Integrating synchronous and asynchronous paradigms: the Fork95 parallel programming language

Christoph W. Keßler   Helmut Seidl

Fachbereich IV – Informatik, Universität Trier
D-54286 Trier, Germany
e-mail: kessler@ti.uni-trier.de

Abstract

The SB-PRAM is a lock-step-synchronous, massively parallel multiprocessor currently being built at Saarbrücken University, with up to 4096 RISC-style processing elements and with a (from the programmer's view) physically shared memory of up to 2GByte with uniform memory access time.

Fork95 is a redesign of the PRAM *language* FORK, *based on* ANSI C, *with additional constructs to create parallel processes, hierarchically dividing processor groups into subgroups, managing shared and private address subspaces. Fork95 makes the assembly-level synchronicity of the underlying hardware available to the programmer at the language level. Nevertheless, it provides comfortable facilities for locally asynchronous computation where desired by the programmer.*

We show that Fork95 offers full expressibility for the implementation of practically relevant parallel algorithms. We do this by examining all known parallel programming paradigms used for the parallel solution of real–world problems, such as strictly synchronous execution, asynchronous processes, pipelining and systolic algorithms, parallel divide and conquer, parallel prefix computation, data parallelism, etc., and show how these parallel programming paradigms are supported by the Fork95 language and run time system.

1 Introduction

It seems to be generally accepted that the most convenient machines to write parallel programs for, are synchronous MIMD (*M*ultiple *I*nstruction *M*ultiple *D*ata) computers with shared memory, well–known to theoreticians as PRAMs (i.e., *P*arallel *R*andom *A*ccess *M*achines). Although widely believed to be impossible, a realization of such a machine in hardware, the SB-PRAM, is undertaken by a project of W.J. Paul at Saarbrücken [1, 21]. The shared memory with random access to any location in *one* CPU cycle by *any* processor (PRIORITY-CRCW-PRAM) allows for a fast and easy exchange of data between the processors, while the common clock guarantees deterministic and, if desired, lock-step-synchronous program execution. Accordingly, a huge number of algorithms has been invented for this type of architecture in the last two decades. Surprisingly enough, not many attempts have been made to develop languages which allow both the convenient expression of algorithms and generation of efficient PRAM–code for them.

One approach of introducing parallelism into languages consists in decorating sequential programs meant to be executed by ordinary processors with extra primitives 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. While PVM is based on CSP [25], and therefore better suited for distributed memory architectures, the P4 library and its relatives support various concepts of parallel programming. The most basic primitives it provides for shared memory, are semaphores and locks. Moreover, it provides shared storage allocation and a flexible monitor mechanism including barrier synchronization [5, 6]. This approach is well suited if the computations executed by the different threads of the program are "loosely coupled", i.e., 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 so. Attempts to design synchronous languages have been made for the data–parallel programming paradigm. This type of computation frequently arises in numerical computations. It mainly consists in the parallel execution of iterations over large arrays. Data parallel imperative languages have been designed especially to program SIMD (*S*ingle *I*nstruction *M*ultiple *D*ata) computers like, e.g., pipelined vector processors or the CM2. Examples of such languages are C* [29] or its relatives Dataparallel C [15] and DBC [30]. The limitations of these languages, however, are obvious. There is just one global name space. Other programming paradigms like parallel recursive divide–and–conquer as suggested in [3, 9, 10, 16, 17] are not supported.

The only attempt we are aware of which allows both parallely recursive and synchronous programming are the imperative parallel languages FORK [14] and ll [27]. Based on a subset of Pascal (no jumps), ll controls parallelism by means of a parallel *do*–loop which 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 over the available tasks. Given fixed sized machines, the latter approach seems better suited to exploit the processor resources.

The design of FORK [14] was a rather theoretical one: Pointers, dynamic arrays, nontrivial data types and non–structured control flow were sacrificed to facilitate correctness proofs. In this way, however, the language became completely unusable. — In order to provide a full–fledged language for real use, we have added all the language features which are well–known from sequential programming. Thus, the new FORK dialect *Fork95* has become (more or less) a superset of C. To achieve this goal we decided to extend the ANSI-C syntax — instead of clinging to the original one. Which also meant that (for the sequential parts) we had to adopt C's philosophy. We introduced the possibility of locally asynchronous computation to save synchronization points and to enable more freedom of choice for the programming model. Furthermore, we have abandoned the tremendous run time overhead of virtual processor emulation by limiting the number of processes to the hardware resources, resulting in a very lean code generation and run time system.

