Appeared in: Int. Journal of Parallel Programming **25**(1), Feb. 1997, pages 17–49.

The Fork95 Parallel Programming Language: Design, Implementation, Application

Christoph W. Keßler Helmut Seidl
Fachbereich IV - Informatik, Universität Trier
D-54286 Trier, Germany
e-mail: {kessler,seidl}@psi.uni-trier.de

Abstract

Fork95 is an imperative parallel programming language intended to express algorithms for synchronous shared memory machines (PRAMs). It is based on ANSI C and offers additional constructs to hierarchically divide processor groups into subgroups and manage 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 supports locally asynchronous computation where desired by the programmer.

We present a one–pass compiler, `fcc`, which compiles Fork95 and C programs to the SB-PRAM machine. The SB-PRAM is a lock–step synchronous, massively parallel multiprocessor currently being built at Saarbrücken University, with a physically shared memory and uniform memory access time.

We examine three important types of parallel computation frequently used for the parallel solution of real–world problems. While farming and parallel divide–and–conquer are directly supported by Fork95 language constructs, pipelining can be easily expressed using existing language features; an additional language construct for pipelining is not required.

Key words: Fork95, parallel programming language, synchronous program execution, PRAM, parallel programming paradigms

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, 2]. The shared memory with random access to any location in *one* CPU cycle by *any* processor 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 [3], and therefore better suited for distributed memory architectures, the P4 library 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 [4], [5]. 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 MODULA-2* [6], VECTOR C [7] and C* [8] or its relatives DATAPARALLEL C [9] and DBC [10].

However, there is just one global name space supported by these languages. Other parallel computation paradigms like a parallel recursive divide–and–conquer style as suggested in [11], [12], [13], [14], [15] are not supported. On the other hand, most process–oriented languages, such as e.g. [16] or [17], do not offer a mode of strictly synchronous program execution.

The only attempt we are aware of which allows both parallely recursive and synchronous programming are the imperative parallel languages FORK [18] and *ll* [19]. 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 [18] 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. This 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. Within the 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 leaf 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.

In this article we give a comprehensive overview over the language Fork95 and an impression of

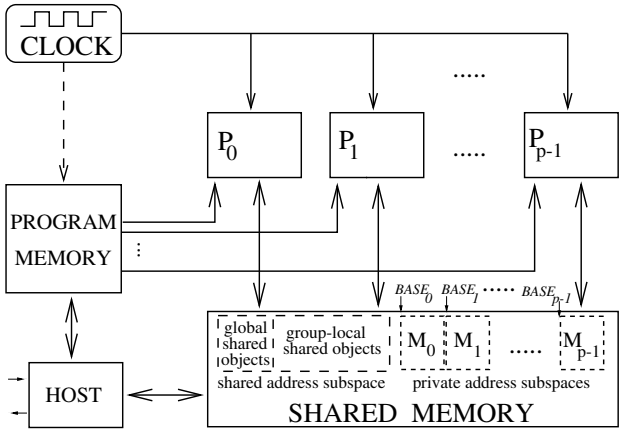

Figure 1: Block diagram of the SB-PRAM from the programmer's view.

its implementation. We examine three important parallel computation models used for the parallel solution of real–world problems, namely farming, parallel divide–and–conquer, and pipelining, and show how these are supported by the Fork95 language, its compiler, and run time system.

The rest of the paper is organized as follows. Section 2 presents the SB-PRAM architecture from the programmer's (and compiler's) point of view. Section 3 explains the basic concepts and mechanisms of Fork95 to control parallel execution. Section 4 describes the compiler and the runtime environment. Section 5 deals with Fork95's expressivity. The Appendix lists some example programs.

2 The SB-PRAM from the programmer's view

The SB-PRAM [20] is a lock-step-synchronous, massively parallel MIMD multiprocessor currently under construction at Saarbrücken University, with up to 4096 RISC-style processing elements and with a physically shared memory of up to 2GByte. Private address subspaces are embedded into the shared memory by handling private addresses relative to a BASE pointer (see Figure 1). 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; it takes one CPU cycle (i.e., the same time as one integer or floatingpoint operation) to store and two cycles to load a 32 bit word. This ideal ratio of communication to computation has been achieved by techniques like hashing, latency hiding, "intelligent" combining network nodes etc. Furthermore, a special node processor chip [2] had to be designed.

Each processor works on the same node program (SPMD mode). 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[2] for integer addition, maximization, bitwise AND and bitwise OR which also execute within two CPU cycles.

The modulo bit is a globally visible flag which is inverted after each machine cycle. Thus it can be used to separate reading and writing accesses to the same shared memory location, as required by the PRAM model.

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

[2] The functionality of the multiprefix instructions will be explained in Subsection 3.5.

Parallel I/O (to/from local hard disks) and sequential I/O (to/from the host) features have been added. A single–user, single–tasking operating system is already available, a multi–user and multi–tasking system is planned for the near future. System software (assembler, linker, loader) has been completed, as well as an asynchronous C compiler which has been extended by an implementation of the P4 parallel macro package [21, 22].

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

The current university prototype implementation provides a (of course, not very exciting) processor speed of 0.25 MFlops. However, this could be easily improved by one or even two orders of magnitude by using faster chip and interconnection technology and by exploiting the usually large potential of cycles doing only private computation. As shown in [23], it is indeed possible to reach on representative benchmarks a sustained node performance very close to that of current distributed memory multiprocessors, without sacrificing the PRAM behaviour.

Since the SB-PRAM hardware is not yet fully available[3] we use a simulator for the machine that allows to measure exact program execution times.

We would like to emphasize that the SB-PRAM is indeed a physical realization of a MULTIPREFIX–CRCW–PRAM, the strongest PRAM model suggested so far.

