FORK

A High–Level Language for *PRAM*s

T. Hagerup[1] A. Schmitt[2] H. Seidl[2]

April 22, 1994

22/1990

ABSTRACT

We present a new programming language designed to allow the convenient expression of algorithms for a parallel random access machine (*PRAM*). The language attempts to satisfy two potentially conflicting goals: On the one hand, it should be simple and clear enough to serve as a vehicle for human-to-human communication of algorithmic ideas. On the other hand, it should be automatically translatable to efficient machine (i.e., *PRAM*) code, and it should allow precise statements to be made about the amount of resources (primarily time) consumed by a given program. In the sequential setting, both objectives are reasonably well met by the Algol-like languages, e.g., with the *RAM* as the underlying machine model, but we are not aware of any language that allows a satisfactory expression of typical *PRAM* algorithms. Our contribution should be seen as a modest attempt to fill this gap.

Fachbereich 14
Universität des Saarlandes
Im Stadtwald
6600 Saarbrücken

---

[1]Supported by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2

[2]Supported by the Deutsche Forschungsgemeinschaft, SFB 124, TP C1

1 Introduction

A *PRAM* is a parallel machine whose main components are a set of processors and a global memory. Although every real machine is finite, we consider an ideal *PRAM* to have a countably infinite number of both processors and global memory cells, of which only a finite number is used in any finite computation. Both the processors and the global memory cells are numbered consecutively starting at 0; the number of a processor is called its *processor number* or its *index*, and the number of a memory cell is, as usual, also known as its address. Each processor has an infinite local memory and a local program counter. All processors are controlled by the same global clock and execute precisely one instruction in each clock cycle. A *PRAM* may hence also be characterized as a synchronous shared–memory MIMD (multiple-instruction multiple-data) machine.

The set of instructions available to each processor is a superset of those found in a standard *RAM* (see, e.g., [2]). The additional instructions not present in a *RAM* are an instruction LOADINDEX to load the index of the executing processor into a cell in its local memory and instructions READ and WRITE to copy the contents of a given global memory cell to a given cell in the local memory of the executing processor, and vice versa. All processors can access a global memory cell in the same step, with some restrictions concerning concurrent access by several processors to the same cell (see Section 2.5).

Among researchers working on the development of concrete algorithms, the *PRAM* is one of the most popular models of parallel computation, and the number of published *PRAM* algorithms is large and steadily growing. This is due mainly to the convenient and very powerful mechanism for inter-processor communication provided by the global memory. Curiously, there is no standard *PRAM* programming language, and each researcher, in so far as he wants to provide a formal description of his algorithms, develops his own notation from scratch. The disadvantages of this are evident:

1. At least potentially, difficulties of communication are aggravated by the lack of a common language;

2. The same or very similar definitions are repeated again and again, resulting in a waste of human time and journal space;

3. Since the designer of an algorithm is more interested in the algorithm than in the notation used to describe it, any language fragments that he may introduce are not likely to be up to current standards in programming language design.

In the wider area of parallel computing in general, much effort has gone into the development of adequate programming languages. Most of these languages, however, are intended to be used with loosely coupled multiprocessor systems consisting of autonomous computers, each with its own clock, that run mainly independently, but occasionally exchange messages. The facilities provided for inter-processor communication and synchronization are therefore based on concepts such as message exchange (Ada [15]; OCCAM [14]; Concurrent C [10]) or protected shared variables (Concurrent Pascal [12]). In particular, a global memory simultaneously accessible to all processors is not supported by such languages, and it can be simulated only with an unacceptably high overhead. While such languages may be excellent tools in the area of distributed computing, they are not suited to the description of *PRAM* algorithms.

Before we go on to discuss other languages more similar in spirit to ours, we describe what we consider to be important features of such languages. Most obviously, they must

offer a way to state that certain operations can be executed in parallel. Secondly, we want to write programs for a shared–memory machine. Therefore, the language should distinguish between shared variables, which exist only once and can be accessed by a certain group of processors, and private variables, of which there might be several instances, each residing in a different processor's private memory and possibly having a different value.

Also, the machine facilities of synchronous access to shared data, should be reflected in the language. Finally, program constructs like recursion, which are well suited for writing clear and well structured sequential programs, should be allowed to be freely combined with parallelism. Recursion is characterized by a subdivision of a given problem into a set of subproblems that can be solved independently and possibly in parallel. Each subproblem may again be worked on by several processors. Therefore, the programming language should provide the programmer with a means of generating independently working subgroups of synchronously running processors. Since the efficiency of many algorithms relies on a subtle distribution of processors over tasks, an explicit method should be available to assign processors to newly created subgroups.

A frequently used tool for indicating parallelly executable program sections is a *for* loop where all loop iterations are supposed to be executed in parallel. Such a construct is, e.g., used in extensions of sequential imperative languages like FORCE [13] and *ParC* [4]. Also textbooks about *PRAM* algorithms, e.g. [3, 11], usually employ some Algol–style notation together with a statement like **for** i:=1 **to** n **pardo** ... **endpardo** .

A different approach is taken in the language GATT [7]. In GATT all processors are started simultaneously at the beginning. During procedure calls subgroups can be formed to solve designated subproblems. However, since GATT is designed for describing efficient algorithms on processor networks, GATT lacks the concept of shared variables. Instead, every variable has to reside in the private memory of one of the processors.

For *PRAM*s, there are various examples of descriptions of recursive algorithms using an informal group concept, e.g., see [3, sect. 4.6.3, p. 101], [8, 6]. An attempt to formulate a recursive *PRAM* algorithm more precisely is made in [5]. Corresponding to the machine-level *fork* instruction of [9], a *fork* statement is introduced, which allows a given group of synchronously working processors to be divided into subgroups. This *fork* statement gave the name to our language.

The present paper embeds the *fork* statement suggested in [5] into a complete programming language. In detail, the contributions of *FORK* are the following:

- It adds a *start* construct, which allows a set of new processors with indices in a specified range to be started.
- It makes precise the extent to which the semantics guarantees synchronous program execution (and hence synchronous data access).
- Besides the implicit synchronization at the beginning of every statement, as proposed in [5], it introduces implicit splitting into subgroups at every branching point of the program where the branch taken by a processor depends on its private variables.

It is argued that the available program constructs can be freely nested. In particular, iteration and recursion are compatible both with the starting of new processors and the forking of subprocesses.

The paper is organized as follows. In Section 2 we explain the mechanism of synchronism of *FORK* together with the new constructs in *FORK* for maintaining parallelism. Moreover, we introduce the three basic concepts of *FORK*, namely the concepts of a

"logical processor", of a "group" of logical processors and of "synchronous program execution". These basic concepts are used in Section 3, which presents a description of the syntax of *FORK* together with an informal description of its semantics. Section 4 concludes with some hints on how programs of the proposed language can be compiled to efficiently running *PRAM* machine code.

It should be emphasized that although our language design aims to satisfy the needs of theoreticians, we want to provide a practical language. The language *FORK* was developed in close connection with a research project at the Saarbrücken Computer Science Department that in detail explores the possibilities of constructing a *PRAM* [1] and is going to build a prototype. We plan to write a compiler for our language that produces code for this physical machine.

Both a formal semantics of *FORK* and a more precise description of a compiler for *FORK* are in preparation.

2 An overview on the programming language *FORK*

Parallelism in *FORK* is controlled by two orthogonal instructions, namely **start** [<*expr*>..<*expr*>] and **fork** [<*expr*>..<*expr*>]. The *start* instruction can be used to readjust the group of processors available to the *PRAM* for program execution, whereas the *fork* instruction leaves the number of processors unchanged, but creates independently operating subgroups for distinct subtasks and allows for a precise distribution of the available processors among them. The effect of these instructions together with *FORK*'s concept of synchronous program execution will be explained by the examples below.