Fork95 offers two different programming modes: the synchronous mode (which was the only one in old FORK) and the asynchronous mode. Each function is classified as either synchronous or asynchronous. In synchronous mode, processors form groups that can be recursively subdivided into subgroups, forming a tree–like hierarchy of groups. Shared variables and objects exist once for the group that created them; private variables and objects exist once for each processor. All processors within a group operate synchronously. In the asynchronous mode, the Fork95 run-time library offers important routines for various kinds of locks, semaphores, barriers, self–balancing parallel loops, and parallel queues, which are required for comfortable implementation of asynchronous algorithms. Carefully chosen defaults allow for inclusion of existing sequential ANSI C sources without any syntactical change.

We will show that Fork95 offers full expressibility for the implementation of practically relevant parallel algorithms. We do this by examining all known parallel programming paradigms used for the parallel solution of real–world problems, and show how these are supported by the Fork95 language and run time system.

2 The SB-PRAM from the programmer's view

The SB-PRAM [1] is a lock-step-synchronous, massively parallel MIMD multiprocessor currently under construction at Saarbrücken University, with up to 4096 RISC-style processing elements with a common clock and with a physically shared memory of up to 2GByte. The memory access time is uniform for each processor and each memory location; it takes one CPU cycle (i.e., the same time as one integer or floating-point operation) to store and two cycles to load a 32 bit word. This ideal behaviour of communication and computation has been achieved by several architectural clues like hashing, latency hiding, "intelligent" combining network nodes etc. Furthermore, a special node processor chip [21] had to be designed.

Each processor works on the same node program (SPMD programming paradigm). The SB-PRAM offers a private address space for each node processor which is embedded into the shared memory. Each processor has 30 general–purpose 32-bit registers. In the present prototype, all standard data types (also characters) are 32 bit wide. Double–precision floatingpoint numbers are not supported by hardware so far. The instruction set is based on that of the Berkeley-RISC-1 but provides more arithmetic operations[1] including integer multiplication and base-2-logarithm. Usually, these are three–address–instructions (two operands and one result). Arithmetic operations can only be carried out on registers. — The SB-PRAM offers built–in parallel multiprefix operations for integer addition, maximization, logical `and` and `or` which also take only two cycles.

Because of its architectural properties, the SB-PRAM is particularly suitable for the implementation of irregular numerical computations, non-numerical algorithms, and database applications.

Since the SB-PRAM hardware is not yet available (a 32-PE-prototype is currently being tested; the full extension is expected for 1996), we use a simulator that allows to measure exact program execution times.

We would like to emphasize that the SB-PRAM is indeed the physical realization of a PRIORITY–CRCW–PRAM, the strongest PRAM model known in theory. What the SB-PRAM cannot offer, of course, is unlimited storage size, unlimited number of processors, and unlimited word length – which however, are too ideal resource requirements for any physically existing computer.

3 Fork95 language design

Fork95 is based on ANSI C [2]. Additionally, it offers constructs to create parallel processes, to hierarchically divide groups of processors into subgroups, to manage shared and private address subspaces. Fork95 makes the assembly-level synchronicity of the underlying hardware available to the programmer. It further enables direct access to the hardware-supplied multiprefix operations.

3.1 Shared and private variables

The entire shared memory of the PRAM 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, variables are classified either

[1] Unfortunately, division for integer as well as for floating-point numbers has to be realized in software.

as private (`pr`, this is the default) or as shared (`sh`), where "shared" always relates to the processor group that defined that variable. Private objects exist once in each processor's private address subspace, whereas shared objects exist only once in the shared memory subspace of the processor group that declared them.

There is a special private variable `$` which is initially set to `__PROC_NR__` and a special shared variable `@`. `@` is meant to hold the current processor group ID, and `$` the current group–relative processor ID, during program execution. These variables are automatically saved and restored at group forming operations. However, the user is responsible to assign reasonable values to them (e.g., at the `fork` instruction).