3 Fork95 language design

Processes (in the common sense) are executed in Fork95 by groups of processors, whereas what is called a "thread" in other programming languages most closely corresponds to an individual processor in Fork95 — the only difference being that the total number of processors in Fork95 is limited to the number of physically available PRAM processors. It has turned out that the implementational overhead to maintain *synchronously executing* virtual processors is a too high price to pay for a potential gain of comfort of programming. On the other hand, execution of an arbitrary number of *asynchronous threads* can be easily implemented in Fork95 (see Subsection 5.1).

Fork95 is based on ANSI C [24]. The new constructs handle the group organization, shared and private address subspaces, and various levels of synchronicity. Furthermore, direct access to hardware-supplied multiprefix operations is enabled.

This section is organized as follows: Subsection 3.1 introduces the concept of variables in Fork95. Subsection 3.2 deals with constructs controlling synchronous and asynchronous program execution. The concept of hierarchical processor groups is explained in subsection 3.3. Subsection 3.4 describes pointers and heaps, whereas subsection 3.5 presents the powerful multiprefix operators of Fork95. A brief overview over the new language constructs added to C can be found in Table i.

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 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.

[3] Currently, a 128 PE submachine, corresponding to 4 processor boards, is running, under a preliminary version of the SB-PRAM operating system (status at end of July 1996).

Fork95 special program variables				
name	meaning	sharity	type	remark
`__STARTED_PROCS__`	number of all available processors	`sh`	`int`	read–only
`__PROC_NR__`	physical processor ID	`pr`	`int`	read–only
`@`	current group ID	`sh`	`int`	may be redefined
`$`	current processor ID	`pr`	`int`	may be redefined

Fork95 function (pointer) type qualifiers			
name	meaning	remark	example for usage
`sync`	declare (pointer to) synchronous function		`sync int foo( sh int k ) {...}`
`async`	declare (pointer to) asynchronous function	default	`sh void (*task[Pmax])(void);`

Fork95 storage class qualifiers			
`sh`	declare a shared variable		`sh int k=1, *pk, a[32];`
`pr`	declare a private variable	default	`pr int i,*pi, b[100];`

Fork95 language constructs: statements				
name	meaning	example for usage	mode	body
`start`	all processors enter synchronous mode	`start { a[$]=$; }`	asynchr.	synchr.
`farm`	enter asynchronous mode	`farm puts("Hello");`	synchr.	asynchr.
`fork`	split current group in subgroups	`fork(3;@=$%3;$=$/3)...`	synchr.	synchr.
`barrier`	group–local barrier synchronization	`barrier;`	asynchr.	—

Fork95 language constructs: operators					
name	meaning	example for usage	mode	operands	result
`mpadd(ps,ex)`	multiprefix sum	`$=mpadd(&p,1);`	both	`int *, int`	`pr int`
`mpmax(ps,ex)`	multiprefix maximum	`mpmax(&m,a[$]);`	both	`int *, int`	`pr int`
`mpand(ps,ex)`	multiprefix bitwise AND	`k=mpand(&m,0);`	both	`int *, int`	`pr int`
`mpor(ps,ex)`	multiprefix bitwise OR	`k=mpor(&m,1);`	both	`int *, int`	`pr int`
`ilog2(k)`	floor of base 2 logarithm	`l=ilog2(k);`	both	`int`	`int`

Fork95 memory allocation routines				
name	mode	type	parameters	meaning
`malloc`	`async`	`char *`	`pr uint`	allocate block on private heap
`free`	`async`	`void`	`pr char *`	free block on private heap
`shmalloc`	`async`	`char *`	`pr uint`	allocate block on global shared heap
`shfree`	`async`	`void`	`pr char *`	free block on global shared heap
`shalloc`	`sync`	`char *`	`sh uint`	allocate block on automatic shared heap
`shavail`	`async`	`uint`	`void`	get size of free automatic shared heap space
`shallfree`	`sync`	`void`	`void`	free all blocks `shalloc`ed so far in current function

Fork95 group structure inspection				
name	mode	type	parameters	meaning
`groupsize`	`sync`	`int`	`void`	get number of processors in my current group
`async_groupsize`	`async`	`int`	`void`	same as `groupsize`, for asynchronous call sites
`parentgroupsize`	`sync`	`int`	`void`	get number of processors in my parent group

Table i: Fork95 at a glance. `uint` is an abbreviation for `unsigned int`.

The *total number of started processors* is accessible through the constant shared variable

```
__STARTED_PROCS__
```

The *physical processor ID* of each processor is accessible through the constant private variable

```
__PROC_NR__
```

The special private variable `$`[4] is intended to hold the current *group–relative processor ID*; it is initially set to the physical processor ID but may be redefined during program execution.

It is not part of the language to fix what happens if several processors write the same (shared) memory location in the same cycle. Instead, Fork95 inherits the write conflict resolution method from the target machine. In the case of the SB-PRAM, the processor with highest `__PROC_NR__` will win 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 subsection 3.5) that cover practically relevant applications for concurrent write.

The return value of a non–`void` function is always private.

3.2 Synchronous and asynchronous regions in a Fork95 program

Fork95 offers two different programming modes that are statically associated with source code regions: *synchronous mode* and *asynchronous mode*.

In synchronous mode, processors remain synchronized on the statement level and maintain the *synchronicity invariant* which says that in synchronous mode, all processors belonging to the same (active) group operate strictly synchronous, ie. their program counters are equal at each time step.

In asynchronous mode, the synchronicity invariant 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 using the `barrier` statement.

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 function `main()`.

Functions are classified as either synchronous (declared with type qualifier `sync`) or asynchronous (`async`, this is the default). `main()` is asynchronous by default. A synchronous function is executed in synchronous mode, except from blocks starting with a `farm` statement

```
farm  <stmt>
```

which enters asynchronous mode and re-installs synchronous mode after execution of `<stmt>` by a group–local exact barrier synchronization.