2.1 Creating new processors: The *start* statement

A basic concept of *FORK* is a logical processor. Logical processors are meant to be mapped onto the physical processors provided by the hardware architecture. However, the number of actually working logical processors may vary during program execution; also, the number of logical processors may exceed the number of physically available processors. Therefore, these two kinds of processors should not be confused. In the sequel, if we loosely speak of "processors" we always mean "logical processors". If we mean physical processors we will state so explicitly.

Every (logical) processor p owns a distinguished integer constant $\sharp$ whose value is referred to as the *processor number* of p. Also, it may have other private constants, private types, and private variables which are only accessible by itself. Objects declared as *shared* by a group of processors can be accessed by all processors of the given group.

As a first example consider the following problem. Assume that we are given a forest F with nodes $1, \ldots, N$. F is described by an array A of N integers, where $A[i] = i$ if i is a root of F, and $A[i]$ contains the father of i otherwise. For an integer constant N, the following program computes an array R of N integers such that $R[i]$ contains the root of the tree to which i belongs in F.

As in PASCAL, the integer constant N, the loop variable t, and the arrays A and R must be declared in the surrounding context; in *FORK* this declaration indicates whether variables are *shared* (as in the example) or *private* and hence only accessible to the individual processor itself.

```
...                                                              (1)
shared const N = ...;                                            (2)
shared var t : integer;                                          (3)
shared var A : array [1 .. N] of integer;                        (4)
shared var R : array [1 .. N] of integer;                        (5)
...                                                              (6)
start [1..N]                                                     (7)
    R[♯] := A[♯] ;                                               (8)
    for t := 1 to log(N) do                                      (9)
    /* log(N) denotes ?$\lceil\log_2(N)\rceil$? */               (10)
        R[♯] := R[R[♯]]                                          (11)
    enddo                                                        (12)
endstart                                                         (13)
...                                                              (14)
```

Initially there is just one processor with processor number 0. The instruction *start* [1..N] in line (7) starts processors with processor numbers $1, \ldots, N$. The corresponding instruction *endstart* stops these processors again and reestablishes the former processors. Hence the sequence of an instruction *start* [1..N] immediately followed by an instruction *start* [1..M] does not start NM processors but only M processors. An occurrence of *endstart* finishes the phase where M processors were running and again there are N processors with numbers $1, \ldots, N$.

At the machine level every instruction consumes exactly one time unit. However, the semantics of a high-level program should be independent of the special features of the translation schemes. Therefore, it should be left unspecified how many time units are precisely consumed by, e.g., an assignment statement of *FORK*.

For this reason a notion of synchronous program execution is needed which only depends on the program text itself. Again, the semantic notion of "synchronous program execution" should not be confused with the notion of a global clock of a physical *PRAM*. For example, the underlying hardware may allow different processors to execute different instructions within the same clock cycle, whereas our notion of synchronism does *not* allow for a synchronous execution of different statements. Being synchronous is a property of a set of processors. It implies that all processors within this set are at the same program point. This means that they not only execute the same statement within the same loop within the same procedure. It also means that the "history" of the recursive call to that procedure and the number of iterations within the same loop agree. There is no explicit synchronization mechanism in *FORK*. Implicit synchronization in *FORK* is done statement by statement. At the end of each statement there is an (implicit) *synchronization point*. This means that if a set of processors synchronously executes a statement sequence

$$<statement>_1 ; <statement>_2$$

the processors of this set first synchronously execute $<statement>_1$. When all processors of this set have finished the execution of $<statement>_1$ they synchronously execute $<statement>_2$. Note that within the execution of $<statement>_1$ different processors may reach different program points. Thus they may become asynchronous in between.

FORK is well structured; there are no *goto*s. Hence implicit synchronization points cannot be circumvened. Nontermination caused by infinite waiting for deviating processors is therefore not possible.

In the given example all the processors execute the same code. According to our convention they execute statement by statement synchronously. Hence, first every processor copies the value of $A[\sharp]$ to $R[\sharp]$. Recall that the constant $\sharp$ is distinct for every processor. Then all processors assign 1 to the variable t, followed by the execution of line (11). Then they assign 2 to t, and so forth. Since the upper bound for t depends on shared data only (namely on N), all processors finish the *for* loop at the same time.

Observe here that the synchronous execution of an assignment statement is subdivided into three synchronously executed steps: first, the right-hand side is evaluated; secondly, the variable corresponding to the left-hand side is determined; finally, the value of the right hand side is assigned to the variable described by the left-hand side.

In our example, in line (11), first the value of $R[R[\sharp]]$ is computed in parallel for every processor, secondly, the variable $R[\sharp]$ is determined, which receives its new value in step three.

2.2 Forming groups of processors: The *fork* statement

FORK allows free combination of parallelism and recursion. This gives rise to the second basic concept of *FORK*: a *group*. Groups are formed by a (possibly empty) set of processors. Shared variables are always shared relative to a group of processors, meaning that they can be accessed by processors within this group but not by processors from the outside.

Groups can be divided into subgroups. This is done by the *fork* construct of *FORK*. The most recently established groups are called *leaf groups*. Leaf groups play a special role in *FORK*. As a minimum, the processors within one leaf group work synchronously. Also, if new shared objects are declared, they are established as shared relative to the leaf group executing this declaration.

Every group has a group number. The group number of the most recently created group can be accessed by its members through the distinguished private integer constant @. Clearly, the values of @ are equal throughout that group. Initially, there is just one group with group number 0 which consists of the processor with processor number 0.

As an example, consider the following generic divide-and-conquer algorithm *DC*. *DC* has a recursion parameter N describing the maximal number of processors available to solve the given problem and additional parameters containing the necessary data, which for simplicity are indicated by .... Assuming that the problem size is reduced to its square root at every recursion step, *DC* may be programmed as follows:

```
procedure DC(shared const N: integer; ... );                          (1)
...                                                                   (2)
if trivial(N)                                                         (3)
    then conquer( ... )                                               (4)
    else                                                              (5)
        fork [0 .. sqrt(N)−1]                                         (6)
        @ = ♯ div sqrt(N) ;                                           (7)
        ♯ = ♯ mod sqrt(N) ;                                           (8)
            DC(sqrt(N), ... ) /* sqrt(N) denotes ?$\lceil\sqrt N\rceil$? */
                                                                      (9)
        endfork;                                                      (10)
        combine( ... )                                                (11)
endif;                                                                (12)
```

... (13)

When a leaf group reaches the *fork* instruction in line (6), a set of subgroups with group numbers $0, \ldots, sqrt(N) - 1$ is created. These newly created groups are leaf groups during the execution of the rest of the *fork* statement, which, in the example, consists of line (9). Observe that procedure calls inside a *fork* may allocate distinct instances of the same shared variable for each of the new leaf groups.

Executing the right–hand side of line (7), every processor determines the leaf group to which it will belong.

In order to make the call to a recursive procedure simpler it may be reasonable for a processor to receive a new processor number w.r.t. the chosen leaf group. In the example this new number is computed in line (8).

When the new leaf groups have been formed, the existing processors have been distributed among these groups, and the processor numbers have been redefined, the leaf groups independently execute the statement list inside the *fork* construct. In the example this consists just of a recursive call to *DC*. Clearly, the parameters of this recursive call which contain the specification of the subproblem in general depend on the value of the constant @ of its associated leaf group.

When the statements inside a *fork* statement are finished the leaf groups disappear — in the example at line (10). The original group is reestablished as a leaf group, and all the processors continue to synchronously execute the next statement (11).