An expression is private if it is not guaranteed to evaluate to the same value on each processor. We usually consider an expression to be private if a private subexpression (e.g., a variable) may occur in it.

If several processors write the same (shared) memory location in the same cycle, the processor with least `__PROC_NR__` will win[2] and write its value (PRIORITY–CRCW-PRAM). However, as several other write conflict resolution schemes (like ARBITRARY) are also used in theory, meaningful Fork95 programs should not be dependent on such specific conflict resolution schemes; there are better language elements (multiprefix instructions, see below) that cover practically relevant applications for concurrent write.

3.2 Synchronous and asynchronous regions in a Fork95 program

Functions are classified to be either synchronous (`sync`) or asynchronous (`async`). `main()` is asynchronous by default.

Initially, all processors of the PRAM partition 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 by calling function `main()`.

The statement `start(`e`) stmt;` whose shared expression e evaluates to some integer value k, means that k processors synchronize and execute `stmt` simultaneously and synchronously, with unique processor IDs `$` numbered successively from 0 to $k-1$. If the expression e is omitted, then all available processors executing this program are started.[3] If the value of e exceeds the number of available processors, a run-time error occurs, and the program aborts.

The `start` statement, only permitted in asynchronous mode, switches to synchronous mode for its body `stmt`. In synchronous mode, in turn, it is always possible to switch to asynchronous mode for the body of a `farm` statement:

```
farm <statement>
```

Within the `farm` body, any synchronization is suspended; at the end of a `farm` environment, the processors synchronize explicitly within their current group.

To maintain this static classification of code into synchronous and asynchronous regions, within an asynchronous region, can be called. In the other way, calling an `async` function from a synchronous region results in an implicit entering of the asynchronous mode; the programmer receives a warning. Using `farm` within an asynchronous region is superfluous and may even introduce a deadlock (a warning is emitted).

Currently we allow only one level of `start`, i.e. the synchronous regions of a program are contiguous. A generalization of `start` that allows dynamic nesting of `start` ... `farm` ... `start` ... is planned.

3.3 The group concept

At each point of program execution in synchronous mode, Fork95 maintains the invariant that all processors belonging to the same active processor group are operating strictly synchronously, i.e., 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" objects exist once for each group allocating them.

At the beginning, the started processors form one single processor group. However, it may be possible that control flow diverges at branches whose conditional depends on private values. To guarantee the above invariant, the current group must then be split into subgroups and maintaining the invariant only within each of the subgroups.

Shared `if` or loop conditions do not affect the synchronicity, as the branch taken is the same for all processors executing it.

At an `if` statement, a (potentially) private condition causes the current processor group to be split into two subgroups: the processors for which the condition evaluates to true form the first child group and execute the `then` part while the remaining processors execute the `else` part. The available shared address space of the parent group is subdivided among the new child groups before the splitting. When all processors finished the execution of the `if` statement, the two subgroups are merged again by explicit synchronization of all processors of the parent group. A similar subgroup construction is required also at loops with private exit condition. All processors that will execute the first iteration of the loop enter the child group and stay therein as long as they iterate. However, at loops it is not necessary to split the parent group's shared memory subspace, since processors that leave the loop body are just waiting at the end of the loop for the

[2] The Fork95 programmer has the possibility to change `__PROC_NR__` during program execution and thus to influence the write conflict resolution method within some limits.

[3] The present implementation allows a `start` statement to occur only at the top level of the program, i.e. it should not occur inside a loop, and there should not more than one `start` be active at the same time.

last processors of their (parent) group to complete loop execution.

Subgroup construction can, in contrast to the implicit construction at the private `if`, also be done explicitly, by the `fork` statement. Executing

`fork (`e_1`; `e_2`; `e_3`) <statement>`

means the following: First, the shared expression e_1 are evaluated to the number of subgroups to be created. Then the current leaf group is split into that many subgroups. Evaluating e_2, every processor determines the number of the newly created leaf group it will be member of. Finally, by evaluating e_3, the processor can readjust its current processor number within the new leaf group. Note that empty subgroups (with no processors) are possible; an empty subgroup's work is immediately finished, though. It is on the user's responsibility that such subgroups make sense. Continuing, we partition the parent group's shared memory subspace into that many equally–sized slices and assign each of them to one subgroup, such that each subgroup has its own shared memory space. Now, each subgroup continues on executing `<statement>`; the processors within each subgroup work synchronously, but different subgroups can choose different control flow paths. After the body `<statement>` has been completed, the processors of all subgroups are synchronized; the shared memory subspaces are re–merged, the parent group is reactivated as the current leaf group, and the statement following the `fork` statement is executed synchronously by all processors of the group.

