'91), Passau, Germany, Aug. 1991, Springer LNCS.

[C2] Christoph W. Keßler, Wolfgang J. Paul, and Thomas Rauber. *Scheduling Vector Straight Line Code on Vector Processors.* in: R. Giegerich and S. Graham [Eds.]: Code Generation - Concepts, Techniques, Tools. Springer Workshops in Computer Science, Springer, 1992.

[C3] Christoph W. Keßler and Thomas Rauber. *Efficient Register Allocation for Large Basic Blocks.* Proc. of 5th Int. Symposium on Progr. Lang. Implementation and Logic Programming (PLILP'93), Springer LNCS 714, 1993.

[C4] Christoph W. Keßler and Wolfgang J. Paul. *Automatic Parallelization by Pattern Matching.* Proc. of 2nd Int. ACPC Conference, Gmunden, Austria, Oct. 1993, Springer LNCS Vol. 734.

[C5] Christoph W. Keßler. *Exact Array Data Flow Analysis and Pattern Recognition in Dense Matrix Computations.* In: K.M. Decker and R. Rehmann [Eds.]: Proc. of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Systems, Verlag Birkhäuser, April 1994.

[C6] Christoph W. Keßler. *Pattern-Driven Automatic Program Transformation and Parallelization.* Proc. of 3rd Euromicro Workshop on Parallel and Distributed Processing, San Remo, Italy, Jan. 1995. IEEE CS Press.

[C7] Christoph W. Keßler and Thomas Rauber. *Optimal Contiguous Expression DAG Evaluations.* Proc. Int. Conf. on Fundamentals of Computation Theory (FCT'95), Dresden, Germany, Aug. 1995, Springer LNCS.

[C8] Christoph W. Keßler and Helmut Seidl. *Integrating Synchronous and Asynchronous Paradigms: the Fork95 Parallel Programming Language.* Proc. of MPPM-95 Conference on Massively Parallel Programming Models, Berlin, Oct. 9-12, 1995, IEEE CS press. See also: Technical Report Nr. 95-05 of University of Trier, Department for Mathematics and Computer Science, May 1995.

[C9] Beniamino di Martino and Christoph W. Keßler. *Program Comprehension Engines for Automatic Parallelization: A Comparative Study.* Proc. of 1st Int. Workshop on Software Engineering for Parallel and Distributed Systems, Chapman&Hall, March 25-26, 1996, Berlin, Germany.

[C10] Christoph W. Keßler and Jesper L. Träff. *A Library of Basic PRAM Algorithms in FORK (Research Summary).* Proc. of SPAA'96 ACM Symposium on Parallel Algorithms and Architectures, Padua, Italy, June 24-26, 1996. ACM Press.

[C11] Christoph W. Keßler. *Parallel Fourier-Motzkin Elimination.* Proc. of Euro-Par'96, Lyon, France, Aug. 1996. Springer LNCS 1124, pp. 66-71.

[C12] Christoph W. Keßler. *Scheduling Expression DAGs for Minimal Register Need.* Technical Report 96-12, Univ. Trier, FB IV Mathematik/Informatik. Proc. of 8th Int. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'96), Springer LNCS 1140, pp. 228-242, Aachen, Sept. 27-29, 1996.

[C13] Christoph W. Keßler and Jesper L. Träff. *Language and Library Support for Practical PRAM Programming.* Proc. of PDP'97 Fifth Euromicro Workshop on Parallel and Distributed Processing, London, UK, Jan. 22-24, 1997, pp. 216-221. IEEE CS Press.

[C14] Christoph W. Keßler and Helmut Seidl. *Language Support for Synchronous Parallel Critical Sections.* Proc. APDC'97 Int. Conf. on Advances in Parallel and Distributed Computing, Shanghai, China, March 19–21, 1997. IEEE CS Press.

[C15] Christoph W. Keßler. *Applicability of Program Comprehension to Sparse Matrix Computations.* Proc. of Euro-Par'97, Passau, Aug. 26-28, 1997, Springer LNCS vol. 1300.

[C16] Christoph W. Keßler and Helmut Seidl. ForkLight*: A Control–Synchronous Parallel Programming Language.* Proc. HPCN'99 High-Performance Computing and Networking, Amsterdam, Apr. 12–14, 1999, pages 525–534. Springer LNCS vol. 1593.

[C17] Christoph W. Keßler and Craig H. Smith. *The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations.* Proc. IWPC'99 Int. Workshop on Program Comprehension, Pittsburgh, May 5-7, 1999. 8 pages. IEEE CS Press. Long version: see [M11]

[C18] Christoph W. Keßler. *NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model.* Proc. PDPTA'99 Int. Conf. on Parallel and Distributed Processing Techniques and Applications Vol. II, pages 613–619. CSREA Press, June 28–July 1 1999.

Invited Conference / Workshop Papers

[I1] Christoph W. Keßler. *Pattern Recognition Enables Automatic Parallelization of Numerical Codes.* in: H.J. Sips [Ed.]: Proc. of CPC'93 4th Int. Workshop on Compilers for Parallel Computers, Delft University of Technology, the Netherlands, Dec. 13-16, 1994. pp. 385-397

[I2] Christoph W. Keßler. *The PARAMAT Project: Current Status and Plans for the Future.* Proc. of AP'95 2nd Workshop on Automatic Data Layout and Performance Prediction, CRPC-TR95548, Rice University, Houston, Apr. 1995.