2.3 Why no *pardo* statement?

There is no *pardo* statement in *FORK*. This choice was motivated by the observation that in general *pardo* is used simply in the sense of our *start*. A difference occurs for nested *pardo*s. Consider the program segment

```
for i := 1 to n pardo                (1)
    for j := 1 to m pardo            (2)
        op(i,j)                      (3)
    endpardo                         (4)
endpardo                             (5)
```

Using a similar semantics as for the *start* instruction in *FORK*, the second *pardo* simply would overwrite the first one, which means that on the whole only m processors execute line (3); moreover, the value of i in line (3) would no longer be defined. This is not the intended meaning.

Instead, two nested *pardo*s as in lines (1) and (2) are meant to start nm processors indexed by pairs (i, j). Precisely, a *pardo* statement of the form

for i := <*expr*>$_1$ **to** <*expr*>$_2$ **pardo** <*statement*> **endpardo**

where the expressions <*expr*>$_1$ and <*expr*>$_2$ and the statement <*statement*> do not use any private objects, can be simulated as follows:

```
begin                                (1)
  /* declare two new auxiliary constants in order   (2)
     to avoid double evaluation of the               (3)
     expressions ?$\NT{expr}_1$? and ?$\NT{expr}_2$? (4)
  */                                 (5)
```

```
shared const a1 = ?$\NT{expr}_1$?;                                  (6)
shared const a2 = ?$\NT{expr}_2$?;                                  (7)

/* start a2−a1+1 new processors ?$\ldots$? */                       (8)
start[a1 .. a2]                                                     (9)
  /*?$\ldots$? and distribute them among a2−a1+1 new groups */      (10)
  fork[a1 .. a2]                                                   (11)
    @ = ♯;                                                         (12)
    ♯ = 0;                                                         (13)
    /* each leaf group creates a new variable i and                (14)
       initilizes it with the group number                         (15)
    */                                                             (16)
    begin                                                          (17)
      shared var i : integer;                                      (18)

      i := @;                                                      (19)
      ?\NT{statement}?                                             (20)
    end                                                            (21)
  endfork                                                          (22)
endstart                                                           (23)
end /* of the pardo simulation */                                 (24)
```

In order to avoid redundancies we decided not to include the *pardo* construct in *FORK*.

On the other hand one may argue that the *fork* construct as provided by *FORK* is overly complicated. Using the very simple *pardo* would suffice in every relevant situation. Using *pardo* a generic divide-and-conquer algorithm may look as follows:

```
procedure DC(shared const N: integer; ... );                       (1)
...                                                                (2)
if trivial(N)                                                      (3)
    then conquer( ... )                                            (4)
    else                                                           (5)
        for i := 1 to sqrt(N) pardo                                (6)
            DC(sqrt(N), ... )                                      (7)
        endpardo;                                                  (8)
        combine( ... )                                             (9)
endif;                                                            (10)
...                                                               (11)
```

In the *pardo* version of DC beginning with one processor, successively more and more processors are started. In particular, every subtask is always supplied with one processor to solve it. Opposed to that, in the *fork* version the leaf group of processors is successively subdivided and distributed among the subtasks. The leaf group calling *DC* does not necessarily form a contiguous interval. Hence there might be subtasks which receive an empty set of processors and thus are not executed at all. In fact, this capability is essentially exploited in the order-chaining algorithm of [5]. This algorithm is not easily expressible using *pardo*s. This was one of the reasons for introducing the *fork* construct.

2.4 Forming subgroups implicitly: The *if* statement

So far we have not explained what happens if the processors of a given leaf group synchronously arrive at a conditional branching point within the program. As an example, assume that for some algorithm the processors $1, \ldots, N$ are conceptually organized in the form of a tree of height $\log(N)$. At time t, a processor should execute a procedure *op1*(♯) if its height in the tree is at most t, and another procedure *op2*(♯) otherwise. The corresponding piece of program may look like:

```
shared var t : integer;                          (1)
...                                              (2)
for t := 1 to log(N) do                          (3)
    if height(♯) <= t                            (4)
      then op1(♯)                                (5)
      else op2(♯)                                (6)
    endif                                        (7)
enddo                                            (8)
...                                              (9)
```

For every t the condition of line (4) may evaluate to *true* for some processors, and to *false* for others. Moreover, the evaluation of both *op1*(♯) and *op2*(♯) may introduce local shared variables, which are distinct even if they have the same names. Therefore, every *if–then–else* statement whose condition depends on private variables implicitly introduces two new leaf groups of processors, namely those that evaluate the condition to *true* and those that evaluate it to *false*. Both groups receive the group number of their father group, i.e. the private constants @ are not redefined.

Clearly, within each new leaf group every processor is at the same program point. Hence, they in fact can work synchronously as demanded by *FORK*'s group concept. As soon as the two leaf groups have finished the *then* and the *else* parts, respectively, (i.e., at the instruction *endif*) the original leaf group is reestablished and the synchronous execution proceeds with the next statement. *Case* statements and loops are treated analogously.

In the above example the condition of the *for* loop in line (3) depends only on the shared variable t. Therefore, the present leaf group is not subdivided into subgroups after line (3). However, this subdivision occurs after line (4). The two groups for the *then* and the *else* parts execute lines (5) and (6) in parallel, each group internally synchronously but asynchronously w.r.t. the processors of the other group. Line (7) reestablishes the original leaf group, which in return synchronously executes the next round of the loop, and so on.

The fact that we implicitly form subgroups at branching points whose conditions depend on private data allows for an unrestricted nesting of *if*s, loops, procedure calls and *fork*s.

Observe that at every program point the system of groups and subgroups containing a given processor forms a hierarchy. Corresponding to that hierarchy, the shared variables can be organized in a tree–like fashion. Each node corresponds to a group in the hierarchy and contains the shared variables relative to that group. For a processor of a leaf group all those variables are relevant that are situated on the path from this leaf group to the root. Along this path, the ordinary scoping rules hold.

For returning results at the end of a *fork* or for exchanging data between different leaf groups of processors it is necessary also to have at least in some cases a synchronous access to data shared between different subgroups of processors.

Consider the following example:

```
...                                                          (1)
shared var A : array [0..N−1] of integer;                    (2)
shared var i : integer;                                      (3)
...                                                          (4)
fork [0..N−1]                                                (5)
@ = ... ;                                                    (6)
♯ = ... ;                                                    (7)
    for i := 0 to N−1 do                                     (8)
        A[ (@+1) mod N ] := result(i,@,A)                    (9)
    enddo                                                   (10)
endfork                                                     (11)
...                                                         (12)
```

In this example the array A is used as a mail box for communication between the groups $1, \ldots, N$. The loop index i and the limits of the *for* loop of lines (8) and (9) are shared not only by the processors within every leaf group, but also between all groups generated in line (5). If (as in the example) the loop condition depends only on variables shared by all the existing groups, the semantics of *FORK* guarantees that the loop is executed synchronously throughout those groups.

Hence, the results computed in round i are available to all groups in round $i + 1$. The general rule by which every processor (and hence also the programmer) can determine the largest surrounding group within which it runs synchronously is described in detail in Section 3.4.3.

2.5 How to solve read and write conflicts

So far we have explained the activation of processors and the generation of subgroups. We left open what happens when several processors access the same shared variable synchronously. In this case, failure or success and the effect of the succeeding access is determined according to an initially fixed regime for solving access conflicts. Most *PRAM* models allow common read operations. However, we do not restrict ourselves to such a model. The semantics of a *FORK* program also may be determined, e.g., w.r.t. an exclusive read regime where synchronous read accesses of more than one processor to the same shared variable leads to program abortion. Also, several regimes for solving write conflicts are possible. For example, we may fix a regime where common writes are allowed provided all processors assign the same value to the shared variable. In this case, the *for* loop in the example above is executed successfully, whereas if we fix a regime where common writes are forbidden, a *for* loop with a shared loop parameter causes a failure of program execution.

