Making FORK Practical

Christoph Keßler, Helmut Seidl

SFB 124–C1
Report 1/95

December 20, 1994

Christoph Keßler
Fachbereich Informatik
Universität des Saarlandes
Postfach 151150
D–66041 Saarbrücken
Germany
email: `kessler@cs.uni-sb.de`

Helmut Seidl
FB IV – Informatik
Universität Trier

D–54286 Trier
Germany
`seidl@ti.uni-trier.de`

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). A realization of such a machine in hardware, the `SB-PRAM`, is undertaken by a project of W.J. Paul at Saarbrücken [AKP91]. The shared memory with random access allows for a fast and easy exchange of data between the processors, while the common clock guarantees deterministic program execution. Accordingly, a huge amount of algorithms has been invented for this type of architecture.

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$^+$], [Col89], [dlTK92], [HR92a], [HR92b] are not supported.

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

The design of `FORK` , though, was a rather theoretical one. By choosing a syntax for its language constructs which is similar but different to other imperative languages the designers wanted to avoid the confusion caused by identically looking expressions having strongly deviating semantics. At the same time, however, they prohibited the reuse of existing program libraries. Also, in order to allow for a rigorous definition of the semantics [Sch91], [Sch92], [RS92], [Sei93] of the parallel constructs **fork** and **start** they insisted on a rigidly modular program organization (no jumps) and excluded any other data structure besides arrays and records. Being overly restrictive in this respect, they on the other hand allowed nested starting of processors; also, the number of started processors could exceed the number of existing ones. The price to be paid was an overly complicated runtime system.

The current redesign tries to eliminate these deficiencies. Therefore, the new design restricts the applicability of the *start*–construct and the amount of synchronous program execution guaranteed by the language. At least in our opinion, the (small) loss in expressiveness is more than overly compensated by the gains in efficiency through a tremendous reduction of runtime overhead. On the other hand, in order to provide a full–fledged language for real use, all the language features have been added which are well–known from sequential programming. Thus, the new `FORK` dialect 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. This has also impacts on the syntax (and semantics) of our basic parallel constructs **fork** and **start**.

The rest of the paper is organized as follows. Section 2 explains the basic concepts and mechanisms of `FORK95` to control parallel execution. Section 3 explains the new features, namely the heap management, together with the semantics of pointers in a parallel environment; it explains the new concepts **async** and **sync** which allow to use conventional `C` routines for local computations on every processor; finally, the semantics of jumps and its limitations are discussed. Section 4 describes the compiler and the runtime environment; finally, Section 5 concludes.

2 Basic Features

Similar to the former design, there are two kinds of storage, namely *private* (identified by the key word **pr**) and *shared* (identified by **sh**). The default is private. A list of definitions of shared variables may, e.g., look like:

sh int `i`, `a`[20], ∗`p`

Then space for variables `i`, `a`[0]...`a`[19] and `p` is allocated in the shared memory.

Note that the last definition does not define `p` as a pointer to a shared object of type **int** but as a *shared object* of type pointer–to–**int**.

2.1 The *start*–Statement

`FORK95`–Programs are executed by *threads* which we prefer to call *processors* since they are to be mapped statically to the physical processors of the target architecture. Processors possess private data which cannot (directly) be accessed by other processors. To allow for references of each other, every processor possesses one distinct private variable \$, its *processor number*. [1]

Initially, there is just one processor, the *initial processor*, equipped with processor number 0. In order to activate processors from a larger range one may use the *start*–construct. Executing

start (e) `<stat>`

processor 0 first evaluates the expression e (necessarily of type **int**) to some value v; then v processors are activated with processor numbers from the range $[0..v-1]$. These processors synchronously execute statement `<stat>`. When they have finished, they are deactivated again, and only processor 0 proceeds. If the value v exceeds the number of available processors, a runtime error occurs. In this respect, we differ from the original definition of `FORK`. Also in contrast to the original language, nested occurrences of *start*–statements are forbidden. Furthermore, the parameter for **start** is made optional. Executing

start `<stat>`

all processors available by the hardware are started.

In order to exploit the synchronism provided by the hardware, the started processors are meant to operate in lockstep mode.

2.2 The *fork*–Statement

Processors are organized in *processes* or *groups*. A group of processors corresponds to a designated task on the solution of which the members of this group work jointly and synchronously. Groups may again be subdivided into subgroups thus corresponding to a possibly hierarchical subdivision of an initial task. It follows that at some point of program execution, all presently existing groups are organized in a group hierarchy. Only the leaf groups of the group hierarchy are active.

Subgroups of a group can be distinguished by their *group number*. The group number of the leaf group a processor p is member of can be accessed by p through the shared variable @.

[1] In `FORK`, the processor number was denoted by #. In `C`, however, this symbol marks preprocessor directives.

