UNIVERSITÄT TRIER
Mathematik / Informatik

Forschungsbericht Nr. 97–12

ISSN 0944-0488

Practical PRAM Programming with Fork95

A TUTORIAL

Christoph W. Keßler

Universität Trier, FB IV – Informatik
D-54286 Trier, Germany
kessler@psi.uni-trier.de
http://www.informatik.uni-trier.de/~kessler

This report contains the draft of a chapter in the forthcoming textbook Practical PRAM Programming *by Jörg Keller, Christoph Keßler, and Jesper Träff. This chapter describes thoroughly the PRAM programming language Fork95; it contains many example programs, and covers several parallel algorithmic techniques. Familiarity of the reader with the PRAM model of parallel computation is assumed.*

September 9, 1997

Contents

1 Fork95

Fork95 is a high–level programming language for PRAMs. Its programming model is summarized in Figure 1.1.

Fork95 is based on ANSI C [ANS90]. The new language constructs allow to control parallel program execution, e.g. by organizing processors into groups, managing shared and private address subspaces, and handling various levels of synchronous parallel program execution. By carefully chosen defaults, existing (sequential) C codes can be easily integrated into Fork95 programs. Furthermore, direct access to hardware-supplied multiprefix operations is enabled.

In contrast to most other programming languages, the philosophy of Fork95 is not to start with a sequential thread of computation and successively spawn new ones but to take the explicitly given set of available processors — which is active from the beginning and remains constant during parallel program execution — and distribute them explicitly across the problem sizes. Virtual processor emulation is not supported; each processor works on exactly one process all the time. It has turned out that the implementational overhead to maintain *synchronously* executing *virtual* processors is a too high price to pay for a potential gain of comfort of programming. On the other hand, execution of an arbitrary number of asynchronous tasks can be easily implemented in Fork95.

The default mode of parallel program execution is the asynchronous mode. Sections 1.1 to 1.3 introduce the Fork95 language constructs for asynchronous programming and give some suggestions of how they may be used.

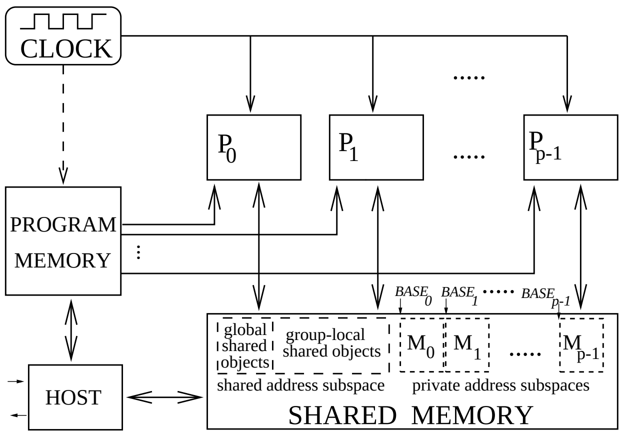


Figure 1.1 Block diagram of Fork95's programming model. Fork95 assumes that private address subspaces are embedded into the shared memory by handling private addresses relative to a BASE pointer. All processors receive the same clock signal, thus the machine is synchronous on the machine instruction level. The memory access time is uniform for each processor and each memory location. Each processor works on the same node program (SPMD mode).

The exact synchronicity available at the machine instruction level is made available at the source language level. This is technically made possible by Fork95's group concept: Each processor belongs to exactly one (active) group of processors. In synchronous mode of execution, Fork95 maintains the *synchronicity invariant* which says that all processors in the same active group work exactly synchronous on the operator level, i.e. their program counters are equal at any time (quasi–SIMD–execution). In general, this eliminates "race conditions", i.e. it enables deterministic behaviour of concurrent write and read accesses to shared memory locations without additional protection mechanisms (e.g., locks). Where control flow may diverge due to private branch conditions, the current group is split correspondingly into subgroups; exact synchronicity is only maintained within each subgroup. The subgroups are active until execution of all branches is finished; then the parent group is reactivated. Group splitting can be nested; thus, at any time of program execution, the group hierarchy forms a logical tree. In asynchronous mode, the synchronicity invariant is not enforced. Section 1.4 shows how to switch from synchronous to asynchronous mode and vice versa. Section 1.5 discusses the group concept and the synchronous mode of execution.