Thus at each point of program execution, the processor groups form a tree–like hierarchy: the starting group is the root, whereas the currently active groups are the leaves. Only the processors within a leaf group are guaranteed to operate strictly synchronously. Clearly, if all leaf groups consist of only one processor, the effect is the same as using the asynchronous context. However, the latter avoids the expensive time penalty of continued subgroup formation and throttling of computation by continued shared memory space fragmentation.

3.4 Pointers and heaps

Fork95 offers pointers, as opposed to its predecessor `FORK`. The usage of pointers in Fork95 is as flexible as in C, since all private address subspaces have been embedded into the global shared memory of the SB-PRAM. Thus, shared pointer variables may point to private objects, and vice versa. The programmer is responsible for such assignments making sense.

Fork95 supplies two kinds of heaps: a shared heap and one private heap for each processor. While space on the private heaps can be allocated by the private (asynchronous) `malloc` function known from C, space on the shared heap is allocated temporarily using the shared (synchronous) `shalloc` function. The life range of objects allocated by `shalloc` is limited to the life range of the group in which that `shalloc` was executed. Thus, such objects are automatically removed if the group allocating them is released. Supplying a third variant, a "permanent" version of `shalloc`, is an issue of future Fork95 library programming.

Pointers to functions are also supported. However, special attention must be paid when using private pointers to functions in a synchronous context. Since each processor may then call a different function (and it is statically not known which one), calling a function using a private pointer in synchronous context would correspond to a huge switch, opening a separate subgroup for each function possibly being called — a tremendous waste in shared memory space! For this reason, calls to functions via private pointers automatically switch to the asynchronous mode if they are located in synchronous context. Private pointers may thus only point to `async` functions.

3.5 Multiprefix instructions

The SB-PRAM supports powerful built-in multiprefix instructions which allow the computation of multiprefix integer addition, maximization, `and` and `or` for up to 4096 processors within 2 CPU cycles. We have made available these machine instructions as Fork95 operators (atomic expression operators, not functions). Clearly, these should only be used in synchronous context. The order of the processors within a group is determined by their hardcoded absolute processor ID `__PROC_NR__`. For instance, the instruction

```
k = mpadd( &shmemloc, expression );
```

first evaluates `expression` locally on each processor participating in this instruction into a private integer value e_j and then assigns on the processor with i–th largest `__PROC_NR__` the private integer variable `k` to the value $e_0 + e_1 + \ldots + e_{i-1}$. `shmemloc` must be a shared integer variable. After the execution of the `mpadd` instruction, `shmemloc` contains the global sum $\sum_j e_j$ of all participating expressions. Thus, `mpadd` can as well be "misused" to compute a global sum by ignoring the value of `k`.

Unfortunately, these powerful instructions are only available for integer computations, because of hardware cost considerations. Floatingpoint variants of `mpadd` and `mpmax` clearly would have been of great use in parallel linear algebra applications [22].

3.6 Useful macros

The following macro from the `<fork.h>` header may be used as well in synchronous as in asynchronous context in order to enhance program understandability:

```
#define forall(i,lb,ub,p) \
    for(i=$+(lb);i<(ub);i+=p)
```

Thus,

```
gs = groupsize();
forall(i,lb,ub,gs) <statement>
```

executes `<statement>` within a parallel loop with loop variable `i`, ranging from `lb` to `ub`, using all processors belonging to the current leaf group, if suitable indexing `$` successively ranging from 0 to `groupsize()-1` has been provided by the programmer. In asynchronous context, this is also possible as long as the programmer guarantees for all required processors to arrive at that statement.