As another example consider a regime where common writes are allowed, and the result is determined according to the processor numbers of the involved processors, e.g., the processor with the smallest number wins. This regime works fine if the processors only access shared data within the present leaf group. If processors synchronously write to variables declared in a larger group g they may solve the write conflict according to their processor numbers relative to that group g. Consider the following example:

```
...                                                          (1)
shared var A : array[0 .. N−1] of integer;                   (2)
...                                                          (3)
start[0 .. 2∗N−1]                                            (4)
    fork[0 .. 1]                                             (5)
    @ = ♯ div N;                                             (6)
    ♯ = ♯ mod N;                                             (7)
    A[♯] := result(A,@,♯)                                    (8)
    endfork                                                  (9)
endstart                                                     (10)
...                                                          (11)
```

The shared variable A is declared for some group g having after line (4) processors numbered $0, \ldots, 2*N-1$. For $n \in \{0, \ldots, N-1\}$ there are processors $p_{n,0}$ and $p_{n,1}$ of processor number n in the first and the second leaf group, respectively, that want to assign a value to $A[n]$ in line (8). This conflict is solved according to the processor numbers of $p_{n,0}$ and $p_{n,1}$ relative to group g, i.e. n and $N+n$, respectively. Hence, in line (8), the result of processor $p_{n,0}$ is stored in $A[n]$.

Observe that this scheme fails if another *start* occurs inside the *fork* statement because the original processor numbers can no longer be determined. In this case program execution fails.

In any case, the semantics of a *FORK* program is determined by the regime for solving read and write conflicts. Hence, *FORK* is intended to give the syntax and a *scheme* for the semantics of *FORK* programs.

2.6 Input/Output in *FORK*

There are at least two natural choices for designing input and output facilities for *FORK*. First, one may provide shared input/output facilities. These can be realized by (one–way infinite) shared arrays. Within this framework, synchronous I/O is realized by synchronous access to the corresponding array where conflicts are solved according to the same regime as with other accesses to shared variables (see Section 2.5).

As a second possibility one may provide private input/output facilities. These can be realized by equipping every logical processor with private input streams from which it can read, and private output streams to which it can write. The latter is the straightforward extension of PASCAL's I/O mechanism to the case of several logical processors.

Which of these choices is more suitable depends on the computing environment, e.g., the available hardware, the capabilities of an operating system and the applications. Therefore, input functions and output procedures are not described in this first draft on *FORK*.

3 Syntax and semantics of *FORK*

In this section we give a description of the *PRAM* language *FORK* by a contextfree grammar. We introduce the context conditions (i.e. correct type of variables, using only variables that are declared, ...) as well as the semantics of the language in an informal way. We assume the reader to be familiar with PASCAL or any similar language.

As usual, nonterminals are enclosed into angled brackets, and we use them also for denoting arbitrary elements of the corresponding syntactical category, e.g., an arbitrary statement may be addressed by <*statement*>. The empty word is denoted by ε.

Programs are executed by processors which are organized in a group hierarchy. For each construct of *FORK* we have to explain how the execution of this construct affects the group hierarchy, the synchronism among the processors, and the scopes of objects.

We use the following terminology. A *group hierarchy* H is a finite rooted tree whose nodes are labeled by sets of processors. A node of H is called a *group (in H)*. Assume G is a group. A *subgroup of G* is a node in the subtree with root G. A *leaf group of G* is a leaf of this tree. A processor p *is contained in a group* G if it is contained in (the label of) a leaf group of G.

In the sequel we define inductively the actual group hierarchy and the notion of a *maximally synchronous* group w.r.t. this hierarchy. According to the inductive definition all processors in a maximally synchronous group are at the same program point. Also, each leaf group is a subgroup of a maximally synchronous group. A group G is called *synchronous* if it is a subgroup of a maximally synchronous group. If we loosely speak of a synchronous group G executing, e.g., a statement, we always mean that all the processors within G synchronously execute this statement.

3.1 *Programs* and *blocks*

A *program* consists of a *program name* and a *block*. This is expressed by the following rule of the grammar:

| <*program*>⟶program <*name*> ; <*block*> .

Initially, the group hierarchy H consists just of one group numbered 0 which contains a single processor also numbered 0. This group is maximally synchronous. At the end of program execution we again have this hierarchy H.

A block contains the *declarations* of constants, data types, variables, and procedures and a sequence of *statements* that should be executed. These statements may only use constants, variables, and procedures according to PASCAL's scoping rules. A block is given by the following rule:

| <*block*>⟶**begin** <*decls*> <*stats*> **end**

A block is syntactically correct if the declarations <*decls*> and the statements <*stats*> are both syntactically correct. Declaration sequences and statement sequences are executed by maximally synchronous groups (w.r.t. the actual group hierarchy). In the sequel G always denotes such a maximally synchronous group w.r.t. the actual group hierarchy and H the subtree with root G. The execution of declarations and statements may change the group hierarchy and the synchronism — but only within the subtree H. Therefore, we only describe these changes.

3.2 *Declarations*

A sequence of declarations is either a single *declaration* or a single declaration followed by a sequence of declarations. A sequence of declarations is (syntactically) correct if all of its

declarations are correct and inside the sequence no name is declared twice. A sequence of declarations is constructed according to the following grammar rule:

$$\begin{array}{l} <\textit{decls}> \longrightarrow \varepsilon \\ \qquad\qquad \longrightarrow <\textit{declaration}> \ ; \\ \qquad\qquad \longrightarrow <\textit{declaration}> \ ; <\textit{decls}> \end{array}$$

Assume G executes a declaration sequence

<*declaration*> ; <*decls*> .

Then G first executes the declaration <*declaration*>. During the execution of <*declaration*> the synchronism among the executing processors and the group hierarchy with root G may change. However, at the end of <*declaration*>, H is reestablished, and G is maximally synchronous again w.r.t. the actual group hierarchy. Now G starts executing <*decls*>. In contrast to PASCAL, declaration sequences have to be executed since the declared objects cannot be determined before program execution (see ,e.g., Section 3.2.1). A *declaration* is either empty or it is a *constant declaration*, a *type declaration*, a *variable declaration*, a *procedure declaration*, or a *function declaration*.

Constant and variable declarations also determine whether the newly created objects are private or shared. For this we have the grammar rule:

$$\begin{array}{l} <\textit{access}> \longrightarrow \textbf{private} \\ \qquad\qquad \longrightarrow \textbf{shared} \end{array}$$

If an object is declared private, a distinct instance of this object is created for every processor executing the declaration. If an object is declared shared, each leaf group executing this declaration receives a distinct instance of the object, which is accessible by all the processors of this leaf group.

3.2.1 *Constant declarations*

A *constant declaration* is of the form

$$<\textit{declaration}> \longrightarrow <\textit{access}> \ \textbf{const} \ <\textit{name}> = <\textit{expr}>$$

Thus a constant declaration specifies the name of the constant to be declared and its value. Moreover it specifies whether this constant should be treated as a private or as a shared constant. The type of the constant <*name*> is the type of the expression <*expr*>.

Contrary to constant declarations in PASCAL, it is not (always) possible to determine the value of a constant at compile time. The difference to variables, however, is that the value of constants once determined cannot be changed any more. Therefore, constants are typically used for declaring types, e.g. for index ranges of arrays.

A constant declaration is syntactically correct if the expression <*expr*> on the right-hand side is a syntactically correct expression.

The way in which the value of a constant is determined is analogous to the method of determining the value which a variable receives during an assignment (see Section 3.4.2).