A dual point of view to this hierarchical group concept is the concept of *parallel processes*: In contrast to common sequential processes known from sequential programming systems, a parallel process is executed concurrently by the processors of a group. Determinism is guaranteed by exact synchronicity. Group splitting is analogous to spawning more processes — where now the child processes are parallel processes with a correspondingly lower degree of available parallelism since the set of processors is also subdivided across the subgroups.

Like C, Fork95 offers pointers and heaps, which are discussed in Section 1.6. We describe some problems and common pitfalls in Section 1.7; Section 1.8 concludes with a survey of related parallel programming languages.

The description of the compilation issues of Fork95 is given in [KS95a, KS97a].

1.1 First steps

1.1.1 Processors, Processor IDs, and "Hello World"

A key feature of Fork95 is the fact that the user program (i.e., the `main()` function) is executed by all available processors of the PRAM partition on which the program has been started. This number of available processors can be inspected at run time by the global shared constant integer variable `__STARTED_PROCS__`.

Moreover, each processor holds a global private constant integer variable `__PROC_NR__` which contains its processor ID at run time. The processors are consecutively numbered from 0 to `__STARTED_PROCS__`-1.

We begin with a simple example program.

```
#include <fork.h>
#include <io.h>

async int main()
{
```

```
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 prS("Hello world\n");
}
```

The header file `<fork.h>` is required for any Fork95 program. The header file `<io.h>` contains Fork95's I/O–routines such as `printf()`. `prS()` is a routine that prints a string.

Save the program as `hello.c` and compile it by the Fork95 compiler. This is done by the command

```
fcc -FORK -m -I$FORKDIR/include hello.c
```

The option `-FORK` tells the Fork95 compiler, `fcc`, that the file `hello.c` to be compiled is a Fork95 source file rather than a C source file. The other options will be explained later.

Now call the PRAM simulator:

```
pramsim hello
```

To execute the program, type

```
g
```

for `go`, and wait until the simulator stops. What now appears on the screen may be somehow unexpected:

```
PRAM P0 = (p0, v0)> g
Hello world
Hello world
Hello world
EXIT: vp=#1, pc=$000001ea
EXIT: vp=#2, pc=$000001ea
EXIT: vp=#3, pc=$000001ea
Stop nach 5486 Runden, 731.467 kIps
28d7 8121C04F  ADD     R3, 4f, R18
PRAM P0 = (p0, v0)>
```

Where is the output of processor 0 that should print the number of processors? In fact, processor 0 is still working on processing its output string, while the other three processors already finished their tasks, reached the program end (the end of `main()`) and therefore caused the simulator to stop. You can see this immediately by typing another `g`. The simulation continues; processors 1, 2 and 3 have nothing to do and crash:

```
PRAM P0 = (p0, v0)> g
error: vp=#1, pc=$000000a0, reason=global bus error
error: vp=#2, pc=$000000a0, reason=global bus error
error: vp=#3, pc=$000000a0, reason=global bus error
Stop nach 3 Runden, Inf kIps
28d9 09294000  LDG     R18, 0, R18
PRAM P0 = (p0, v0)>
```

Another `g` continues the simulation until also processor 0 reaches the end of the program:

```
PRAM P0 = (p0, v0)> g
Program executed by 4 processors

Hello world
EXIT: vp=#0, pc=$000001ea
```

1.1.2 The barrier statement

It would be nicer if all processors terminate the program at the same time. Processor 0 needs much longer than the other processors, thus these should be forced to wait at the end of the program before `EXIT`ing it. For this purpose we use the

```
barrier
```

statement which forces all processors to wait until the last one (here processor 0) has reached this program point. Now our program looks like

```
#include <fork.h>
#include <io.h>

main()
{
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 prS("Hello world\n");
 barrier;
}
```