Only processors of leaf groups are guaranteed to run *synchronously* meaning that they are always at the same program point and execute the next computation step simultaneously in parallel. [2]

Every group may have their own shared data which only can be accessed by its members. If a leaf group G executes a definition of a *private* variable `x` then every processor p of G creates a distinct instance of `x` which is only accessible by p. If in contrast, G executes a definition of a *shared* variable `x` then just one instance of this variable is created which can be accessed only by processors of G.

Initially, the group hierarchy H consists of one group with $@ = 0$ consisting of all processors which have been started. All processors run synchronously, i.e., execute the same program in lockstep mode.

To create new subgroups we can use the *fork*–statement. Its syntax is:

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

where e_i are expressions of type **int**, and e_1 does not depend on private data.

Assume leaf group G executes this *fork*–statement. Then first every processor p of G computes values $v_{p,1}, v_{p,2}$, and $v_{p,3}$ of the expressions e_1, e_2, and e_3 respectively. Since e_1 does not depend on private data, all values $v_{p,1}$ are equal to the same value v_1. This value determines the range $[0..v_1 - 1]$ of the set of group numbers of the new subgroups of G which are to be created. The value $v_{p,2}$ returns the group number of the leaf group that processor p is going to enter, whereas the value $v_{p,3}$ gives the new processor number of p within its new leaf group. Having thus updated the current values of \$ and @, the processors start to execute statement `<stat>`. [3] Having finished the execution of `<stat>`, the leaf groups are removed from the group hierarchy, and all processors continue execution within group G. We do no longer guarantee that all processors within all the newly created leaf groups together run synchronously. Therefore, since G must be guaranteed to execute synchronously, an explicit synchronization is necessary at the end of `<stat>`.

There is the possibility also to make the parameters of **fork** optional. In case of a statement

fork `<stat>`

single–processor groups are created with $@ = \$$ and $\$ = 0$.

[2] Note that we use a weaker regime of of synchronous program execution here than in the original **FORK**–language where also larger hierarchies could run synchronously.

[3] We do not assume that *every* new group is chosen by some processor. Hence, some of the new leaf groups may be empty. In fact, the algorithm in [BDH$^+$] relies on that possibility! An empty group, however, immediately has finished its work.

2.3 Conditionals

A special subtlety occurs with the treatment of conditional branching points. Consider a leaf group G executing

`<stat>` $\equiv$ **if** (e) s_1 **else** s_0

If e only depends on shared data, all the processors compute identical values for e and therefore select the same branch. The situation, however, is totally different as soon as e depends on private data as well. Then some of the processors of G may evaluate e to 0 (i.e., "false"), others to something $\neq 0$ (i.e., "true"). This would imply that the processors of G may reach different program points at the same time, i.e., become asynchronous. This is avoided by splitting G into two subgroups G_0 and G_1 at runtime where G_0 contains all processors of G evaluating e to 0, and G_1 contains all processors of G evaluating e to some value different from 0. The new subgroups G_i inherit the group number of G. They are synchronous and start to execute s_i. When the execution both of s_0 and s_1 has terminated, the leaf groups G_i are removed, and G synchronously continues program execution.

switch–statements

switch (e) `<stat>`

are treated analogous to **if**'s in that (in case the value of e depends on private data) as many subgroups are created as there are alternatives in the body `<stat>`.

2.4 Write Conflicts

So far, we did not fix what happens if several processors simultaneously access the same instance of some variable `x`. There exist many possibilities to do so (see, e.g., [Sch92] for an exhaustive discussion of the subject). For simplicity, we choose the *arbitrary* convention, i.e., we both allow simultaneous read and simultaneous write accesses to `x`. Let P be the set of processors that simultaneously execute a write operation on `x`. Then the resulting value of `x` can be *any* value a processor from P wanted to write. Hence, there are two possible sources of non–determinism in our language. First, different interleavings of asynchronously executed parts of the program are possible. Secondly, simultaneous write accesses may result in different values. A correct implementation of `FORK` realizes one of the set of possible behaviors. Therefore, meaningful programs should be robust against all these kinds of non–deterministic choices.

3 Advanced Features

3.1 Farming

The implicit formation of subgroups at conditionals and switches depending on private variables is rather expensive. It can be avoided by means of the new construct

farm `<stat>`

Additionally, no synchronous program execution is guaranteed during the execution of `<stat>`. Synchronization of the leaf group is provided, though, at the end of the execution of `<stat>`.

The *farm*–statement is useful for portions of the program where the processors just perform local computations. Omission of implicit subgroup formation implies, however, that allocation of shared variables within `<stat>` must be avoided. Especially, functions that are called must have only private parameters and return a private value (if any).