4 Compilation issues of Fork95

To compile Fork95 programs, we first install a shared stack in each group's shared memory subspace, and a private stack in each processor's private memory subspace. A shared stack pointer `sps` and a private stack pointer `spp` are permanently kept in registers on each processor. When calling a synchronous function, a shared procedure frame is allocated on the group's shared stack if the callee has shared arguments or shared local variables. An asynchronous function never has a shared procedure frame.

4.1 Group frames and synchronization

To keep everything consistent, the compiler builds shared and private group frames at each group–forming statement.

A shared group frame 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 synchronization point, each processor decrements this cell by a `mpadd(..,-1)` instruction, and waits until it sees a zero in the synchronization cell. Thereafter the processors are desynchronized by at most 2 clock cycles. After correcting this, the synchronization cell is restored to its original value. The overhead of this synchronization routine is only 10 clock cycles.

The corresponding private group frame is allocated on each processor's private memory subspace. It mainly contains the current values of the group ID `@` and the group–relative processor ID `$`. Private loops only build a shared group frame for the group of iterating processors.

Intermixing procedure frames and group frames on the same stack is not harmful, since subgroup-creating language constructs like private `if` and `fork` are always properly nested within a function. Thus, separate stacks for group frames and procedure frames are not required, preserving scarce memory resources from additional fragmentation.

4.2 Pointers and heaps

The private heap is installed at the end of the private memory subspace of each processor. For each group, its shared heap is installed at the end of its shared memory subspace. The pointer `eps` to its lower boundary is saved at each subgroup–forming operation which 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`.

4.3 Implementation

A prototype compiler for `Fork95` has been implemented. It is partially based on `lcc 1.9`, a one–pass `ANSI C`–compiler developed by C. Fraser and D. Hanson [11, 12]. [23]. gives a more detailed description of the compiler and shows that the overheads introduced by the different constructs of the language are quite low.

The compiler generates assembler code which is then processed into object code in COFF format. The SB-PRAM–linker `plink` produces executable code that runs on the SB-PRAM–simulator `pramsim` but should also run on the SB-PRAM as well once it is available. A window–based source level debugger for Fork95 is currently in preparation. — Extending the functionality of asynchronous context programming, we are also working on a set of routines for self–balancing parallel loops and parallel queues [28].

5 Parallel programming paradigms supported by Fork95

For synchronous shared memory parallel environments, several models for parallel programming models are widely accepted and could be incorporated into imperative parallel programming languages:

- *strictly synchronous execution*: This is the standard PRAM programming style. The programmer can rely on a fixed execution time for each operation which is the same for all processors at any time of program execution. Thus, no special care has to be taken to avoid race conditions because these should not occur (unless explicitly desired, as in the ARBITRARY CRCW PRAM model).
- *farming*: Several slave processes are spawned and work independently on their local tasks. They do not communicate nor synchronize with each other during their tasks.
- *pipelining* and systolic algorithms: Several slave processes are arranged in a logical network of stages which solve subproblems and propagate their partial solutions to subsequent stages. The network stepwise computes the overall solution by feeding the input data into it one by another. The topological structure of the network is usually a line, grid, or a tree, but may be any directed graph (usually acyclic). The time to execute one step of the pipeline is determined by the maximum execution time of a stage.
- *divide and conquer*: The problem and the processor set working on it is recursively divided into subsets, until either the subproblem is trivial or the processor subset consists of only one processor. The partial solutions are computed and combined when returning through the recursion tree.

- *data parallelism*: The same arithmetical operation is executed simultaneously on different data, usually disjoint sections of an array. Execution need not be synchronous, unless data dependencies may be affected. Typically, data parallelism is exploited by using a parallel loop. Array syntax, as in Fortran 90, can be used to abbreviate dataparallel operations on arrays. Sometimes, reduction operations like global sum of array elements, are also considered as dataparallel operations and supported by many dataparallel programming languages like APL [19], Fortran 90 and its successors, and dataparallel C dialects.

- *geometric parallelism*: Each slave process works on a subproblem of equal size and computational complexity. Boundary values are to be exchanged between the processors in regular time intervals. This scenario, which often occurs in scientific applications, e.g., at spatial PDE discretization, could easily profit from synchronous execution in order to save overhead due to explicit synchronization before boundary exchange. This is a special case of data parallelism.