The output generated by the SB-PRAM simulator for this program is

```
PRAM P0 = (p0, v0)> g
Hello world
Hello world
Hello world
Program executed by 4 processors

Hello world
EXIT: vp=#0, pc=$000001f8
EXIT: vp=#1, pc=$000001f8
EXIT: vp=#2, pc=$000001f8
EXIT: vp=#3, pc=$000001f8
Stop nach 11222 Runden, 641.257 kIps
01f8 18137FFF  POPNG   R6, fffffff, R1
PRAM P0 = (p0, v0)>
```

Although now all processors terminate the program at the same time, the output still may look better if the message indicating the number of processors would appear before all the `Hello world` messages on the screen (these of processor 1, 2 and 3 come earlier since `prS()` needs much less time to print a string than `printf()` takes to compose and print its message.) We achieve this by introducing another `barrier` statement after processor 0's solo:

```
#include <fork.h>
#include <io.h>

async int main()
{
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 barrier;
 printf("Hello world\n");
 barrier;
}
```

In fact the second `barrier` statement immediately before the end of `main()` would be now not really be necessary any more on the SB-PRAM to achieve simultaneous program exiting, since printing the same string takes exactly the same time on all processors. Nevertheless it is good programming style to not rely too much on such architecture–specific features; moreover, at least on a Multiprefix PRAM, the overhead for one `barrier` is marginal, as we have shown in [KS95a]. As output we obtain

```
PRAM P0 = (p0, v0)> g
Program executed by 4 processors

Hello world
Hello world
Hello world
Hello world
EXIT: vp=#0, pc=$000001fc
EXIT: vp=#1, pc=$000001fc
EXIT: vp=#2, pc=$000001fc
EXIT: vp=#3, pc=$000001fc
Stop nach 11242 Runden, 642.400 kIps
01fc 18137FFF  POPNG   R6, fffffff, R1
PRAM P0 = (p0, v0)>
```

1.2 Shared and private variables

As described earlier, the entire shared memory of the PRAM (see Figure 1.1) is partitioned — according to the programmer's wishes — into private address subspaces (one for each processor) and a shared address subspace which may be again dynamically subdivided among the different processor groups. Accordingly, variables are classified either as *private* or as *shared*. The term “shared” always relates to the processor group that defined that variable. We will define the group concept formally later in Section 1.5; for now, it is sufficient to state that a “shared” variable exists exactly once and is accessible by all processors. Also, for now, shared variables must be declared globally; declaring local shared variables requires synchronous program execution which will be addressed later.

1.2.1 Declaration

A shared variable exists only once in the shared memory subspace of the group of processors that executed the declaration. To declare a shared integer variable `v`, we write

```
sh int v;
```

Here `sh` is a storage class qualifier indicating that the *sharity* of `v` is "shared". The sharity may also be specified after the type declaration, but it is convenient programming style to specify it first.

For each variable declared as "private", one copy resides in the private address subspace of each processor that executes the declaration. Private variables are declared as follows

```
pr int w;
int q;
```

where the sharity declarator `pr` is redundant since "private" is the default.

1.2.2 System variables

We already mentioned that the *total number of started processors* is accessible through the constant shared integer variable

```
__STARTED_PROCS__
```

and the *processor ID* of each processor is accessible through the constant private integer variable

```
__PROC_NR__
```

With these, it becomes straightforward to write a simple dataparallel loop:

```
#include <fork.h>
#define N 30
sh int sq[N];
sh int p = __STARTED_PROCS__;

main() {
  pr int i;
  for (i=__PROC_NR__; i<N; i+=p)
     sq[i] = i * i;
  barrier;
}
```

Since the processor IDs are numbered consecutively from 0 to $p-1$, the private variable `i` on processor j loops over the values j, $j+p$, $j+2p$, etc. Thus, this loop computes in the shared array `sq` all squares of the numbers in the range from 0 to $N-1$.