Asynchronous functions are executed in asynchronous mode, except from blocks starting with the `start` statement

```
start <stmt>
```

[4] In the old FORK proposal, the group–relative processor ID was denoted by `#`.

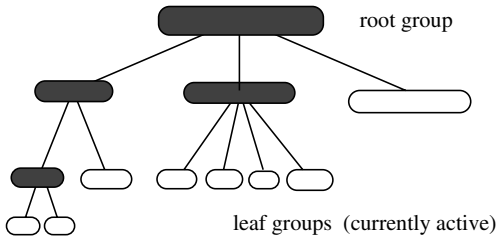

Figure 2: The group hierarchy in Fork95 forms a logical tree.

The `start` statement, only permitted in asynchronous mode, switches to synchronous mode for its body `<stmt>`. It causes all available processors to synchronize, using the exact barrier synchronization routine described in subsection 4.1, and execute `<stmt>` simultaneously and in synchronous mode, with unique processor IDs `$` numbered successively from 0 to `__STARTED_PROCS__`-1.

To maintain this static classification of code into synchronous and asynchronous regions, within an asynchronous region, only `async` functions 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).

Asynchronous functions must not allocate shared local variables. This also means that shared formal parameters for asynchronous functions are forbidden.

The present implementation supports `start` statements only at the top level of the program; there should not be more than one `start` active at the same time. Currently we are working on a generalization of `start` that allows arbitrary nesting of synchronous and asynchronous regions [25].

3.3 The group concept

Fork95 programs are executed by *groups* of processors, rather than by individual processors. Initially, there is just one group containing all available processors. Groups may be recursively subdivided. Thus, at any point of program execution, all presently existing groups form a tree–like *group hierarchy*. Only the *leaf groups* of the group hierarchy are active (see Figure 2).

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 `@`. Initially, `@` is set to 0 but may be redefined appropriately during program execution. Both the group–relative processor ID `$` and the group ID `@` are automatically saved when splitting the current group, and restored when reactivating it. However, the user is responsible to assign reasonable values to them.

At each point of program execution in synchronous mode, Fork95 maintains the synchronicity invariant which requires 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. A processor can inspect the number of processors belonging to its current group using the routine

```
sync int groupsize();
```

At the entry into a synchronous region, the 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 synchronicity within each active subgroup, the current leaf group must then be split into subgroups.

We consider an expression to be private if it is not guaranteed to evaluate to the same value on each processor, i.e. if it contains a private variable, a function call, or a multiprefix operator (see Subsection 3.5). Otherwise it is shared which means that it evaluates to the same value on each processor of the group.

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 private condition causes the current processor group to be split into two subgroups: the processors for which the condition evaluates to a non–zero value form the first subgroup and execute the `then` part while the remaining processors execute the `else` part. The current (parent) group is deactivated; its available shared address space is subdivided among the new subgroups. Each subgroup arranges a shared stack and a shared heap of its own, allowing to declare and allocate shared objects relating only to that subgroup itself. Also, the subgroups inherit their group IDs `@` and the group-relative processor IDs `$` from the parent group but may redefine them locally. When both subgroups have finished the execution of the `if` statement, they are released, and the parent group is reactivated by explicit synchronization of all its processors.

A similar subgroup construction is required also at loops with a private exit condition. Assume that the processors of a leaf group g arrive in synchronous mode at a loop statement. All processors that will execute the first iteration of the loop form a subgroup g' and stay therein as long as they iterate. 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. However, 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.

As an illustration, consider the dataparallel loop

```
sh int p = groupsize();
pr int i;
for (i=$; i<n; i+=p)  a[i] += c * b[i];
```

This causes the private variable `i` of processor k, $0 \leq k \leq p-1$, to loop through the values k, $k+p$, $k+2p$ etc. The whole index range from 0 to `n` -1 is covered provided that the involved processors have ID's `$` consecutively ranging from 0 to $p-1$.

Splitting into subgroups can, in contrast to the implicit subdivision at private branch conditions, also be done explicitly, by the `fork` statement. Executing

`fork (` e_1`; =`e_2`; $=`e_3 `) <stmt>`

means the following: First, the shared expression e_1 is 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. 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 `<stmt>`; the processors within each subgroup work synchronously, but different subgroups can choose different control flow paths. After the body `<stmt>` 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.

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. In particular, one needs not distinguish between pointers to shared and pointers to private objects, in contrast to some other parallel programming languages, for example C^* [8]. 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 which 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`. — Accordingly, if all processors execute the following statement simultaneously

```
sharedpointer = &privatevar;  /*concurrent write, result is deterministic*/
```

the shared pointer variable `sharedpointer` is made point to the private variable `privatevar` of the processor with the highest processor ID participating in this assignment. Thus, a shared pointer variable pointing to a private location nevertheless represents a shared *value*; all processors in the scope of 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.

Fork95 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 heaps can be allocated and freed by the asynchronous functions `malloc()` and `free()` known from C. Space on the permanent shared heap is allocated and freed accordingly using the asynchronous functions `shmalloc()` and `shfree()`. The automatic shared 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 synchronous `shalloc()` function 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. For better space economy, there is a synchronous function `shallfree()` which frees *all* objects `shalloc`ed so far in the current (synchronous) function.

Pointers to functions are also supported. However, special attention must be paid when using private pointers to functions in synchronous mode. 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 mode 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 regions. Private pointers may thus only point to `async` functions.

3.5 Multiprefix operators

The SB-PRAM supports powerful built-in multiprefix instructions called `mpadd`, `mpmax`, `mpand` and `mpor`, which allow the computation of multiprefix integer addition, maximization, AND and OR for up to 4096 processors within two CPU cycles. We have made available these machine instructions as Fork95 operators (atomic expression operators, not functions). These can be used in synchronous as well as in asynchronous mode.

