Fork95 Language and Compiler for the SB-PRAM

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

Fork95 is a redesign of the PRAM language FORK. Fork95 is based on ANSI C and offers 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 also comfortable facilities for locally asynchronous computation where desired by the programmer. Altogether, Fork95 offers full expressibility for the implementation of practically relevant parallel algorithms.

In this paper we give a short overview on the Fork95 language and present a one–pass compiler, `fcc`, for Fork95.

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 `PRAM`s (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 [AKP90, KPS94]. 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 amount of algorithms has been invented for this type of architecture in the last two decades.

Surprisingly enough, not much attempts have been made to develop languages which allow both to conveniently express algorithms and generate 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 [LT93], 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 [BL92], [BL94]. 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 `Vector C` [LS85] and `C*` [RS87] or its relatives `Dataparallel C` [HQ91] and `DBC` [SG92].

The limitations of these languages, however, are obvious. There is just one global name space. Other programming paradigms like a parallel recursive divide–and–conquer style as suggested in [BDH+91], [Col89], [dlTK92], [HR92a], [HR92b] are not supported.

The only attempt we are aware of which allows both parallely recursive and synchronous programming are the imperative parallel languages `FORK` [HSS92] and `ll` [LSRG95]. 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 [HSS92] 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. 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 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.

Fork95 offers full expressibility for the implementation of practically relevant parallel algorithms because it supports all known parallel programming paradigms used for the parallel solution of real–world problems.

In this paper, we will focus on the current implementation of the Fork95 compiler, `fcc`, for the SB–PRAM.

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. The appendix lists some example programs.

2 The SB-PRAM from the programmer's view

The SB-PRAM [AKP90] 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 floatingpoint 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 [KPS94] 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 logical `or` which also execute within two CPU cycles.

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.

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.

Since the SB-PRAM hardware is not yet available (a 128-PE-prototype is to be expected for summer 1995, the full extension for 1996), we use a simulator that allows to measure exact program execution times.

We would like to emphasize that the SB-PRAM is in deed 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 a redesign of the PRAM language FORK [HSS92]. Fork95 is based on ANSI C [ANS90]. 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.

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

3.1 Starting Processors in Parallel

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, there remains only one processor active which starts execution of the program by calling function `main()`.

The statement `start(`e`)` whose shared expression e evaluates to some integer value k, starts k processors simultaneously, with unique (absolute) processor IDs called `__PROC_NR__` numbered successively from 0 to $k-1$. If the expression e is omitted, then all available processors executing this program are started.

3.2 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. 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.

Additionally, 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 *relative* processor ID (relative to `$`) 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 at least one 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.

3.3 The Group Concept

At each point of program execution, Fork95 maintains the invariant that all processors belonging to the same 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. This rule can be relaxed in two ways: by splitting a group into subgroups and maintaining the invariant only within each of the subgroups, or by explicitly entering the asynchronous mode via a `farm` construct

```
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.

Functions are classified to be either synchronous (`sync`) or asynchronous (`async`). Within a `farm` and within an `async` function, only `async` functions can be called. Calling an `async` function from a synchronous context (i.e., the call being located in a `sync` function and not within a `farm` body) results in an implicit entering of the asynchronous mode; the programmer receives a warning. Using

[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.

farm within a farm body or within an async function is superfluous and may even introduce a deadlock (a warning is emitted).

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 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 temporally 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 to compute 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 [Keß94].

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()` 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.

3.7 Caveats in Fork95 Programming

3.7.1 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 context.

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 be unknown 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 context, there is no danger of deadlock.

3.7.2 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.

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 Fast Start of Processors

Each processor has got an inactivity bit (*shadow bit*) which, if set, cancels all global store and multiprefix operations of that processor (push operations are, though, admitted). Also the I/O routines test the shadow bit to mask their activity. Thus, starting and stopping an arbitrary number of available processors is done on-the-fly within a few CPU cycles by only adjusting their shadow bits.

4.2 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 1) 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 `$`.