1.2.3 Concurrent–Write conflict resolution

It is not part of the language to define what happens if several processors write the same (shared) memory location in the same cycle. Instead, Fork95 inherits the write conflict resolution method from the target machine. In the case of the SB-PRAM, the processor with highest hardware processor ID[1] will win and write its value (PRIORITY–CRCW-PRAM). However, as several other write conflict resolution schemes (like ARBITRARY) are also used in theory, meaningful Fork95 programs should not be dependent on such specific conflict resolution schemes; there are better language elements (multiprefix instructions, see Section 1.3.1) that cover practically relevant applications for concurrent write.

1.2.4 Return values of functions

The return value of a non–`void` function is always private. It can nevertheless be made accessible to all processors by assigning it immediately to a shared variable, e.g. in

```
sq[N-1] = square( N-1 );
```

the value returned by function `square()` is assigned to the shared array element `sq[N-1]` which is accessible to all processors.

1.3 Controlling asynchronous access to shared resources

1.3.1 Atomic multiprefix instructions

The SB-PRAM supports powerful built-in multiprefix instructions called `mpadd`, `mpmax`, `mpand` and `mpor`, which allow the computation of multiprefix integer addition, maximization, bitwise AND and bitwise OR for up to 4096 processors within two CPU cycles. These multiprefix instructions are available as Fork95 operators, as they are very useful in practice. Note that they are atomic expression operators like `+` or `*`, not functions. These can be used in synchronous as well as in asynchronous mode.

For instance, assume that the statement

```
k = mpadd( ps, expression );
```

is executed simultaneously by a set P of processors (e.g., a group). `ps` must be a (potentially private) pointer to a shared integer variable, and `expression` must be an integer expression. In this way, different processors may address different `sharedvars`, thus allowing for multiple `mpadd` computations working simultaneously. Let $Q_s \subseteq P$ denote the subset of the processors in P whose pointer expressions `ps` evaluate to the same address

[1]Due to a hardware design bug, the hardware processor IDs in the current SB-PRAM prototype are *different* from the processor ranks in the multiprefix operations. Since the latter ones are more interesting to programmers, we have chosen this rank number as processor ID `__PROC_NR__` in our implementation of Fork95. Thus, `__PROC_NR__` does generally *not* reflect the concurrent write priority! Instead, the hardware ID of a processor can be inspected by a call to a library function.

s. Assume the processors $q_{s,i} \in Q_s$ are subsequently indexed in the order of increasing processor ID `__PROC_NR__`[2]. First, each processor $q_{s,i}$ in each Q_s computes `expression` locally, resulting in a private integer value $e_{s,i}$. For each shared memory location s addressed, let v_s denote the contents of s immediately before executing the `mpadd` instruction. Then, the `mpadd` instruction simultaneously computes for each processor $q_{s,i} \in Q_s$ the (private) integer value

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

which is, in the example above, assigned to the private integer variable `k`. Immediately after execution of the `mpadd` instruction, each memory location s addressed contains the global sum

$$v_s + \sum_{j \in Q_s} e_{s,j}$$

of all participating expressions. Thus, `mpadd` can as well be "misused" to compute a global sum by ignoring the value of `k`. Nevertheless, if we are only interested in this side effect, there are better alternatives, namely `syncadd`, `syncmax`, `syncand`, and `syncor`, which take the same parameters as the corresponding multiprefix operators but do not return a value.