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 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 physical processor ID `__PROC_NR__` [5]. 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

$$v_s + e_{s,0} + e_{s,1} + \ldots + e_{s,i-1}$$

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`.

For instance, determining the size p of the current group, including consecutive renumbering 0,1,...,$p-$1 of the group–relative processor ID `$`, could be done by

```
sh int p = 0;
$ = mpadd( &p, 1 );
```

where the integer variable `p` is shared by all processors of the current group.

The `mpadd` instruction executed in the course of the evaluation of a `mpadd()` expression is assumed to work atomically; on the SB-PRAM it executes within one cycle (plus one delay cycle for availability of the private result values at each processor). Thus, `mpadd()` and the other multiprefix operators can be directly used as atomic *fetch&op* primitives [26] which are very useful to access semaphores in asynchronous mode, e.g., simple locks which should sequentialize access to some shared resource (*critical section*, see e.g. [27]). Let

```
sh int l = 0;       /* simple lock */
```

be a global shared variable realizing a simple lock, where a zero value means that the lock is open, and a nonzero value means the lock to be locked. `l` is initialized to zero at the beginning of the program. Let ℓ denote the address of variable `l`. A processor leaving the critical section clears the lock:

[5] Note that this order is inherited from the hardware and does *not* depend on the current value of the user–defined processor ID `$`.

```
l = 0;
```

Waiting for access to the critical section to be guarded by `l` could be realized by the following program fragment:

```
while (mpadd( &l, 1 )) != 0) ;
```

As long as `l` is locked (i.e., v_ℓ is nonzero), the `mpadd()` expression evaluates to a nonzero value on any processor. Thus, a processor iterates until it obtains a zero from the `mpadd()` expression (i.e., it becomes $q_{\ell,0}$ in our notation used above). Since all other iterating processors (the $q_{\ell,i}$ with $i > 0$) obtain nonzero values, it is guaranteed that only one processor enters the critical section. Fork95 offers several types of locks: simple locks, safe locks, fair locks, and reader–writer locks. The routines manipulating these are implemented in SB-PRAM assembler, because they are often used in asynchronous computations.

The generalization of `mpadd()` etc. to arrays of arbitrary size n with time complexity $O(n/p)$ on p processors is straightforward (see Appendix A.1).

Beyond `mpadd()`, Fork95 also supports the multiprefix operators `mpmax()`, `mpand()` (bitwise AND) and `mpor()` (bitwise OR), which make the corresponding multiprefix instructions of the SB-PRAM directly available to the programmer at the language level. Unfortunately, the SB-PRAM hardware designers supplied these powerful multiprefix instructions only 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 [28]. For instance, parallel prefix allows for a fast solution of recurrence equations [29].

Furthermore, many parallel algorithms, also nonnumerical ones like sorting, can be implemented using parallel prefix operators as basic building blocks [30]. And last but not least, global sum, OR, AND, max, min and similar reductions are an important special case of parallel prefix computation.

Generic Fork95 multiprefix routines for any data type are provided in the PAD library [31], with time complexity $O((n/p) + \log p))$.

3.6 Caveats in Fork95 programming

3.6.1 Spaghetti jumping

All non–structured statements affecting control flow (`goto`, `longjmp`, `break`, `return`, `continue`) are dangerous within a synchronous environment since the jumping processors may not enter or leave groups on the normal way (via subgroup construction or subgroup merge).

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. In this case, the compiler can provide a safe implementation even for the synchronous mode.

For a `goto` jump, 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. Jumps across synchronization points usually will introduce a deadlock. For this reason, `goto` jumps are under the programmer's responsibility. However, as long as source and destination of a `goto` are within the same asynchronous region, there is no danger of deadlock.

3.6.2 Shared memory fragmentation

The reader may already have noticed that it is not wise to have more `fork` or private `if` statements on the recursive branch of a recursive procedure (like parallel depth–first–search, for instance) than

absolutely necessary. Otherwise, after only very few recursion steps, the remaining shared memory fraction of each subgroup has reached an impracticably small size thus resulting in early stack overflow.

If all leaf groups consist of only one processor, one should better switch to asynchronous mode because this avoids the expensive time penalty of continued subgroup formation and throttling of computation by continued shared memory space fragmentation.

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.

As in common C compilers, a procedure frame is allocated on each processor's private stack, holding private arguments (pointed to by a private argument pointer `app`), saved registers, and private local variables, pointed to by a private frame pointer `fpp`. In special cases, up to 4 private arguments can be passed in registers.

When calling a synchronous function, a shared procedure frame is allocated on the group's shared stack if the callee has shared arguments (pointed to by `aps`) or shared local variables (pointed to by `fps`). 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 (see Figure 3) 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 (i.e. the time that is consumed if the processors are already synchronous at entry) is 15 to 16 clock cycles (depending on the current value of the modulo bit); this may still be optimized to 12 cc if the compiler would inline the function and statically keep track of the modulo bit where possible.

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 `$`. Private loops only build a shared group frame for the group of iterating processors. A private group frame is not necessary as long as there are no changes of the values for `@` and `$`.[6]

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

[6] Changes to `@` and `$` inside a loop are currently not supported by the compiler.

```
.globl forklib_sync
forklib_sync: /*no parameter; uses r31, r30*/
bmc   0                 /*force next modulo 0*/
getlo -1,r31            /*load constant -1  0*/
mpadd gps,1,r31         /*decr. sync cell   1*/

FORKLIB_SYNCLOOP:
ldg   gps,1,r30         /*load sync cell    0*/
getlo 1,r31             /*load constant 1   1*/
add   r30,0,r30         /*compare with zero 0*/
bne   FORKLIB_SYNCLOOP /*until zero seen   1*/

ldg   gps,1,r30         /*load sync cell    0*/
mpadd gps,1,r31         /*repair sync cell  1*/
add   r30,0,r30         /*compare with zero 0*/
bne   FORKLIB_SYNCHRON /*late wave: bypass 1*/
nop                     /*delay early wave  0*/
nop                     /*delay early wave  1*/
FORKLIB_SYNCHRON:
return                  /*sync: finished    0*/
nop                     /*delay-slot return 1*/
```