```
.globl forklib_sync
forklib_sync: /*no parameter; uses r31, r30*/
bmc   0               /*force next modulo 0 */
getlo -1,r30          /*load constant -1   0*/
mpadd gps,1,r30       /*decr. sync cell    1*/
nop                   /*delay-slot mpadd   0*/
nop                   /*modulo ++          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: bypass1*/
nop                   /*delay early wave   0*/
nop                   /*delay early wave   1*/
FORKLIB_SYNCHRON:
return                /*sync: finished     0*/
nop                   /*delay-slot return  1*/
```

Figure 1: The synchronization routine in `fcc`, as proposed by Jörg Keller (Saarbrücken University). The modulo flag is a globally visible bit in the status register which is inverted after each machine cycle; thus it can be used as a semaphore to separate reading and writing accesses to the same memory location. The `bmc` instruction causes the processors to enter the SYNCLOOP loop only if the modulo flag is 0. Because the SYNCLOOP loop has length 4, all processors that are inside 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.

Private loops only build a shared group frame for the group of iterating processors. A private group frame is not necessary, as there is usually no need to change the values for `@` and `$`.

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.

4.3 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`.

4.4 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 `elselabel`;

(5) set shared stack pointer and shared heap pointer to the limits of B_0

Table 1: 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 context, 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.

construct:	overhead in `SB-PRAM` clock cycles:
synchronize:	$t_{sync} = 10$
startup:	$150 + 4 \times \|\textit{private .data section}\|$
`start`:	50
loop:	$10 + 5 \times \#\textit{iterations} + t_{sync}$
`if`:	$32 + t_{sync}$
`fork`:	$44 + t_{division} + t_{sync}$
`farm`:	t_{sync}
call, synchr.:	$41 + \#(\textit{used regs}) + \#(\textit{private args})$ $+2 \times \#(\textit{shared args}) + t_{sync}$
call, asynchr.:	$10 + \#(\textit{used regs}) + \#(\textit{private args})$
`malloc`/`shalloc`:	4
division:	$t_{division} = 12 \dots 300$ (data dep.)

(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 `finishedlabel`
(10) `elselabel:` 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) `finishedlabel:` remove shared and private group frame, restore shared stack pointer, heap pointer, and the group pointers; call the synchronization routine (Figure 1)

Important optimizations (as [Käp92, Wel92] 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`s.

Special care has to be taken for `break` or `continue` statements occurring in `statement1` or `statement2`; in these cases, the private group frame has to be removed, the synchronization cell of the parent group has to be decremented, and the pointers have to be restored as in the last item above, before jumping out of the branch.

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 [FH91a, FH91b, FH95].

Table 1 shows the overheads introduced by the different constructs of the language.

The compiler generates assembler code which is processed by the SB-PRAM–assembler `prass` 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.

4.6 Limitations of the Compiler

Conditional expressions $e?l : r$ in synchronous mode do — unlike the private `if` statement — not build group frames. As a private condition e may introduce a deadlock, a warning is emitted in this case. This is due to the strange construction of expression DAGs in the `lcc`. Nevertheless, this does not restrict the programmer — he can use the `if` statement instead.

The same holds for `switch` statements in synchronous mode: Currently, no group frames are 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 sensible for a one–pass–compiler like `fcc`.

The C library is not yet fully implemented. We have provided some important routines e.g. for screen output, string manipulation, and mathematical functions. Extending the functionality of asynchronous context programming, we are also working on a set of primitives to handle self–balancing parallel loops and parallel queues [Röh96].

5 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

[AKP90] 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, 1990.

[ANS90] ANSI American National Standard Institute, Inc., New York. American National Standards for Information Systems, Programming Language C. ANSI X3.159–1989, 1990.

[BDH+91] 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.

[BL92] 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.

[BL94] 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.

[Col89] M.I. Cole. *Algorithmic Sceletons: Structured Management of Parallel Computation.* Pitman and MIT Press, 1989.

[dlTK92] P. de la Torre and C.P. Kruskal. Towards a Single Model of Efficient Computation in Real Parallel Machines. *Future Generation Computer Systems*, 8:395–408, 1992.

[FH91a] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software—Practice & Experience*, 21(9):963–988, Sept. 1991.

[FH91b] C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct. 1991.

[FH95] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin Cummings Publishing Company, 1995.

[HQ91] P.J. Hatcher and M.J. Quinn. *Dataparallel Programming on MIMD Computers.* MIT-Press, 1991.

[HR92a] 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.

[HR92b] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation. Part II: Binary Tree and FFT Algorithms. *Journal of Parallel and Distributed Programming*, 16:233–249, 1992.

[HSS92] T. Hagerup, A. Schmitt, and H. Seidl. FORK: A High–Level Language for PRAMs. *Future Generation Computer Systems*, 8:379–393, 1992.

[Käp92] Karin Käppner. Analysen zur übersetzung von FORK, Teil 1. Master thesis, Universität Saarbrücken, 1992.

[Keß94] Christoph W. Keßler. *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, 1994.

[KPS94] 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.

[LS85] K.-C Li and H. Schwetman. Vector C: A Vector Processing Language. *Journal of Parallel and Distributed Computing*, 2:132–169, 1985.

[LSRG95] 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.

[LT93] Oak Ridge National Laboratory and University of Tennessee. Parallel Virtual Machine Reference Manual, Version 3.2. Technical report, August 1993.

[Röh96] Jochen Röhrig. title to be announced (in german language). Master thesis, Universität Saarbrücken, to appear 1996.

[RS87] J. Rose and G. Steele. C*: An Extended C Language for Data Parallel Programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[SG92] Judith Schlesinger and Maya Gokhale. DBC Reference Manual. Technical Report TR-92-068, Supercomputing Research Center, 1992.

[Str69] V. Strassen. *Numerische Mathematik*, volume 13. 1969.

[Wel92] Markus Welter. Analysen zur übersetzung von FORK, Teil 2. Master thesis, Universität Saarbrücken, 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 they are 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;
         /*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] );
}
```