3.2.2 *Type declarations*

A *type declaration* is of the form

| <*declaration*>⟶**type** <*name*> = <*type_expr*>

Here <*type_expr*> is either a basic type (such as *integer*, *real*, *boolean* or *char*), a name denoting a type (i.e., this name has been previously declared as a type) or an "expression" for defining array and record types. The precise form of a *type expression* <*type_expr*> and its semantics will be described in Section 3.3.

A type declaration is syntactically correct if the type expression <*type_expr*> is. If <*type_expr*> depends on private data (such as the processor number ♯ or names that are declared as private constants) or private types then we treat the new type <*name*> as private. This has the effect that we are not allowed to declare any shared variable of type <*name*>. If the <*type_expr*> does not involve any private data or types, then the new type <*name*> is treated as a shared type. In this case we are allowed to declare both shared and private variables of type <*name*>.

The semantics of a type declaration is similar to that of a constant declaration. If <*type_expr*> denotes a private type then all processors of G evaluate <*type_expr*> to a type value describing the access to the components of an object of this type. These type values may be different for different processors. Now each processor associates the type name <*name*> with its type value and a flag marking this type name as a private type.

If <*type_expr*> does not depend on any private data or types, then all processors of G evaluate <*type_expr*> to a type value that may depend only on the leaf group to which the processor belongs. This means that all processors inside the same leaf group evaluate <*type_expr*> to the same type value. Now each leaf group associates the type name <*name*> with its type value and marks it as a shared type of this leaf group.

Note that it is possible to check statically (i.e., before the run of the program) whether a type used by the program is private or shared. But usually it is impossible to evaluate a type expression before running the program because type expressions can depend on input data. This enables the use of dynamic arrays.

3.2.3 *Variable declaration*

A *variable declaration* is of the form

| <*declaration*>⟶<*access*> **var** <*name*> : <*type_expr*>

A *private variable declaration* is syntactically correct if <*type_expr*> is a syntactically correct type expression. Every processor p of G creates a new distinct instance of this variable and marks it as private w.r.t. p.

A *shared variable declaration* is syntactically correct if <*type_expr*> is a syntactically correct shared type expression. This means that <*type_expr*> may not involve any private data or types. In this case, each leaf group g of G creates a new distinct instance of this variable and marks it as shared relative to group g. Note that in the case of a shared variable declaration, all processors inside the same leaf group evaluate <*type_expr*> to the same type value.

3.2.4 *Procedure and function declarations*

Procedure and *function* declarations are very similar to PASCAL's procedure and function declarations. They are constructed according to the following grammar rule:

<*declaration*> ⟶ **forward procedure** <*name*>(<*formal_par_list*>)
⟶ **forward** <*access*> **function** <*name*>
(<*formal_par_list*>):<*simple_type*>
⟶ **procedure** <*name*>(<*formal_par_list*>); <*block*>
⟶ <*access*> **function** <*name*>
(<*formal_par_list*>):<*simple_type*>; <*block*>
⟶ **procedure** <*name*>; <*block*>
⟶ **function** <*name*>; <*block*>

Forward declarations are used to construct mutually recursive functions or procedures. If there is a forward declaration of the form

forward procedure <*name*>(<*formal_par_list*>)

then the declaration sequence must contain the explicit declaration of procedure <*name*>. This means there has to be a declaration of the form

procedure <*name*>; <*block*>

Note that we do not repeat the list of formal parameters in the explicit declaration of a procedure that has been declared forward previously.

If a declaration sequence contains a declaration of the form

procedure <*name*>; <*block*>

then this procedure declaration must be preceded by a forward declaration of name <*name*> as a procedure. Forward function declarations are treated analogously.

The remaining context conditions of procedure and function declarations are the same as for procedure and function declarations in PASCAL. Only the formal parameters and the return values of functions are treated differently:

- Our language uses *const* parameters instead of PASCAL's *value* parameters.
- Every formal parameter has to be declared as private or as shared.
- In a function declaration we have to specify whether the return value should be treated as a private or as a shared value. If it is declared shared then it is shared relative to the leaf group which called the function.

A formal parameter is of the form:

<*formal_par*> ⟶ <*access*> **const** <*name*> : <*simple_type*>
⟶ <*access*> **var** <*name*> : <*simple_type*>

where <*simple_type*> is either a basic type or the name of a previously declared type.

A *formal parameter list* is of the form:

<*formal_par_list*> ⟶ ε
⟶ <*formal_par*>
⟶ <*formal_par_list*> ; <*formal_par*>

Inside a formal parameter list no name may be declared twice.

3.3 *Type expressions*

Type expressions are similar to PASCAL's type expressions. A type expression is a basic type (such as *integer*, *real*, ...), a type name, or an array or record type. This is expressed by the following grammar rules:

<*basic_type*> ⟶ **integer** | **real** | **boolean** | **char**
<*simple_type*> ⟶ <*basic_type*> | <*name*>

Here <*name*> is the name of a previously defined type.

<*type_expr*> ⟶ <*simple_type*>
⟶ **array** [<*range_list*>] **of** <*type_expr*>
⟶ **record** <*record_list*> **end**
<*range_list*> ⟶ <*const_range*>
⟶ <*range_list*> , <*const_range*>
<*const_range*> ⟶ <*expr*> .. <*expr*>

Here both expressions <*expr*> are *constant* expressions of type integer. This means that they may not involve any variables or user–defined functions.

<*record_list*> ⟶ <*record_item*>
⟶ <*record_list*> ; <*record_item*>
<*record_item*> ⟶ <*name*> : <*type_expr*>

The semantics of type expressions is the straightforward extension of PASCAL's semantics of type expressions. Observe that in contrast to PASCAL we are able to declare dynamic arrays. Also, ♯ and @ are allowed in type expressions.

3.4 *Statements*

Our language supports 7 kinds of statements:

1. assignments
2. branching statements (*if* and *case* statement)
3. loop statements (*while*, *repeat* and *for* statement)
4. procedure calls
5. blocks
6. activation of new processors (*start* statement)
7. splitting of groups into subgroups (*fork* statement)

The statements of types 1—4 are similar to their counterparts in PASCAL. But due to the fact that there are usually several processors executing such a statement synchronously there are some differences in the semantics.

The *start* statement allows the activation of new processors by need. If an algorithm needs a certain number of processors these processors are activated via *start*. When the algorithm has terminated, the new processors are deactivated and the computation continues with the processors that were active before the *start*.

The *fork* statement does not change the number of active processors, but refines their division into subgroups.

A *sequence of statements* is constructed according to the grammar rule

$$\begin{array}{l} <stats> \longrightarrow <statement> \\ \qquad\quad \longrightarrow <statement> \ ; <stats> \end{array}$$

Assume G is a maximally synchronous group executing a statement sequence

$$<statement> \ ; <stats> \ .$$

Then G first executes the statement <*statement*>. During the execution of <*statement*> the synchronism among the executing processors and the group hierarchy H with root G may change. However, at the end of <*statement*>, H is reestablished, and G is maximally synchronous again w.r.t. the actual group hierarchy. Then G starts executing <*stats*>. We now give the syntax and semantics of statements.

3.4.1 *The empty statement*

The *empty statement* is of the form

$$<statement> \longrightarrow \varepsilon$$

It has no effect on the semantics of a program.

3.4.2 *The assignment statement*

The *assignment statement* is of the form

$$<statement> \longrightarrow <expr> := <expr>$$

An assignment is syntactically correct if the following conditions hold:

- The expression <*expr*> on the left–hand side denotes a variable of type t or the name of a function returning a value of type t. In the latter case the assignment has to occur in the statement sequence of that function.
- The expression on the right–hand side denotes a value of type t.