For instance, the number p of currently available processors can be found out by synchronously executing

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

This may be extended in order to obtain a consecutive renumbering 0,1,...,$p-1$ stored in a user–defined processor ID `me`:

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

1.3.2 Critical sections

Now, we apply a slight modification to our "hello world" program. We would like each processor to print its processor ID in its `Hello world` message. Since `prS()` can only print (predefined) strings, we need `printf()` for formatted output:

```
#include <fork.h>
#include <io.h>

async int main()
{
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 barrier;
 printf("Hello world from processor %d\n", __PROC_NR__);
 barrier;
}
```

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

But differently from `prS()` which prints a string as a whole to the screen, `printf()` prints the string character by character. The resulting output is

```
PRAM P0 = (p0, v0)> g
Program executed by 4 processors

HHHHeeeellllllllloooo    wwwwoooorrrrllllldddd    ffffrrrrooooommmm    PPPP1230




EXIT: vp=#0, pc=$000001d2
EXIT: vp=#1, pc=$000001d2
EXIT: vp=#2, pc=$000001d2
EXIT: vp=#3, pc=$000001d2
Stop nach 14614 Runden, 547.200 kIps
```

The explanation is that all processors access a shared resource, namely the screen output, concurrently. This mixes the output [3].

To recover, we must force the processors to sequentialize their output. This means that the `printf("Hello world ... )` statement becomes a piece of code that may be executed by at most one processor at any point of time. Such a piece of code is called a *critical section.*

Critical sections generally occur when shared resources such as files, output devices like screen or printer, or shared memory objects like shared variables and shared heap objects are accessed concurrently. If these concurrent accesses are not protected by some arbitrating mechanism, nondeterminism (also called "race conditions") may occur which means that the result of a program may depend on the relative speed of the processors executing it.

A *semaphore* is a shared data structure used to limit access to a critical section to a fixed set of processors. The most common special case is a *lock*, a binary semaphore that allows only one processor being inside a critical section at any point of time. Thus, access to the critical section is sequentialized.

1.3.2.1 Simple locks

A lock can be realized as a shared variable with two states: 0 (`UNLOCKED`) and 1 (`LOCKED`). Initially it is `UNLOCKED`. A critical section S is protected by forcing all processors to wait at the entry of S until the lock is `UNLOCKED`. As soon as a processor manages to enter S, it sets the lock to `LOCKED`, with the goal to prohibit entry for all other processors attempting to enter S. When the processor leaves S, it resets the lock to `UNLOCKED`.

Unfortunately, the following straightforward implementation is not sufficient:

[3] It is also nice to see here that each PRAM operation takes the same time on each processor, and that all processors finish execution at exactly the same point of time. This is due to the fact that `barrier` provides exact synchronization (see [KS95a]).

```
#include <fork.h>
#include <io.h>

#define UNLOCKED 0
#define LOCKED 1

sh int lock = UNLOCKED;

main()
{
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 barrier;
 while (lock != UNLOCKED) ;   /* BUGGY! no atomic access to lock */
 lock = LOCKED;
 printf("Hello world from P%d\n", __PROC_NR__); /*critical section*/
 lock = UNLOCKED;
 barrier;
}
```

The problem here is that the processors evaluate the condition `(lock != UNLOCKED)` simultaneously; all of them read a zero value and enter the critical section — and we obtain the same output as before. This is not what we wanted.

To avoid this problem with simultaneous or near–simultaneous read and write accesses to shared memory locations, the combination of reading the (old) value (`UNLOCKED`) and writing the new one (`LOCKED`) must be *atomic* for each processor, i.e. at simultaneous accesses by several processors, only one of them reads the old value while the other ones read the new one (or, if different values are written, each processor also reads a different value). This is a classical problem of parallel processing. Atomic operations [GLR83] such as *test&set*, *fetch&add*, or *compare&swap*, are supported by hardware on most systems[4]. In Fork95, the `mpadd()` operator can be used immediately for atomic *fetch&add*, and the `mpmax` instruction for atomic *test&set*. Here we use `mpmax()` for locking:

```
#include <fork.h>
#include <io.h>

#define UNLOCKED 0
#define LOCKED 1  /*LOCKED > UNLOCKED is sufficient when using mpmax*/

sh int lock = UNLOCKED;

main()
{
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 barrier;
 while (mpmax(&lock,LOCKED) != UNLOCKED) ;   /*atomic access to lock*/
 printf("Hello world from P%d\n", __PROC_NR__); /*critical section*/
```