- *asynchronous sequential processes with partial synchronization*: Most of the time, each slave process works asynchronously and independently from the other ones; now and again, however, some data dependences between processes must be taken into account. Such computations usually are arranged using locks for mutual exclusion from shared resources, and by semaphores or barriers to guarantee data dependencies. A well–known parallel programming language following this paradigm is OCCAM [20] based on CSP [18].

- *tuple space*: This is a programmer–friendly implementation of the previous item. It is realized in the LINDA language [7, 8].

- *parallel prefix*: Parallel Prefix computes for a given array $A[0 : n-1]$ and a given binary associative operator $\oplus$ the array $B[0 : n-1]$ with $B[i] = \bigoplus_{j<i} A[j]$ using an $O(\log n)$ algorithm [26] on n processors. This is rather a low–level programming paradigm and should be provided as a basic operator ("scan primitive") in a parallel programming environment. Global sum, or, and, max and similar reductions are a special case of parallel prefix computation. Parallel prefix offers fast solution of recurrence equations [24]. Nevertheless, many parallel algorithms, also nonnumerical ones like sorting, can be formulated using parallel prefix operators as basic building blocks [4]. Furthermore, atomic built–in multiprefix operators support atomic *fetch&op* primitives [13].

- *message passing* is not required in a shared memory environment. Nevertheless, any message-passing program could be transformed into an asynchronous shared–memory program.

We show that Fork95 supports all these parallel programming paradigms at the same time. We will also see that it is not necessary to extend the current language definition by additional constructs to enable usage of these paradigms.

Strictly synchronous execution is the usual mode we are applying within the synchronous part of a Fork95 program. As indicated in the last section, this maps quite directly to the underlying hardware.

Farming can be achieved in asynchronous mode within the `farm` body with no additional overhead. Farming is, clearly, also possible in synchronous mode, at the expense of subgroup creation at each private conditional, but there is no reason why farming should be done in synchronous mode because the single tasks are independent of each other. If farming is the only variant of parallelism occurring in the program, the processes can be spawned using the `start` statement.

Pipelining through an arbitrary graph can be implemented in a rather straightforward manner:

```
/*Pipeline graph consisting of n nodes:*/
struct Node {    Data *data;
                 int *pre;
                 int stage;      }
sh struct Node graph[n];
sync void init_graph(); /*initializes nodes*/
sync void work(); /*specif. work to be done*/
/*Execution of the pipeline with n proc's:*/
sh int t;
init_graph();
for(t = 0; t < end; t++)
   if (t >= graph[$].stage)
      work();
```

The data for every node of the graph through which the data are piped are grouped in structure `Node`. This structure contains a pointer to the local data; a pointer to the vector of predecessors in the graph together with the integer component `stage` containing the number of the round in which the node is going to be activated. All nodes together are grouped within the vector `graph`. For simplicity, let us assume that the n node pipeline is executed by exactly n processors. Then, besides the data structures `Data`, the programmer must provide the functions `init_graph()` and `work()`:

Processor j executing `init_graph()` initializes the entries of node `graph[j]`. For this, it especially needs to compute the predecessors of node j in the graph. Finally, the value of `stage` must be computed. In case the graph is acyclic, one possibility for this might be:

```
graph[$].stage = -1;    /*initialize stage*/
for(t = 0; t < depth; t++)
   if (graph[$].stage < 0
       && non_neg(graph[$].pre))
      /*value of all predecessors computed*/
      graph[$].stage = t;
```

Initially, all `stage` entries are initialized with -1. The stage is determined as the number t of the first iteration where all predecessors already obtained values ≥ 0 while the current stage still equals -1.

`work()` specifies the operation to by executed by processor j at node j. Input data should be read from the data entries of the nodes `graph[`i`]` where i is a predecessor of j.[4]

It may happen, though, that the numbers of processors and nodes *do not* match. A reason might be that we would like to dedicate more than one processor to each node, or too few processors are available for the graph. To handle these cases we modify our generic algorithm as follows:

```
sh int t;
init_graph();
for(t = 0; t < end; t++)
   fork(n; select(t); rename())
      if (t >= graph[@].stage)
         work();
```

Now a new group is created for every node in the graph. At the beginning of iteration t, each processor selects the node in whose group it wants to be member of. Thus, the number of this node can be accessed through the group number `@`. At the end of `work()`, the groups are removed again to allow for a synchronization of all processors in the pipeline and a redistribution at the beginning of the next iteration.