Figure 3: The exact group–local barrier synchronization routine in `fcc`, as proposed by Jörg Keller (Saarbrücken University). A processor's current leaf group's synchronization cell, located in the shared group frame, is addressed by `gps,1`. The `bmc` instruction causes the processors to enter the `SYNCLOOP` loop only if the modulo bit is 0. Because the `SYNCLOOP` loop has length 4, all iterating processors leave the loop (as soon as they see a zero in the synchronization cell) in two waves which are separated by two machine cycles. This delay is corrected in the last part of the routine. Processors that belong to the late wave see already a number different from zero in the synchronization cell, because the processors of the early wave already incremented them. When `return`ing, all processors are exactly synchronous.

`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 Example: Translation of private `if` statements

As an illustration for the compilation of subgroup–creating constructs, we pick the `if` statement with a private condition e:

`if` (e) `statement1 else statement2`

It is translated into the following pseudocode to be executed by each processor of the current group:

(1) divide the remaining free shared memory space of the current group (located between shared stack pointer and shared heap pointer) into two equally–sized blocks B_0 and B_1
(2) evaluate e into a register reg
(3) allocate a new private group frame on the private stack; copy the old values of `$` and `@` to their new location
(4) if ($reg == 0$) goto (10)
(5) set shared stack pointer and shared heap pointer to the limits of B_0
(6) allocate a new shared group frame on that new shared stack
(7) determine current (sub)group size by `mpadd(&synccell,1)`
(8) execute `statement1`
(9) goto (14)
(10) set shared stack pointer and shared heap pointer to the limits of B_1
(11) allocate a new shared group frame on that new shared stack
(12) determine current (sub)group size by `mpadd(&synccell,1)`
(13) execute `statement2`
(14) remove shared and private group frame, restore shared stack pointer, heap pointer, and the group pointers; call the synchronization routine (Figure 3)

construct	overhead in SB-PRAM clock cycles
exact group–local barrier synchronization	$15 \leq t_{sync} \leq 16$
program startup code	$150 + 4 \times \lvert private\ .data\ section \rvert$
`start`:	$27 + t_{sync}$
synchronous loop with private exit condition	$8 + 5 \times \#iterations + t_{sync}$
synchronous `if` with private condition	$32 + t_{sync}$
`fork`	$40 + t_{division} + t_{sync}$
`farm`	t_{sync}
call to a synchronous function	$41 + \#(saved\ regs) + \#(args) + t_{sync}$
call to an asynchronous function	$10 + \#(saved\ regs) + \#(args)$
`shalloc()`	4
integer division	$12 \leq t_{division} \leq 300$ (data dependent)

Table ii: Overheads for Fork95 language constructs

Important optimizations (as [32, 33] did for the old FORK standard) will address waste of memory in the splitting step (first item). For instance, if there is no `else` part, splitting and generation of new group frames is not necessary. A private group frame is also not required if `$` and `@` are not redefined in `statement1` resp. `statement2`. If the memory requirements of one branch are statically known, all remaining memory can be left to the other branch.

In the presence of an `else` part, the synchronization could be saved if the number of machine cycles to be executed in both branches is statically known; then the shorter branch can be padded by a sequence or loop of `nop` instructions.

4.4 Translation of `break`, `continue`, and `return`

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 interjacent group (including the current leaf group), its private group frame (if existing) has to be removed, 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 re–synchronize with them.

4.5 Implementation

A first version of a compiler for Fork95 has been implemented. It is partially based on `lcc 1.9`, a one–pass `ANSI C`–compiler developed by Chris Fraser and David Hanson at Princeton, NY [34, 35, 36].

Table ii shows the overheads introduced by the different constructs of the language. Division has to be implemented in software: therefore the huge (and varying) number in the last line. Also, in a synchronous region, extra synchronization has to occur afterwards. The cost of calls clearly can be reduced by passing arguments in registers (this is standard for most library functions). The cost of `if`s can be reduced drastically whenever at most one of the branches contains function calls.

The compiler generates assembler code which is processed by the SB-PRAM–assembler `prass` into object code in COFF format. The SB-PRAM–linker 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.

4.6 Limitations of the Compiler

Currently, conditional expressions $e?l:r$ in synchronous mode do — unlike the private `if` statement — not cause splitting of the current group if e is a private condition.[7] Thus, a private condition e may introduce asynchronity if the evaluations of expressions l and r take different time on the PRAM, which may cause wrong results or even a deadlock; a warning is emitted in this case. Nevertheless, this does not restrict the programmer: he could use the `if` statement instead.

The same holds for `switch` statements in synchronous mode: Currently, there is no group splitting provided. Thus, a private selector may introduce a deadlock (warning is given). If a synchronous switch over a private selector cannot be avoided, the programmer should replace it by a (if possible, balanced) `if` cascade.

Attention must be paid if `continue` is used in a loop. If between the `continue` and the end of the loop body some synchronization will take place (e.g., at the end of a private `if`, of a private loop, of a `sync` function call or of a `farm`), a deadlock may be introduced. This problem will disappear in a future release of `fcc` by enabling the user to indicate such situations a priori by specifying a compiler option that introduces an extra shared group frame for each loop. A general implementation of `continue` is not reasonable for a one–pass–compiler like `fcc`.

The C library is not yet fully implemented, but we have provided the most important routines, e.g. for screen and file input/output, string manipulation, `qsort()`, pseudo–random number generation, and mathematical functions. Extending the functionality of asynchronous mode programming, we are also working on a set of assembler–coded primitives to handle self–scheduling parallel loops and parallel queues [21].