[4] Software realizations are possible but generally less efficient in practice (see [ZYC96] for a survey of software locking mechanisms).

```
 lock = UNLOCKED;
 barrier;
}
```

And now we get the desired output:

```
PRAM P0 = (p0, v0)> g
Program executed by 4 processors

Hello world from P0
Hello world from P1
Hello world from P2
Hello world from P3
EXIT: vp=#0, pc=$00000214
EXIT: vp=#1, pc=$00000214
EXIT: vp=#2, pc=$00000214
EXIT: vp=#3, pc=$00000214
Stop nach 24964 Runden, 713.257 kIps
0214 18137FFF  POPNG   R6, ffffffff, R1
PRAM P0 = (p0, v0)>
```

Note that the number of cycles required by the program has now considerably increased as the execution of the critical section is sequentialized. The overhead for locking can be slightly improved by using the assembler–coded routine `simple_lockup()`. Using the predefined data type `simple_lock` and the corresponding set of routines defined in `fork.h`, our example program reads

```
#include <fork.h>
#include <io.h>

#define UNLOCKED 0
#define LOCKED 1

sh simple_lock lock;

main()
{
 simple_lock_init( lock );   /* to be executed before first locking */
 if (__PROC_NR__==0)
   printf("Program executed by %d processors\n\n", __STARTED_PROCS__);
 barrier;
 simple_lockup( &lock );
 printf("Hello world from P%d\n", __PROC_NR__);
 simple_unlock( &lock );
 barrier;
}
```

1.3.2.2 Deadlock

The critical section is bracketed by the `lockup`[5] and the corresponding `unlock` call. It is important to maintain this structure always during execution. If e.g. the `unlock` operation is skipped by some processor (e.g. because its program control jumps across it, or the programmer has just forgotten to insert the `unlock`, or he `unlock`s the wrong lock variable), the subsequent processors would wait forever in their `lockup` call. Such a situation is called a *deadlock*. If the `lockup` operation is omitted, several processors may be inside the critical section simultaneously, which again may lead to inconsistency of data (race conditions) and other undesired effects.

Another common source of deadlock is unsuitable nesting of critical sections. In principle, critical sections may be nested, e.g. if a processor needs mutually exclusive access to two or more shared resources. However, problems arise if such a nested critical section is not contiguous in the program, or, conversely spoken, if several pieces of the program are protected by the same lock variable. For instance, the following program[6] causes the processors to mutually block each other:

```
#include <fork.h>
#include <io.h>

sh simple_lock l1, l2;

main() {            /* !deadlock! */
  simple_lock_init( l1 );   simple_lock_init( l2 );
  if (__PROC_NR__ % 2) {
     simple_lockup( &l1 );
     prS("section 1, my processor ID is odd\n");
     simple_lockup( &l2 );
     prS("section 2, my processor ID is odd\n");
     simple_unlock( &l2 );
     simple_unlock( &l1 );
  } else {
     simple_lockup( &l2 );
     prS("section 2, my processor ID is even\n");
     simple_lockup( &l1 );
     prS("section 1, my processor ID is even\n");
     simple_unlock( &l1 );
     simple_unlock( &l2 );
  }
  barrier;
}
```

Started with at least two processors, these block each other:

```
PRAM P0 = (p0, v0)> g
section 1, my processor ID is odd
section 2, my processor ID is even
```

[5] In the literature the `lockup` operation is often referred to as *P–operation*, the `unlock` operation as *V–operation*.

[6] We use the `prS` routine here in order to avoid intermixed output on the screen.