Divide-and-conquer is a natural component of the synchronous mode of Fork95. A generic divide–and–conquer algorithm `DC` may look as follows:

```
void DC(sh int n; ...)
{ if (trivial(n))
     conquer(n, ...);
  else {
     sh int d = sqrt(n);
     fork(d; @=$%d; $=$/d) {
        DC(d, ...);
        combine(n, ...);
} }  }
```

If the size n of the given problem is small enough, a special routine `conquer()` is called. Otherwise, the present group is subdivided into a suitable number of subgroups of processors (in this case, `sqrt(n)` many) each one responsible for the parallel and synchronous solution of one of the subproblems. After their solution, the leaf groups are removed again, and all prosessors of the original group join together to synchronously combine the partial results.

The last section has shown that the compile-time overhead to manage this type of programs is quite low. As an example, a parallel implementation of Strassen's algorithm for matrix multiplication has been included into directory `examples` of the Fork95 distribution. It contains two instances of `DC`, using a `fork` subdividing into seven subgroups as well as of a `fork` subdividing into two subgroups.

[4]Note that this generic implementation both covers pipelining through multidimensional arrays as used by systolic algorithms and all sorts of trees for certain combinatorial algorithms.

Data parallelism is exploitable both in synchronous and in asynchronous mode. As shown in the previous section, we supply macros for parallel loops. A set of routines that provide a self–balancing parallel loop for the asynchronous mode is currently in preparation. `mpadd()` provides a fast reduction operator for integer arrays.

Geometric parallelism: see data parallelism.

Asynchronous sequential processes are available by the `farm` statement and asynchronous functions. The Fork95 library contains all required functions for locks, mutual exclusion, barrier synchronization, semaphores and parallel queues. It should be no problem to support equivalents of the basic *tuple space* operators of LINDA by corresponding Fork95 routines and macros. In contrast to distributed–memory implementations of LINDA, this would result in more predictable execution times for the tuple space operators.

Parallel prefix is directly supported for integer operands and the operators `add`, `max`, `and`, and `or`, since Fork95 makes the corresponding SB-PRAM-operators accessible as atomic operators at the language level. Generalization to arrays of arbitrary size with linear speedup is straightforward (see the appendix). Unfortunately, the SB-PRAM designers renounced to support such powerful operators also for floatingpoint operands. Thus, Fork95 must implement these in the usual way (time: $O((n/p)\log n)$).

6 Availability of the compiler

The Fork95 compiler including all sources is available from `ftp.informatik.uni-trier.de` in directory `/pub/users/Kessler` by anonymous ftp. This distribution also contains documentation, example programs and a preliminary distribution of the SB-PRAM system software tools including assembler, linker, loader and simulator. The Fork95 documentation is also available by www via the URL `http://www-wjp.cs.uni-sb.de/ fork95/index.html`.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On Physical Realizations of the Theoretical PRAM Model. Technical Report 21/1990, SFB 124 VLSI Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1990.

[2] American National Standard Institute, Inc., New York. American National Standards for Information Systems, Programming Language C. ANSI X3.159–1989, 1990.

[3] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, S. Saxena, and T. Radzik. Improved Deterministic Parallel Integer Sorting. *Information and Computation*, 94, 1991.

[4] G. Blelloch. Scans as Primitive Parallel Operations. *IEEE Trans. Comput.*, 38(11):1526–1538, Nov. 1989.

[5] R. Butler and E.L. Lusk. User's Guide to the P4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[6] R. Butler and E.L. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Journal of Parallel Computing*, 20(4):547–564, April 1994.

[7] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Comp. Surv.*, 21(3):323–357, 1989.

[8] 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, April 1994.

[9] M.I. Cole. *Algorithmic Sceletons: Structured Management of Parallel Computation.* Pitman and MIT Press, 1989.

[10] 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.

[11] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software—Practice & Experience*, 21(9):963–988, Sept. 1991.