The following further optimizations may be useful in a future implementation but cannot be implemented in the current prototype due to the one-pass-nature of the compiler:

- Due to the possibility of multiple occurrences of `return` in a called function, the caller has to provide a shared group frame for each synchronous function call. If the callee were statically known to have just one such exit point, the overhead of creating this group frame and synchronizing after the return could be avoided.
- Opening a loop group also for a shared loop exit condition is not necessary if the loop body contains no `break` or `return` statement inside a one-sided `if` statement.

4.7 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 on the WWW via the URLs `http://www-wjp.cs.uni-sb.de/fork95/index.html` and `http://www.informatik.uni-trier.de/~kessler/fork95.html`.

5 Parallel programming paradigms supported by Fork95

[7] This is partly due to technical reasons, namely the strange construction of expression DAGs in the `lcc`.

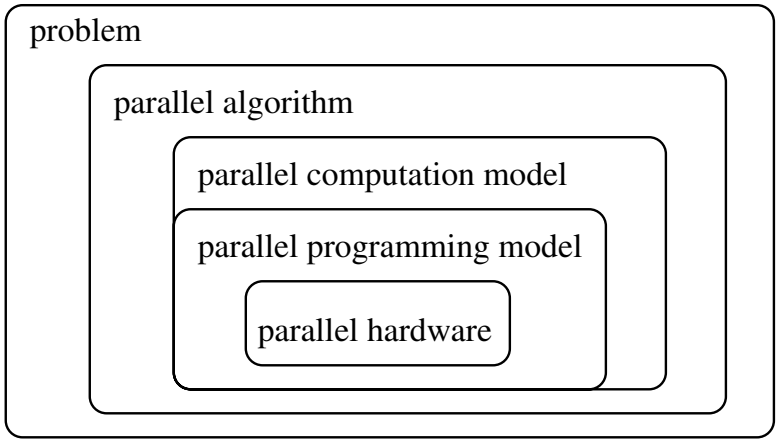

Figure 4: The levels of parallel programming

A parallel *architecture* can be (very roughly) characterized as a combination of features from each of the following three hardware categories:

$$\left\{\begin{array}{c}\text{synchronous}\\ \text{asynchronous}\end{array}\right\}\left\{\begin{array}{c}\text{shared memory}\\ \text{message passing}\end{array}\right\}\left\{\begin{array}{c}\text{MIMD}\\ \text{SIMD}\end{array}\right\}$$

where, in each category, the upper feature is widely regarded as stronger than the lower feature. Not all of these combinations make sense, such as e.g. asynchronous SIMD. The SB-PRAM offers the top combination of the stronger hardware features.

Usually, (parallel) programming languages do not offer direct access to the hardware but establish, as an abstract view of the hardware, a *programming model* which can more or less efficiently be emulated by the hardware. E.g., physical wires may be abstracted by logical channels, as in Occam [37] and CSP [38], or a virtual shared memory may be simulated on top of a distributed memory message passing system.

The goal of the programming language Fork95 is to make the benefits of each category of hardware features available as a programming model. Thus the other common parallel programming models can easily be emulated in Fork95. E.g., asynchronous processes with monitors can be implemented using the multiprefix operators in a straightforward way, as we exemplified in subsection 3.5, or synchronous operations on processor arrays may be implemented using dataparallel loops, as we mentioned in subsection 3.3.

Even more abstract are parallel *computational models*, sometimes also referred to as *skeletons* in the literature (e.g., [12]). They allow for a high–level organization of parallel tasks. Here we consider three general and widely used computational models: farming, parallel divide–and–conquer, and pipelining.

5.1 Farming

Farming means that several slave threads are spawned and work independently on their local tasks.

If these do not need to communicate nor synchronize with each other, farming can be achieved in asynchronous mode with no additional overhead. This is, clearly, also possible in synchronous mode, at the expense of subgroup creation at each private conditional, incurring superfluous overhead.

Nevertheless, in the case of data dependencies that have to be preserved, synchronous mode may be even more suitable than asynchronous mode. For instance, a 1D Jacobi–like relaxation step may be programmed in synchronous mode just as follows:

```
sync void relax( sh float a[] )
{
  pr int k = $+1;
  a[k] = 0.25*(a[k-1]+2*a[k]+a[k+1]);  /*compute weighted average */
}                                      /*over neighboured elements*/
```

In asynchronous mode, though, one would need a temporary array `temp[]` and a `barrier` statement to preserve the inter–task data dependences:

```
async void a_relax( float a[], float temp[] )
{
  pr int k = $+1;
  temp[k] = 0.25*(a[k-1]+2*a[k]+a[k+1]);
  barrier;
  a[k] = temp[k];
}
```

In asynchronous mode, the `mpadd()` operator is very useful to program self–scheduling parallel loops. Self–scheduling (see e.g. [39] for a survey) dynamically assigns loop iterations to processors if the iterations are independent but take different execution times to complete, which are typically unknown at compile time. A shared loop counter

```
sh int loopcounter = 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 fetch its next iteration:

```
pr int i;
for ( i=mpadd(&loopcounter,1); i<N; i=mpadd(&loopcounter,1) )
    iteration ( i );
```

On a hardware platform with built–in `mpadd` instruction such as the SB-PRAM, parallel accesses to the `loopcounter` are not sequentialized, and the additional overhead is marginal.

Furthermore, when partial results computed by farming are to be composed in parallel into a shared array data structure, the `mpadd()` operator may also be of great help, as we have shown in [40] for an implementation of a solver for a system of linear inequalities in Fork95.

5.2 Parallel Divide–and–Conquer

Parallel divide–and–conquer means that the problem and the processor set working on it is recursively subdivided 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.

Parallel divide–and–conquer is a natural feature of the synchronous mode of Fork95. A generic parallel divide–and–conquer algorithm `DC` may look as follows:

```
sync void DC ( sh int n, ... )
{
 sh int d;
```

```
 if (trivial(n))
    conquer( n, ... );
 else {
    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) where each one is responsible for the parallel and synchronous solution of one of the subproblems. After their solution, the leaf groups are removed again, and all processors of the original group join together to synchronously combine the partial results. Section 4 has shown that the compile-time overhead to manage this type of programs is quite low.