Run time results (in SB-PRAM clock cycles) for the parallel prefix example:

# 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

A.2 Example: Divide–and–Conquer

As illustration for the application of the `fork` statement, we have implemented Strassen's recursive matrix multiplication algorithm [Str69], where at the main divide–and–conquer phase we solve the 7 subproblems in parallel by subdividing the current processor group into 7 subgroups.

```
sh int *A, *B, *C;
sh int N = 16;      /* matrix extent, must be a power of 2 */

main() {
 pr int i, j, p;
 start (343) {  /* 7^3 */
   A = (int *) shalloc(N*N);
   B = (int *) shalloc(N*N);
   C = (int *) shalloc(N*N);
   p = groupsize();
   farm init_matrices();
   strassen_mult( A, N, B, N, C, N, N );
   farm output_array( C, N );           /*print the resulting array*/
 }
}

sync void add(
   sh int *a, sh int sa,      /* matrix and its allocated extent */
   sh int *b, sh int sb,
   sh int *c, sh int sc,
   sh int n )                 /* problem size */
{ /* n x n - add two arrays a, b: */
  pr int i, j;
  sh int p = groupsize();
  farm
   for(i=$; i<n; i+=p)    /* parallel loop */
     for (j=0; j<n; j++) {
        c[i*sc + j] = a[i*sa + j] + b[i*sb + j];
     }
}

sync void inv( sh int *a, sh int sa, sh int *c, sh int sc, sh int n )
{
  pr int i, j;
  sh int p = groupsize();
  farm
   for(i=$; i<n; i+=p)    /* parallel loop */
     for (j=0; j<n; j++)
        c[i*sc+j] = - a[i*sa+j];
}

async void mult_directly( int *a, int sa, int *b, int sb, int *c, int sc, int n )
{
  pr int i, j, k;           /*sequential matrix multiplication*/
  pr int val;

  for(i=0; i<n; i++)
     for (j=0; j<n; j++) {
        val = 0;
        for (k=0; k<n; k++)
          val += a[i + k*n] * b[k + j*n];
        c[i*n + j] = val;
     }
}

/* comp1(), ..., comp7()  install Strassen's set of formulae: */

sync void comp1( sh int *a11, sh int *a22, sh int sa,
                 sh int *b11, sh int *b22, sh int sb,
                 sh int *q1, sh int ndiv2 )
{
  sh int *t1 = (int *) shalloc(ndiv2 * ndiv2);
  sh int *t2 = (int *) shalloc(ndiv2 * ndiv2);
  add( a11, sa, a22, sa, t1, ndiv2, ndiv2 );
  add( b11, sb, b22, sb, t2, ndiv2, ndiv2 );
  strassen_mult( t1, ndiv2, t2, ndiv2, q1, ndiv2, ndiv2 );
}

[ ... ]    /* comp2(), ..., comp7() work in the same way */
```

```
sync void strassen_mult(
  sh int *a,      /* operand array, allocated size sa x sa, extent n x n */