First all processors of G synchronously evaluate the two expressions (see Section 3.5). Each processor of G evaluates the left–hand side expression to a private or shared variable. Then the processors of G synchronously bind the value of the right–hand side expression to this variable. The effect of this is defined as follows:

1. if the variable is private, then after the assignment it contains the value written by the processor.
2. if the variable is shared, then the regime for solving write conflicts (see Section 2.5) determines the success or failure of the assignment, and, in case of success, the value to which the variable is bound.

Example Assume that four processors numbered 0 — 3 synchronously execute the assignment

$$x := @ + \sharp$$

where x is a variable of type integer. The value of x after the assignment depends on whether x is declared as private or as shared. We list some cases below.

1. x is a private variable. In this case there are four distinct incarnations of x. Then after the assignment x contains for each processor the value of the expression $@ + \sharp$.

2. the four processors form a leaf group and the variable x is shared relative to that leaf group. In this case the four processors write to the same variable and this write conflict is solved according to the regime for solving write conflicts (see Section 2.5).

3. the four processors form two different leaf groups (i.e. processors 0 and 1 are in the first one, 2 and 3 in the second one). Each leaf group has a distinct instance of the variable x. Then processors 0 and 1 write to the same variable and so do processors 2 and 3. The regime for solving write conflicts determines the value of the (two distinct) variables x after the assignment.

Above we have described the assignment of basic values. Our language supports assignment of structured values (e.g., $a := b$, where a and b are arrays of the same type), which is carried out component by component synchronously.

3.4.3 *The if statement*

The *if* statement is constructed according to the following grammar rules:

<*statement*> ⟶ **if** <*expr*> **then** <*stats*> <*else_part*>
<*else_part*> ⟶ **endif**
⟶ **else** <*stats*> **endif**

The *if* statement is syntactically correct if <*expr*> denotes an expression of type boolean and <*stats*> is a syntactically correct sequence of statements.

All processors of G first evaluate the expression <*expr*> synchronously. Depending on the result of this evaluation the processors synchronously execute the statements of the *then* or of the *else* part, respectively. Since different processors may evaluate <*expr*> to different values, we cannot make sure that all of them continue to work synchronously. The group hierarchies and the new maximally synchronous groups executing the *then* and the *else* part (if present), respectively, are determined according to the constants, variables and functions on which the expression <*expr*> "depends". Precisely, we say <*expr*> *depends* on a variable x if x occurs in <*expr*> outside any actual parameter of a function call. The case of constants and functions is analogous. We have to treat two cases:

1. *the expression does not depend on any private variables, constants, or functions.*

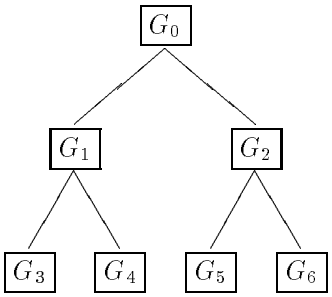
   In this case we do not change the group hierarchy H. Only the synchronism among the processors may change. Consider a processor p of G. We choose the maximal group g_p of H which satisfies the following conditions:

- g_p is a subgroup (not necessarily a proper one) of G.
- p is contained in g_p.
- The condition <*expr*> does not depend on any shared variables or other shared data relative to a proper subgroup of g_p. Note that the return value of a shared function is a shared datum only relative to a leaf group (see Section 3.5).

Under these conditions all processors in g_p evaluate <*expr*> to the same value. Therefore all these processors choose the same branch of the *if* statement and hence are at the same program point.

Note that a processor outside of g_p runs asynchronously with the processor p even if it evaluates expression <*expr*> to the same value.

Example Assume that during program execution we have obtained the following group hierarchy.



and that all processors execute synchronously the *if* statement

if $x = 5$ **then** S_1 **else** S_2 **endif**

where an instance of x is a shared variable relative to G_1, and another instance of x is a shared variable relative to G_2. Then the processors of group G_3 work synchronously with the processors of group G_4 and the same holds for groups G_5 and G_6, respectively. The processors of group G_4 work asynchronously with the processors of group G_6 even if the two instances of variable x contain the same value. □

When a processor of G has finished the execution of <*stats*> it waits until the other processors of G have finished their statement sequences. When all processors of G have arrived at the end of the *if* statement, G becomes maximally synchronous again. This means that even if two processors of G work asynchronously inside the *if* statement they become synchronous again after the *if* statement.

2. *The expression depends on private variables, constants, or functions*

In this case we cannot even be certain that all processors inside the same leaf group evaluate <*expr*> to the same value. In this case both the group hierarchy H and the synchronism are changed as follows.

Each leaf group of G generates two new leaf groups: The first one contains all processors of the leaf group that evaluate <*expr*> to *true*, and the second one contains the rest. All new leaf groups obtain the group number of their father group. The processors inside the same new subgroup work synchronously, while the processors of different subgroups work asynchronously.

Again, when a processor reaches the end of the *if* statement, it waits until the other processors reach this point. When all the newly generated leaf groups have terminated the execution of the *if* statement, i.e., when all processors have reached *endif*, the leaf groups are removed. The original group hierarchy H is reestablished and G is again maximally synchronous.

3.4.4 *The case statement*

A *case statement* is constructed according to the grammar rules

<*statement*> ⟶ **case** <*expr*> **of** <*case_list*> <*end_case*>
<*case_list*> ⟶ <*case_item*>
⟶ <*case_list*> ; <*case_item*>
<*end_case*> ⟶ **endcase**
⟶ **else** <*stats*> **endcase**
<*case_item*> ⟶ <*expr*> : <*stats*>
⟶ <*range*> : <*stats*>
<*range*> ⟶ <*expr*> .. <*expr*>

A *case statement* is syntactically correct if the expression <*expr*> is of type integer. Moreover, in the list <*case_list*> of case items all expressions have to be expressions of type integer.

The semantics of the *case statement* is similar to the semantics of PASCAL's case statement. The group hierarchy and the synchronism among the processors executing the different branches of the *case statement* is determined in the same fashion as for the *if statement* (see Section 3.4.3).

3.4.5 *The while statement*

The *while statement* is of the form

<*statement*> ⟶ **while** <*expr*> **do** <*stats*> **enddo**

It is syntactically correct if <*expr*> is an expression of type boolean and <*stats*> is a syntactically correct sequence of statements.

All processors of G first evaluate <*expr*> to a boolean value. Those processors that evaluate <*expr*> to *true* start to execute the statements <*stats*>. The others wait until all processors have finished executing the *while* statement. The group hierarchy and the synchronism among the processors executing <*stats*> is determined in the same way as for the *if* statement (see Section 3.4.3). When the processors have finished the execution of <*stats*> they again try to execute the *while statement*. Those that evaluate <*expr*> to *true* will again execute <*stats*> while the others have terminated the *while statement*. Thus the execution of the *while* statement

while <*expr*> **do** <*stats*> **enddo**

is semantically equivalent to the execution of

if <*expr*> **then**
  <*stats*> ;
  while <*expr*> **do** <*stats*> **enddo**
endif

3.4.6 *The repeat statement*

The *repeat statement* is of the form

| <*statement*>⟶**repeat** <*stats*> **until** <*expr*>

It is syntactically correct if <*stats*> is a syntactically correct sequence of statements and <*expr*> is an expression of type boolean.

The *repeat statement*

repeat <*stats*> **until** <*expr*>

is just an abbreviation for the sequence of statements

<*stats*>;
while not (<*expr*>) **do** <*stats*> **enddo**

3.4.7 *The for statement*

The *for statement* is of the form

| <*statement*>⟶**for** <*name*> := <*expr*> **to** <*expr*> <*step*> **do**
| <*stats*> **enddo**
| <*step*> ⟶ε
| ⟶**step** <*expr*>

It is syntactically correct if

1. <*name*> is a variable of type integer,
2. the expressions <*expr*> are of type integer,
3. inside <*stats*> there are no assignments to <*name*>, nor is <*name*> used as a var parameter in procedure and function calls. This means that inside <*stats*>, <*name*> is treated like a constant of type integer.