Subdivision into *two* subgroups can also be achieved using the `if` statement with a private condition, as exemplified in the FFT example (see Appendix A.2). More example programs for parallel DC are contained in the `examples` directory of the Fork95 distribution, e.g. a parallel implementation of Strassen's recursive algorithm for matrix multiplication [41] which uses a `fork` subdividing into seven subgroups (see also [42]).

5.3 Pipelining

For *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 its source nodes 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 and leveled). The time to execute one step of the pipeline is determined by the maximum execution time of a stage.

Pipelining through an arbitrary graph can be implemented in a rather straight forward manner. The graph may be defined as a shared data structure the data for every node of the graph through which the data are piped are grouped in a structure `Node`.

```
struct Node {
    Data *data;    /* buffer to pass intermediate values to next stage */
    int *pre;      /* array of indices of predecessor nodes in graph[] */
    int stage;     /* depth in pipeline */
}
```

This structure contains a pointer to some local data, a vector of references to 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`:

```
sh struct Node graph[n];  /* Pipeline graph consisting of n nodes */
```

The graph nodes are allocated, and the fields `data` and `pre` are initialized by some routine

```
sync void init_graph();   /* initializes nodes */
```

The proper work to be done by a processor at each stage of the pipeline is given by a routine

```
sync void work();  /* specifies work to be done */
```

For simplicity, let us first assume that the n node pipeline is executed by exactly n processors. We assign to each processor j one node `graph[`j`]` of the pipeline graph. To execute the pipeline with `n` processors, a processor starts to work on its node as soon as the first wave of computation, implemented by a time step counter `t`, reaches its stage in the pipeline graph:

```
sh int t;
init_graph();
for(t = 0; t < end; t++)
   if (t >= graph[$].stage)
      work();
```

Besides the data structure `Data`, the programmer essentially must provide the two functions `init_graph()` and `work()`.

`init_graph()`: 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:

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

All `stage` entries are initialized with -1. The call to `non_neg()` tests whether all predecessor nodes of `graph[`j`]` have been assigned a non–negative `stage` entry. The stage of a node is determined as the number t of the first iteration where all its predecessors already obtained values ≥ 0 while its current stage entry still equals -1.

`work()`: specifies the operation to by executed by processor j at node `graph[`j`]`. Input data should be read from the `data` entries of the nodes `graph[`i`]` which are predecessors of `graph[`j`]`.

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.

It may happen, though, that we would like to dedicate more than one processor to each node. To handle this case we modify our generic algorithm as follows:

```
sh int t;
init_graph();
for (t = 0; t < end; t++)
   fork (n; @=select(t); $=rename() )   /*create n subgroups*/
      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 index of this node can be accessed through the group ID `@`. 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.

We observe that pipelining and systolic computations can be easily and flexibly implemented in Fork95 using the available language features. An additional construct for pipelining is not required.

6 Conclusion

We have presented the Fork95 language and a prototype compiler for the SB-PRAM machine. We have further shown that Fork95 allows to express important parallel computation models that occur in parallel algorithms for real problems. Fork95 enables easy integration of existing (sequential) C codes; for instance, a register allocator consisting of 1000 lines of C code has been successfully ported to Fork95 within a few hours.

Fork95 can be efficiently implemented on the SB-PRAM machine. The compiler and required tools are available.

Fork95 offers a broad basis for experiments on parallelism. We have used Fork95 successfully as a first parallel programming language in a parallel programming course at the University of Trier. A library of frequently used basic PRAM algorithms like parallel prefix computations, searching, merging and sorting, is currently being implemented in Fork95 [31].

Future research may address the implementation of Fork95 (or a closely related modification) on further hardware platforms. The language itself may gain much attractivity by integrating object–oriented features, e.g. by extending the base language from C to C++.

Acknowledgement

The authors thank Jesper L. Träff for his helpful comments on an earlier version of this paper.

References

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

[2] Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. Realization of PRAMs: Processor Design. In *Proc. WDAG94, 8th Int. Workshop on Distributed Algorithms, Springer Lecture Notes in Computer Science vol. 857*, pages 17–27, 1994.

[3] 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.

[4] Ralph Butler and Ewing L. Lusk. User's Guide to the P4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.

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

[6] Michael Philippsen and Markus U. Mock. Data and Process Alignment in Modula-2*. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 177–191. Wiesbaden: Vieweg, 1994.

[7] K.-C Li and H. Schwetman. Vector C: A Vector Processing Language. *Journal of Parallel and Distributed Computing*, 2:132–169, 1985.

[8] J. Rose and G. Steele. C*: an Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Inc., Cambridge, MA, 1987.

[9] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming in MIMD Computers.* MIT Press, 1991.

[10] Judith Schlesinger and Maya Gokhale. DBC Reference Manual. Technical Report TR-92-068, Supercomputing Research Center, 1992.

[11] 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.

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

[13] 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.

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

[15] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. Part II: Binary Tree and FFT Algorithms. *Journal of Parallel and Distributed Computing*, 16:233–249, 1992.

[16] Y. Ben-Asher, D.G. Feitelson, and L. Rudolph. ParC — An Extension of C for Shared Memory Parallel Processing. *Software – Practice and Experience*, 26(5):581–612, May 1996.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multi-threaded run-time system. In *Proc. 5th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 207–216, 1995.

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

[19] C. León, F. Sande, C. Rodríguez, and F. García. A PRAM Oriented Language. In *EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing*, pages 182–191. Los Alamitos: IEEE Computer Society Press, Jan. 1995.

[20] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, Dec. 1993.

[21] Jochen Röhrig. Implementierung der P4-Laufzeitbibliothek auf der SB-PRAM. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1996.

[22] T. Grün, T. Rauber, and J. Röhrig. The Programming Environment of the SB-PRAM. In *Proc. ISMM'95*, 1995. `http: //www-wjp. cs.uni-sb.de /SBPRAM/`.

[23] Arno Formella, Jörg Keller, and Thomas Walle. HPP: A High–Performance PRAM. In *Proc. 2nd Int. Euro-Par Conference.* Springer LNCS, 1996.

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

[25] Christoph W. Keßler and Helmut Seidl. Language Support for Synchronous Parallel Critical Sections. In *Proc. APDC'97 Int. Conf. on Advances in Parallel and Distributed Computing, Shanghai, China.* Los Alamitos: IEEE Computer Society Press, March 1997.

[26] 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):164–189, April 1983. (see also: Ultracomputer Note No. 16, Dec. 1980, New York University).

[27] Henri Bal. *Programming Distributed Systems.* Prentice Hall, 1990.

[28] Christoph W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität des Saarlandes, Saarbrücken (Germany), 1994.

[29] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, Aug. 1973.

[30] Guy E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, Nov. 1989.

[31] C.W. Keßler and J.L. Träff. A Library of Basic PRAM Algorithms and its Implementation in FORK. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 193–195. New York: ACM Press, June 24–26 1996.

[32] Karin Käppner. Analysen zur Übersetzung von FORK, Teil 1. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1992.

[33] Markus Welter. Analysen zur Übersetzung von FORK, Teil 2. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1992.

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

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

[36] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin Cummings Publishing Company, 1995.

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

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

[39] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[40] C.W. Keßler. Parallel Fourier–Motzkin Elimination. In *Proc. 2nd Int. Euro-Par Conference*, pages 66–71. Springer LNCS 1124, Aug. 27–29 1996.

[41] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[42] Christoph W. Keßler and Helmut Seidl. Fork95 Language and Compiler for the SB-PRAM. In E.L. Zapata, editor, *Proc. 5th Workshop on Compilers for Parallel Computers*, pages 409–421. Dept. of Computer Architecture, University of Malaga, Spain. Report No. UMA-DAC-95/09, June 28–30 1995. `http://www.informatik.uni-trier.de/ ~kessler/fork95.html`.

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 they are not, this can be provided in $O(1)$ time by a `mpadd` operation). Run time results for the SB-PRAM are given in Table iii.