[12] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

[13] A. Gottlieb, B. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Large Numbers of Cooperating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 5(2), April 1983.

[14] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A High–Level Language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[15] P.J. Hatcher and M.J. Quinn. *Dataparallel Programming on MIMD Computers.* MIT-Press, 1991.

[16] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. Part I: The Model. *Journal of Parallel and Distributed Programming*, 16:212–232, 1992.

[17] —, Part II: Binary Tree and FFT Algorithms. *Journal of Parallel and Distributed Programming*, 16:233–249, 1992.

[18] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall Int. Series in Computer Science, 1985.

[19] K. Iverson. *A Programming Language.* Wiley, 1962.

[20] G. Jones and M. Goldsmith. *Programming in Occam 2.* Prentice-Hall, 1988.

[21] J. Keller, W.J. Paul, and D. Scheerer. Realization of PRAMs: Processor Design. In *WDAG94, 8th Int. Worksh. on Distr. Algorithms, Springer LNCS 857*, 1994.

[22] C.W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, 1994.

[23] C.W. Keßler and H. Seidl. Fork95 Language and Compiler for the SB-PRAM. 5th Int. Worksh. Compilers for Par. Computers, 1995. `http:// www-wjp.cs.uni-sb.de/fork95.html`.

[24] P. Kogge and H. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Trans. Comput.*, C-22(8):786–793, August 1973.

[25] Oak Ridge National Laboratory and University of Tennessee. Parallel Virtual Machine Reference Manual, Version 3.2. Technical report, August 1993.

[26] R.E. Ladner and M.J. Fisher. Parallel Prefix Computations. *Journal of the ACM*, 27(4):831–838, 1980.

[27] C. León, F. Sande, C. Rodríguez, and F. García. A PRAM Oriented Language. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 182–191, 1995.

[28] J. Röhrig. title to be announced (in german language). Master thesis, Universität Saarbrücken, to appear 1995.

[29] J. Rose and G. Steele. C*: An Extended C Language for Data Parallel Programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[30] J. Schlesinger and M. Gokhale. DBC Reference Manual. Techn. Rep. 92-068, Supercomp. Research Center, 1992.

A Appendix

A.1 Example: Multiprefix sum

The following routine performs a general integer multiprefix-ADD implementation in Fork95. It takes time $O(n/p)$ on a p-processor SB-PRAM with built-in `mpadd` operator running in O(1) time. This is optimal. Only one additional shared memory cell is required (as proposed by J. Roehrig). The only precondition is that group-relative processor ID's `$` must be consecutively numbered from 0 to `groupsize() - 1` (if not, this can be provided in $O(1)$ time by a `mpadd` operation).

```
sync void parallel_prefix_add(
  sh int *in,     /*operand array*/
  sh int n,       /*problem size*/
  sh int *out,    /*result array*/
  sh int initsum) /*global offset*/
{
  sh int p = groupsize();
  sh int sum = initsum; /*temp. accum. cell*/
  pr int i;
  /*step over n/p slices of array:*/
  for (i=$; i<n; i+=p)
     out[i] = mpadd( &sum, in[i] );
}
```

Run time results (in SB-PRAM clock cycles):

# processors	cc, $n = 10000$	cc, $n = 100000$
2	430906	4300906
4	215906	2150906
16	54656	538406
64	14406	135322
256	4344	34530
1024	1764	9332
4096	1162	3054

A.2 Example: Divide–and–conquer

```
/* parallel QUICKSORT from [BDH+91] */

pr int value, pos = 0;
extern sh int a[];   /*the array to be sorted*/

sync void quicksort( sh int *weight ) {
  sh int mid = value;
  sh int leftweight = 0, rightweight = 0;
  pr int left = (value<mid), right = (value>mid);
  if (value != mid)
     if (left)  quicksort( &leftweight );
     else       quicksort( &rightweight );
  if (value == mid)  pos = leftweight + 1;
  if (right)         pos += 1 + leftweight;
  *weight = leftweight + rightweight + 1;
}

main(){
  start {     /* we need as many processors
        as there are array elements to sort */
    sh int weight;
    value = a[$];
    quicksort( &weight );
    a[pos-1] = value;
} }
```