We give the semantics of the *for statement* by reducing a *for statement* to a while loop, or more precisely: we rewrite a program that uses $n \geq 1$ *for* statements into a program that contains only $n - 1 \geq 0$ *for* statements. By iterating this process we are able to eliminate all *for* loops. Observe that, in contrast with the simulation of *for* by *while* in a sequential context, we have to be careful about where auxiliary variables are declared.

1. The statement

   for <*name*> := <*expr*>$_1$ **to** <*expr*>$_2$ **do**
   <*stats*>
   enddo

   is an abbreviation for

   <*name*> := <*expr*>$_1$;
   <*name*>$_2$:= <*expr*>$_2$;
   while <*name*> <= <*name*>$_2$ **do**
   <*stats*> ;
   <*name*> := <*name*> + 1
   enddo

2. The statement

   for <*name*> := <*expr*>$_1$ **to** <*expr*>$_2$ **step** <*expr*>$_3$ **do**
   <*stats*>
   enddo

   is an abbreviation for

   <*name*> := <*expr*>$_1$;
   <*name*>$_2$:= <*expr*>$_2$;
   <*name*>$_3$:= <*expr*>$_3$;
   if <*name*>$_3$ > 0 **then**
   while <*name*> <= <*name*>$_2$ **do**
   <*stats*> ;
   <*name*> := <*name*> + <*name*>$_3$
   enddo
   else
   while <*name*> >= <*name*>$_2$ **do**
   <*stats*> ;
   <*name*> := <*name*> + <*name*>$_3$
   enddo
   endif

Here <*name*>$_2$ and <*name*>$_3$ denote variables of type integer such that the following holds:

- The names are new, i.e. these names are not used anywhere in the program containing the *for* statement.
- In the simulating program these names are declared in the same declaration sequence as the index <*name*>.
- If <*name*> is declared to be private (shared), then <*name*>$_2$ and <*name*>$_3$ are declared private (shared).

3.4.8 *Procedure calls*

Syntax and semantics of procedure calls are similar to those of PASCAL. There are some slight differences due to the fact that processors can access two different kinds of objects: shared and private objects. This imposes some restrictions on the actual parameters of a procedure call.

The syntax of a procedure call is given by the following grammar rules:

<*statement*> ⟶ <*name*> (<*expr_list*>)
<*expr_list*> ⟶ ε
⟶ <*expr*>
⟶ <*expr_list*> , <*expr*>

A procedure call is syntactically correct if the following conditions hold:

- <*name*> has to be the name of a procedure declared previously.
- the list of actual parameters (<*expr_list*>) and the list of formal parameters have to match, i.e.,

1. Both lists have the same length.
2. Corresponding actual and formal parameters are of the same type.
3. If a formal parameter is a shared–var–parameter then the corresponding actual parameter has to be a variable which does not depend on any private object, e.g., ar[♯] is not allowed as an actual shared var parameter even if ar is a shared array.
4. If a formal parameter is a private–var–parameter then the corresponding actual parameter can be a private or a shared variable.
5. If a formal parameter is a shared–const–parameter then the corresponding actual parameter may denote a private or a shared value.
6. If a formal parameter is a private–const–paramenter then the corresponding actual parameter may denote a private or a shared value.

The case where a formal shared–var–parameter is bound to a private variable is explicitly excluded since it allows some processor to modify the contents of this private variable, which may belong to a different processor. In contrast, in the case of a formal shared–const–parameter the value of that formal parameter during the execution of the procedure is determined by the regime for solving write conflicts. This is done in the same way as when determining the value of a shared variable after the assignment of a private value (see Section 3.4.2).

Assume the maximally synchronous group G executes a procedure call

$$<name>(<expr>_1, \ldots, <expr>_n) \ .$$

First the actual parameters $<expr>_1, \ldots, <expr>_n$ are synchronously evaluated and bound to the corresponding formal parameters synchronously from left to right. Then G synchronously executes the procedure block, i.e. the declarations and the statements of that procedure.

3.4.9 Blocks as statements

| <*statement*>⟶<*block*>

A statement which is a block enables the declaration of new objects that are valid only inside <*block*>.

3.4.10 *The start statement*

The *start statement* is used for activating new processors. The syntax is given by the rule:

| <*statement*>⟶**start** [<*range*>] <*stats*> **endstart**

The *start statement* is syntactically correct if the expressions of <*range*> are of type integer and do not depend on any private data. Moreover <*stats*> may not use any private types, variables or constants, except @ and ♯, that are declared outside the *start statement*.

When the processors of G execute the statement

start [$<expr>_1..<expr>_2$] <*stats*> **endstart**

they first evaluate the range <*expr*>$_1$ .. <*expr*>$_2$. Since this range does not depend on any private data this gives for all processors in the same leaf group g the same range $v_{g,1}, \ldots, v_{g,2}$. At every leaf group g of G a new leaf group is added which contains $v_{g,2} - v_{g,1} + 1$ new processors numbered with the elements of $\{v_{g,1}, \ldots, v_{g,2}\}$. The group numbers of the new leaf groups are the same as for their father groups. G remains maximally synchronous. Now G executes <*stats*>, which means that the new processors of the new leaf groups execute <*stats*> synchronously. When G reaches *endstart*, the leaf groups are removed, i.e. the original hierarchy H is reestablished.

3.4.11 *The fork statement*

The *fork statement* is used to generate several new leaf groups explicitly. The new leaf groups obtain new group numbers and the processors inside the new leaf groups are renumbered.

The syntax of the *fork statement* is given by the following grammar rules:

<*statement*> ⟶ **fork**[<*range*>] <*new_values*> ; <*stats*> **endfork**
<*new_values*> ⟶ <*new_group*> ; <*new_proc*>
⟶ <*new_proc*> ; <*new_group*>
<*new_group*> ⟶ @ = <*expr*>
<*new_proc*> ⟶ ♯ = <*expr*>

The *fork statement* is syntactically correct if the expressions <*expr*> are of type integer and the expressions of <*range*> do not depend on any private data.

Assume the processors of G execute the statement

fork[<*expr*>$_1$ .. <*expr*>$_2$]
@ = <*expr*>$_3$;
♯ = <*expr*>$_4$;
<*stats*>
endfork

First each processor p of G evaluates the expressions <*expr*>$_1, \ldots,$ <*expr*>$_4$. Since the expressions <*expr*>$_1$ and <*expr*>$_2$ do not depend on any private data, all processors within the same leaf group g of G evaluate <*expr*>$_1$ and <*expr*>$_2$ to the same values $v_{g,1}$ and $v_{g,2}$ respectively. At every leaf group g of G $v_{g,2} - v_{g,1} + 1$ new leaf groups are added which are numbered with the elements of $\{v_{g,1}, \ldots, v_{g,2}\}$. The new leaf group with number i is labeled by the subset of those processors of g which evaluate <*expr*>$_3$ to i. Each of these processors obtains the value of <*expr*>$_4$ as its new processor number. G remains maximally synchronous and executes <*stats*>. When G reaches *endfork*, the new leaf groups are removed, i.e., the original group hierarchy is reestablished.

3.5 *Expressions*

Syntax and semantics of expressions are similar to those of PASCAL. There are just two new predefined constants: the processor number ♯ and the group number @, which both are private constants of type integer. The context conditions for function calls are analogous to those for procedure calls (see Section 3.4.8). Every processor of the maximally synchronous group G evaluates the expression and returns a value. The return value of a shared function is determined according to the choosen regime for solving write conflicts seperately for each

leaf group of G and is treated as a shared object of this leaf group. Note that the evaluation of expressions may cause read conflicts, which are solved according to the choosen regime for solving read conflicts.