[I3] Christoph W. Keßler and Helmut Seidl. *Fork95 Language and Compiler for the SB-PRAM.* Proc. of CPC'95 5th Int. Workshop on Compilers for Parallel Computers, Malaga, June 28-30, 1995.

[I4] Christoph W. Keßler and Helmut Seidl. *Language and Compiler Support for Synchronous Parallel Critical Sections.* Proc. of CPC'96 6th Int. Workshop on Compilers for Parallel Computers, Aachen, Dec. 1996. See also: Technical Report Nr. 95-23 of University of Trier, Departement for Mathematics and Computer Science, Nov. 1995.

[I5] Christoph W. Keßler. *On the Applicability of Program Comprehension Techniques to the Automatic Parallelization of Sparse Matrix Computations.* Proc. of AP'97 3rd Workshop on Automatic Data Layout and Performance Prediction, Research report of Departament d'Arquitectura de Computadors, Universitat Polytechnica de Catalunya, Barcelona, Spain, Jan. 1997.

[I6] Arno Formella and Thomas Grün and Christoph W. Keßler. *The SB-PRAM: Concept, Design and Construction.* Proceedings of MPPM-97 3rd Int. Conference on Massively Parallel Programming Models, London, Nov. 1997. IEEE Computer Society Press, 1998.

[I7] Christoph W. Keßler. *Applicability of Automatic Program Comprehension to Sparse Matrix Computations.* Proc. of CPC'98 7th Int. Workshop on Compilers for Parallel Computers, Linköping (Sweden), pp. 218-230, July 1998.

[I8] Christoph W. Keßler. *NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model.* (see also: [C18]) Proc. of CPC'00 8th Int. Workshop on Compilers for Parallel Computers, Aussois (France), pp. 13-19, Jan. 2000.

Diploma and PhD Thesis

[D1] Christoph W. Keßler. *Code–Optimierung quasi–skalarer vektorieller Grundblöcke für Vektorrechner.* Diploma thesis, 65 pages, Universität des Saarlandes, Saarbrücken, Germany, 1990.

[D2] Christoph W. Keßler. *Automatische Parallelisierung Numerischer Programme durch Mustererkennung.* Ph.D. dissertation, 200 pages, Universität des Saarlandes, Saarbrücken, Germany, 1994.

Miscellaneen

[M1] Christoph W. Keßler. *Knowledge-Based Automatic Parallelization by Pattern Recognition.* in: C.W. Keßler [Ed.]: Automatic Parallelization - New Approaches to Code Generation, Data Distribution, and Performance Prediction, Verlag Vieweg, Wiesbaden, 1994.

[M2] Christoph W. Keßler [Editor]. *Automatic Parallelization - New Approaches to Code Generation, Data Distribution, and Performance Prediction.* Book (221 pages, softcover), Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden 1994, ISBN 3-528-05401-8.
Based on the Proc. of AP'93 First Int. Workshop on Automatic Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction, held in March 1-3,1993, at Saarbrücken, Germany.

[M3] Christoph W. Keßler. *Pattern-Driven Automatic Parallelization, Data Distribution, and Performance Prediction.* in: Poster contributions at CONPAR'94, Technical report No. 94-48 of RISC Linz, Austria, Sept. 1994. pp. 17-20

[M4] Christoph W. Keßler and Helmut Seidl. *Making* FORK *Practical.* Technical Report 01/95, Sonderforschungsbereich 124 VLSI–Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1995.

[M5] Christoph W. Keßler. *Automatische Parallelisierung.* Vorlesungsskript (Course script), University of Trier, Department for Mathematics and Computer Science, SS 1995.

[M6] Christoph W. Keßler. *Scheduling Expression DAGs for Minimal Register Need.* Technical Report 96-12, Univ. Trier, FB IV Mathematik/Informatik.

[M7] Christoph W. Keßler. *Parallel Fourier-Motzkin Elimination.* Manuscript, University of Trier, Department for Mathematics and Computer Science, Feb. 1997.

[M8] Christoph W. Keßler. *Practical PRAM Programming in Fork95 - A Tutorial.* Technical Report No. 97-12, University of Trier, Department for Mathematics and Computer Science, 62 pages, May 1997.

[M9] Beniamino di Martino and Christoph W. Keßler. *Two Program Comprehension Tools for Automatic Parallelization: A Comparative Study.* Technical Report No. 97-23, University of Trier, Department for Mathematics and Computer Science, 23 pages, Nov. 1997.

[M10] Christoph W. Keßler and Helmut Seidl. ForkLight: *A Control-Synchronous Parallel Programming Language.* Technical Report No. 98-13, University of Trier, Department for Mathematics and Computer Science, 19 pages, Sep. 1998.

[M11] Christoph W. Keßler and Helmut Seidl and Craig H. Smith. *The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations.* Technical Report No. 99-10, University of Trier, Department for Mathematics and Computer Science, 21 pages, March 1999. Short version: see [C17]

Index