Type qualifiers **sync** and **async** are introduced to distinguish between ordinary functions and those that can be called within the *farm*–statement. A function declared, e.g., as

sync int fun();

can be called outside *farm*–statements but must not be called inside. Accordingly, a function

async int fun();

may only be used inside a *farm*–statement. Also, asynchronous functions must not contain any of the new constructs of `FORK95` or call functions qualified as synchronous.

Since ordinary `C` programs should be executable within *farm*–statements without any syntactical changes, the general default is **async**.

3.2 Pointers and Heaps

The most important innovation is that the new `FORK` supports pointers. Our key observation about the `SB-PRAM` (but which also can be seen as a hint for an implementation on any other synchronous shared–memory MIMD architecture) is that in fact, no "true" private storage areas exist. This is due to the belief of the hardware designers that a larger amount of easily accessible shared memory is much more flexible to use than a smaller shared memory together with a private memory for every processor which cannot be accessed much faster.

Thus, the *language concept* of private variables can be implemented by areas within the single shared storage *owned* by processors for private use. Hence, we find it legal that a shared pointer variable may point to some private object and vice versa – the question whether or not this is good style, though, is left to the taste of the programmer. In any case, prevention turns out to be much more complicated than admission.

As usual in C, there is an address operator '&' (returning the L–value of an expression) and dereferencing by means of '$*$'. In difference, however, to sequential C, there are two kinds of heaps for dynamically created objects to reside in, namely *private heaps* situated in the processors' owned portions of the shared memory, and *shared heaps*. Consequently, there are also two library routines, namely **malloc** and **shalloc**. Both operate analogously to the C **malloc** – with the only exception that **malloc**(n) executed by processor p, allocates n cells in the private heap of p, whereas **shalloc**(n) executed synchronously by all the processors of some leaf group G, allocates n cells in the portion of shared memory given to G. Both return a pointer to the first allocated memory cell.

There are just two points where special care must be taken. When the leaf groups G_i at group G are removed from the current group hierarchy, then all the G_i's segments of shared memory are united to form the free storage for group G. This design decision has been made for reasons of space economy. However, it implies that all objects allocated in one of the G_i's shared heaps are automatically removed.

The second problem arises from pointers to functions. First note that in FORK95 unlike in the original language, all return values of functions are private. Thus,

sh sync int ($*$f)(**int**)

does *not* denote a pointer to a synchronous function that returns a shared value but a *shared pointer* to a synchronous function.

In case the pointer is shared, all processors within the leaf group in question will execute the same call, and everything is fine. A call to a function through a private pointer, however, can be looked at as a huge switch. Consequently, separate groups must be created for every function of appropriate type in the system. Thus, the implementer either has to detemine those groups which are non–empty or must provide a seperate piece of storage for every function possibly pointed at. Since in general, most of the functions will not be called, the second alternative would cause a tremendous waste of shared memory and program size. The first alternative, on the contrary, is inacceptably time–consuming. This is the reason why, in the present version of FORK95, calls of functions through private pointers are executed asynchronously. Technically, this is achieved by implicitly enclosing them into a *farm*–construct provided they are not already situated within such a statement or a function qualified as **async**. Accordingly, private pointers may only point to functions qualified as **async**.

3.3 Tamed Jumps

In FORK95, jumps occur in five disguises:

1. **goto** l jumps to the statement labeled l;
2. **continue** jumps to the end of the innermost including loop–body;
3. **break** jumps to the end of the innermost including *switch*– resp. loop-statement;
4. **return** jumps to the end of the current function body;
5. **exit** jumps to the end of the program.

Both *start*– and *farm*–statements are, at least in this context, viewed as special loops (namely parallel ones) whereas *fork*–statements are not. The reason for this decision was to extend the analogy between *fork*–statements and *if*–statements where the condition depends on private variables: both constructs create new subgroups and distribute processors. Therefore, in FORK95, both also behave similar w.r.t. **break** and **continue**.

In a parallel setting, jumps may be even more harmful than in a sequential environment.

1. By jumping outside a *start*–statement a processor might escape from being killed at the end of the body.
2. By jumping outside a *fork*–statement or a conditional with private condition a processor may "leave" its present group; accordingly, if it jumps into the body of another **fork**, it may irregularly "enter" another group.
3. It is absolutely unclear what it should mean that a processor crosses the boundaries of a *farm*–statement.

The problem which arises in any of these situations is that the numbers of processors within a group and thus the number of processors involved, e.g., in synchronization operations for this group may change unpredictably.

At the present status of the language definition and its implementation special care is only taken for **continue**, **break** and **return**. In all these cases, the direction of the jump is "from the inside to the outside", namely to the end of some surrounding statement. Especially, the number of surrounding group building constructs (i.e., the length of the path in the group hierarchy from the present leaf group to the group being leaf at the destination of the jump) can be determined at compile-time.