Example

```
...
shared var a: array[1 .. 10] of integer;
...
... := a[3]+3
...
```

Determining the variables corresponding to the subexpression a[3] does not cause a read conflict, whereas determining the values of these variables may cause read conflicts. □

4 Implementation

In this section we sketch some ideas showing that programs of *FORK* can not only be translated to semantically equivalent *PRAM* code, but also to code that runs efficiently. These considerations are supposed to be useful both to theoreticians and to compiler writers, who may have different realizations of *PRAM*s available, possibly without powerful operating systems for memory management and processor allocation.

Our basic idea for compiling *FORK* is to extend the usual stack-based implementation of recursive procedure calls of, e.g., the P-machine [16] by a corresponding regime for the shared data structures, a synchronization mechanism, and a management of group and processor numbers. Hence, here we address only the following issues:

- creating new subgroups;
- synchronization;
- starting new processors with *start*.

4.1 Creating new subgroups

The variables which are shared relative to some group have to be placed into some portion of the shared memory of the *PRAM*, which is reserved for this group. Therefore, the crucial point in creating new subgroups is the question of how the subgroups obtain distinct portions of shared memory. There are (at least) two ways to do this with little computational overhead:

1. by address arithmetic, as suggested in [5],
2. taking into account that in practice the available shared memory always is finite, by equally subdividing the remaining free space among the newly created subgroups.

The first method corresponds to an addressing scheme where the remaining storage is viewed as a two–dimensional matrix. Its rows are indexed by the group numbers, whereas the column index gives the address of a storage cell relative to a given group. For the second method the role of rows and columns are simply exchanged. In both cases splitting into subgroups can be executed in constant time. Also, the addresses in the physical shared memory can be computed from the (virtual) addresses corresponding to the shared

memory of a subgroup in constant time. This memory allocation scheme is well suited to group hierarchies with balanced space requirements. It may lead to an exponential waste of space in other cases. Consider the following while loop, whose condition depends on private variables:

```
while cond(♯) do                (1)
      work(♯)                   (2)
enddo                           (3)
```

Whenever the group of synchronously working processors executes line (1), it is subdivided into two groups, one consisting of the processors that no longer satisfy *cond*(♯), and one consisting of the remaining processors. Hence, the first group needs no shared memory space (besides perhaps some constant amount for organizational reasons); a fair subdivision into two equal portions would unnecessarily halve the space available to the second group to execute *work*(♯) of line (2).

However, there is an immediate optimization to the above storage distribution scheme: we attach only a fixed constant amount of space to groups of which it is known at compile time that they do not need new shared memory space, and subdivide the remaining space equally among the other subgroups.

This optimization clearly can be performed automatically for loops as in the given example, but also for one-sided *if*s, or *if*s where one alternative does not involve blocks with a non–empty declaration part.

4.2 Synchronization

In order to reestablish a group g, the runtime system has to determine when all the subgroups of group g have finished. This is the termination detection problem for subgroups.

If all processors run synchronously, no explicit synchronization is necessary. In the general case where the subgroups of group g run asynchronously, there are the following possibilities for implementing termination detection:

1. Use of special hardware support such as a unit–time *fetch&add* operation which allows the processors within a group to simultaneously add an integer to a shared variable.

2. Static analysis (possibly assisted by user annotation); most of the *PRAM* algorithms published in the literature are of such a simple and regular structure that the relative times of execution sequences can be determined in this way.

3. Use of a termination detection algorithm at runtime. The latter is always possible; however, complicated programs cheating a static analyzer will be punished by an extra loss of efficiency.

4.3 Starting new processors

The following method which is analogous to the storage distribution scheme works only for concurrent–read machines with a finite number of processors. Before running the program, all processors are started, all of them with processor number $\sharp = 0$. If in a subsequent *start* statement of the program fewer processors are started than what are physically available in the leaf group executing this statement, then several physical processors may remain "identical", i.e., receive the same new processor number. These identical processors elect

a "leader". All of them execute the program but only the leader is allowed to perform write operations to shared variables. Consider the following example. Assume that we are given 512 physical processors.

```
start [0 .. 127]                                    (1)
    if ♯ < 64 then                                  (2)
        start [0 .. 212]                            (3)
            compute(212)                            (4)
        endstart                                    (5)
    endif                                           (6)
endstart                                            (7)
```

Before line (1), all the 512 physical processors are started. After line (1), there are always four processors having the same processor number ♯. Having executed the condition of line (2) all the processors whose processor number is less than 64 enter the *then* part: these are 256. All of them are available for the *start* instruction of line (3), where they receive the new numbers $0, \ldots, 212$. Two physical processors are assigned to each of the first 43 logical processors whereas one physical processor is assigned to each of the remaining 170 logical processors. The original processor identities are put onto the private system stack. When the *endstart* in line (5) is reached, the processors reestablish their former processor numbers.

If more processors are started than physially available in the present group, then every processor within that group has to simulate an appropriate subset of the newly started processors.

In both cases *start* can be executed in constant time by every leaf group consisting of a contiguous interval of processors: this is the case, e.g., for *start*s occuring in the statement sequence of the toplevel block. Thus, a programming style is encouraged where the logical processors necessary for program execution are either started at the beginning, i.e., before splitting the initial group into subgroups, or are started in a balanced way by contiguous groups.

To maximally exploit the resources of the given *PRAM* architecture, a programmer may wish to write programs which use different algorithms for different numbers of physically available processors. Therefore, a (shared) system constant of type integer should be provided, whose value is the number of physical processors available on the given *PRAM*. This allows programs to adopt themselves to the underlying hardware.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. Überblick über PRAM–Simulationen und ihre Realisierbarkeit. In *Proceedings der Tagung der SFBs 124 und 182 in Dagstuhl, Sept. 1990*, Informatik Fachberichte. Springer Verlag, to appear.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, Reading Massachusetts, 1974.

[3] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[4] Y. Ben–Asher, D.G. Feitelson, and L. Rudolph. ParC — an extension of C for Shared Memory Parallel Processing. Technical report, The Hebrew University of Jerusalem, 1990.

[5] 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.

[6] A. Borodin and J.E. Hopcroft. Routing, merging and sorting on parallel models of computation. *J. Comp. Sys. Sci. 30*, pages 130 – 145, 1985.

[7] M. Dietzfelbinger and F. Meyer auf der Heide (ed.). Das GATT-Manual. In: Analyse paralleler Algorithmen unter dem Aspekt der Implementierbarkeit auf verschiedenen parallelen Rechenmodellen. Technical report, Universität Dortmund, 1989.

[8] F.E. Fich, P. Ragde, and A. Widgerson. Simulations among concurrent-write PRAMs. *Algorithmica 3*, pages 43 – 51, 1988.

[9] S. Fortune and J. Wyllie. Parallelism in random access machines. In *10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[10] N.H. Gehani and W.D. Roome. Concurrent C. In N.H. Gehani and A.D. McGettric, editors, *Concurrent Programming*, pages 112–141. Addison Wesley, 1988.

[11] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[12] P.B. Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engeneering 1(2)*, pages 199–207, June 1975.

[13] H.F. Jordan. Structuring parallel algorithms in a MIMD, shared memory environment. *Parallel Comp. 3*, pages 93–110, 1986.

[14] Inmos Ltd. *OCCAM Programming Manual.* Prentice Hall, New Jersey, 1984.

[15] United States Department of Defense. Reference manual for the Ada programming language. ANSI/MIL-STD-1815A-1983.

[16] St. Pemberton and M. Daniels. *Pascal implementation: The P4 compiler.* Ellis Horwood, 1982.