  sh int sa,
  sh int *b,      /* operand array, length n x n */
  sh int sb,
  sh int *c,      /* result array, length n x n */
  sh int sc,
  sh int n )      /* problem size */
{
  sh int ndiv2 = n>>1;
  sh int *a11, *a12, *a21, *a22, *b11, *b12, *b21, *b22, *c11, *c12, *c21, *c22;
  sh int *q1, *q2, *q3, *q4, *q5, *q6, *q7;
  sh int *t11, *t12, *t21, *t22;
  sh int p = groupsize();

  farm if (p == 1) {    /* no more processors available: */
          mult_directly( a, sa, b, sb, c, sc, n );
          return;
  }

  if (n==1) {   /* the trivial case */
     *c = *a * *b;
     farm prI( *c, 0 );
     return;
  }

  a11 = a;               a12 = a + ndiv2;
  a21 = a + sa*ndiv2;    a22 = a + sa*ndiv2 + ndiv2;
  b11 = b;               b12 = b + ndiv2;
  b21 = b + sb*ndiv2;    b22 = b + sb*ndiv2 + ndiv2;
  c11 = c;               c12 = c + ndiv2;
  c21 = c + sc*ndiv2;    c22 = c + sc*ndiv2 + ndiv2;

  q1 = (int *) shalloc(ndiv2 * ndiv2);
  [ ... same for q2, ..., q7 ... ]

  /* explicitly open 7 subgroups: */

  fork ( 7; @ = $%7; $=$/7 ) {     /* the @-th group computes q{@+1}: */
     switch (@) {          /* no additional group frame construction */
        case 0: comp1( a11, a22, sa, b11, b22, sb, q1, ndiv2 ); break;
        case 1: comp2( a21, a22, sa, b11, sb, q2, ndiv2 ); break;
        case 2: comp3( a11, sa, b12, b22, sb, q3, ndiv2 ); break;
        case 3: comp4( a22, sa, b11, b21, sb, q4, ndiv2 ); break;
        case 4: comp5( a21, a22, sa, b11, sb, q5, ndiv2 ); break;
        case 5: comp6( a11, a21, sa, b11, b12, sb, q6, ndiv2 ); break;
        case 6: comp7( a12, a22, sa, b21, b22, sb, q7, ndiv2 ); break;
     }
  } /* end of fork */

  /* explicitly reopen 2 subgroups: */

  fork( 2; @ = $%2; $ = $/2 )
      if (@==0) {    /* shared if condition */
                /* the first subgroup computes c11 and c21 */
                t11 = (int *) shalloc(ndiv2 * ndiv2);
                t12 = (int *) shalloc(ndiv2 * ndiv2);
                inv( q5, ndiv2, t11, ndiv2, ndiv2 );
                add( t11, ndiv2, q7, ndiv2, t12, ndiv2, ndiv2 );
                add( t12, ndiv2, q1, ndiv2, t11, ndiv2, ndiv2 );
                add( t11, ndiv2, q4, ndiv2, c11, sc, ndiv2 );
                add( q2, ndiv2, q4, ndiv2, c21, sc, ndiv2 );
      }
      else {    /* the second subgroup computes c12 and c22 */
                [... similar as for c11 and c21 ... ]
      }
}
```