Therefore, a corresponding number of updates can be generated to recompute the number of remaining processors of all groups inbetween.

Note however, that an extra synchronization cell and extra code for synchronization has to be added to the translation of loops which contain **continue** or **break**. In case of **break**s, synchronization has to be inserted immediately after the loop; in case of **continue**, immediately after the body of the loop. Analogously, the execution of functions containing **return**s must be terminated by a synchronization.

4 The Implementation

A first version of a compiler for `FORK95` has been implemented. It is based on the `lcc`, a one–pass `ANSI C`–compiler developed by Chris Fraser and David Hanson at Princeton, NY [FH91a], [FH91b], [FH94]. For synchronization our compiler makes intensive use of the hardware–supplied multiprefix operations of the `SB-PRAM`. These are also available to the user as `FORK95` operators. Since the compiler maps at most one thread to every physical processor, the facilities for rapid context switches are not exploited.

Here is a list of the overheads introduced by the different constructs of the language:

Constructs:	Number of `SB-PRAM`–Cycles:
synchronization:	$t_{sync} = 10$
startup:	$150 + 4 \times \|data\|$
start:	50
loop:	$10 + 5 \times \#iterations + t_{sync}$
if:	$32 + t_{sync}$
fork:	$44 + t_{division} + t_{sync}$
farm:	t_{sync}
call, synchronous:	$41 + \#(used\ regs) + 2 \times \#(shared\ args)$ $+ \#(private\ args) + t_{sync}$
call, asynchronous:	$10 + \#(used\ regs) + \#(private\ args) + t_{sync}$
malloc/shalloc:	4
division:	$t_{division} = 12 \ldots 300$ (data dependent)

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. Synchronization, however, is suppressed whenever the divisor is shared or constant. The cost of calls clearly can be reduced by passing aruments 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.

Since the realization of the `SB-PRAM` is not yet completed, a hardware simulator for the `SB-PRAM` has been implemented [KPS94]. The object code produced by the compiler is in COFF format. It can be executed and debugged on this simulator. The compiler together with the simulator can be obtained via anonymous ftp from `lunix.cs.uni-sb.de` in directory `fork95`.

5 Conclusion

We described how the parallel programming language `FORK` can be modified to become a superset of the sequential language `C`. Depending on the application the programmer has in mind, embedding of `C` functions into `FORK95`–programs can be done in three ways. The programmer may include `C` routines into a *farm*–construct to execute local computations on every processor. He/She may also use plain `C` routines to implement the sequential parts of the program between *start*–statements. As a third alternative, the programmer might choose to keep all data within the shared memory and use synchronous functions for their manipulation. In this case, however, he/she has to add qualifiers **sh** and **sync** to data and function definitions, respectively. Furthermore, care must be taken to move private return values of functions back into shared variables.

The present redesign and its compiler is an effort to simplify the implementation of parallel algorithms. Using existing code for the sequential parts of the algorithm, the programmer really may concentrate on the implementation of the parallel aspects.

It must be emphasized that the language is an experimental one. Future research has to investigate language extensions that support also other kinds of algorithmic skeletons like, e.g., pipelining.

References

[AKP91] F. Abolhassan, J. Keller, and W.J. Paul. On physical realizations of the theoretical PRAM model. pages 2–9, December 1991.

[BDH+] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, S. Saxena, and T. Radzik. Improved Deterministic Parallel Integer Sorting. *Information and Computation.* to appear.

[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] Chris Fraser and David Hanson. A Code Generation Interface for ANSI C. *Software – Practice and Experience*, 21(9):963–988, September 1991.

[FH91b] Chris Fraser and David Hanson. A Retargetable Compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[FH94] Chris Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin Cummings, 1994.

[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 `PRAM`s. *Future Generation Computer Systems*, 8:379–393, 1992.

[KPS94] J. Keller, W.J. Paul, and D. Scheerer. Realization of PRAMs: Processor Design. 1994.

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

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

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

[RS92] G. Rünger and K. Sieber. A Trace–Based Denotationial Semantics for the `PRAM`–Language `FORK`. Technical Report C1-1/92, Universität des Saarlandes SFB 124, 1992.

[Sch91] A. Schmitt. A Formal Semantics of `FORK`. Technical Report C1-11/91, Universität des Saarlandes SFB 124, 1991.

[Sch92] A. Schmitt. *Semantische Grundlagen der* `PRAM`*–Sprache* `FORK`. PhD thesis, Universität des Saarlandes, 1992.

[Sei93] H. Seidl. Equality of Instances of Variables in `FORK`. Technical Report C1-6/93, Universität des Saarlandes SFB 124, 1993.

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