```
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;    /*temporary accumulator cell*/
  pr int i;

  /*step over n/p slices of array:*/
  for (i=$; i<n; i+=p)
     out[i] = mpadd( &sum, in[i] );
}
```

# processors	cc, $n = 10000$	cc, $n = 100000$
2	430906	4300906
4	215906	2150906
8	108406	1075906
16	54656	538406
32	27822	269656
64	14406	135322
128	7698	68156
256	4344	34530
512	2624	17760
1024	1764	9332
2048	1334	5118
4096	1162	3054

Table iii: Run time results (in SB-PRAM clock cycles) for `parprefix.c`

A.2 Divide–and–conquer using private `if`: Recursive FFT

The following recursive core routine of a simple parallel implementation of the Fast Fourier Transform requires as parameters the complex operand array, its size, and an array containing all powers 1, w, w^2,...,w^{n-1} of a complex nth root of unity, w. We assume `n` is a power of 2. The operand array `a` is not overwritten. The routine returns the Discrete Fourier Transform of `a`. Run time results for the SB-PRAM are shown in Table iv.

```
sync cplx *fft( sh cplx *a, sh int n, sh cplx *w )
{
 sh cplx *ft;                              /*result array*/
 sh cplx *even, *odd, *fteven, *ftodd;     /*temporary pointers*/
 sh int p = 0;
 sh int ndiv2;
 pr int i;

 $ = mpadd( &p, 1 );  /* ensure consecutive numbering of procs */

 if (n==1)  {
    farm if ($==0) {
       ft = shmalloc(1);
       ft[0] = cnum( a[0]->re, a[0]->im );
    }
    return ft;
 }
 if (p==1)  return seq_fft( a, n, w );

 /* general case: compute temporary arrays even[], odd[]    *
  *               and in parallel call fft for each of them */
 ft = (cplx *) shmalloc( n );  /* allocate result array */
 ndiv2 = n>>1;
 even = (cplx *) shalloc( ndiv2 );
 odd  = (cplx *) shalloc( ndiv2 );
 for( i=$; i<ndiv2; i+=p ) {     /*dataparallel loop*/
   even[i] = a[2*i];             /* copy pointer to same number */
   odd[i]  = a[2*i+1];
 }
```

processors	SB-PRAM clock cycles
1	16877614
4	4203828
16	1050008
64	265852
256	71520
1024	23996

Table iv: Run time results for the FFT implementation, measured for a complex array of 4096 elements.

```
 if ($<p/2)  /*split current group into two equally sized subgroups:*/
      fteven = fft( even, ndiv2, w );
 else                                  ftodd = fft( odd, ndiv2, w );

 for( i=$; i<ndiv2; i+=p ) {     /*dataparallel loop*/
   pr cplx t = cmul( w[i], ftodd[i] );
   ft[i]        = cadd( fteven[i], t );
   ft[i+ndiv2]  = csub( fteven[i], t );
   freecplx( t );
 }
 shfreecplxarray( fteven, ndiv2 ); shfreecplxarray( ftodd, ndiv2 );
 shallfree();  /* free even, odd */
 return ft;
}
```