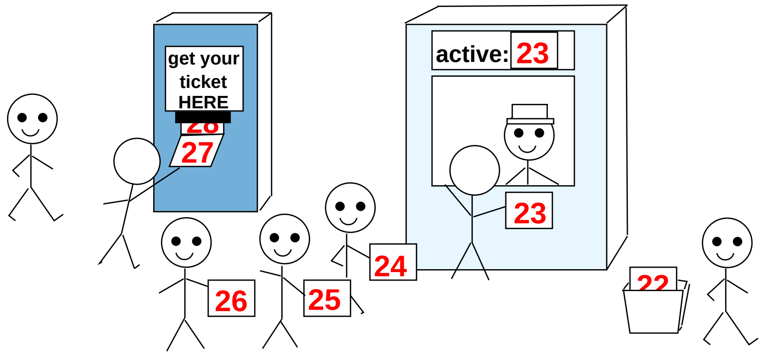


Figure 1.2 The analogy of a booking-office management for the fair lock implementation.

and we have to abort the simulation by `Ctrl-C` to escape from this infinite loop.

Generally, a deadlock can be caused by a set of processors whose requests for shared resources form a cyclic chain such that each processor waits for a resource which is currently held by its predecessor. The deadlock may be resolved if this cyclic chain can be broken, e.g. if one of the processors releases its own resource, or if a processor can alter its order of accessing the resources.

1.3.2.3 Fair locks

A simple lock sequentializes the accesses to a critical section. However it is not guaranteed that processors will get access in the original order of their arrival at the `lockup` operation. A processor may even be forced to wait quite a long time while other processors coming later are served earlier. Thus, the `simple_lock` is not *fair*. This may become critical if there exists an upper limit for the completion time of the processor's computation, e.g. in real–time applications. Exceeding this limit is called *starvation*.

A *fair lock* sequentializes accesses while maintaining the order of arrival times of the processors at the `lockup` operation. One implementation works similar to an automatic booking-office management system (see Figure 1.2): Each customer who want to access the booking-office must first get a waiting ticket from a (shared) ticket automaton. After having served a customer, the booking-clerk increments a globally visible counter indicating the currently active ticket index. Only the customer whose waiting ticket index equals this counter value will be served next. The others have to wait until their ticket index becomes active.

In Fork95, we model this booking-office by two shared counters, one for the ticket automaton, and one for the ready-to-serve counter.

```
typedef struct {
  unsigned int ticket;   /* the ticket automaton */
  unsigned int active;   /* the currently active ticket */
} fair_lock;

sh fair_lock fl;
```

Both counters have to be accessed by atomic *fetch&increment* operations [7]. They are initialized to zero:

```
void fair_lock_init( fair_lock *pfl )
{
  pfl->ticket = pfl->active = 0;
}
```

The `lockup` operation gets a ticket `t` and waits until `t` becomes active:

```
void fair_lockup( fair_lock *pfl )
{
  pr int t = mpadd( &(pfl->ticket), 1 );      /* get a ticket t */
  while (t != pfl->active)  ;    /* wait until t becomes active */
}
```

The `unlock` operation increments the `active` counter:

```
void fair_unlock( fair_lock *pfl )
{
  mpadd( &(pfl->active), 1 );  /* atomically increment counter */
}
```

The Fork95 library contains assembler implementations of these routines. The space required to store a fair lock is two words, twice the space of a simple lock. The run time required by the `fair_lockup` and the `fair_unlock` operation is practically the same as the time taken by `simple_lockup` resp. `simple_unlock`.

1.3.2.4 Other types of locks

Several other types of locks have been described in the literature [Wil88, Röh96].

For instance, the Reader–Writer lock handles the case that there are two types of processors: the *readers* which want to read the shared resource, and the *writers* which want to write to it. On a practical PRAM architecture, such as the SB-PRAM, concurrent read should be always possible. Thus, mutual exclusion is naturally only required to sequentialize different write accesses with respect to each other, and different write and read accesses. In contrast, different read accesses need not be sequentialized with respect to each other. The current status of a reader–writer lock can thus be described as a pair $(w, r) \in \{0, 1\}^2$ where $w = 0$ means that no writer is inside the critical section, $w = 1$ means that a writer is working inside. $r = 0$ means that the critical section is free of readers, and $r = 1$ means that some reader is working inside. Readers may only enter if $w = 0$ while writers may only enter if $w = 0 \wedge r = 0$.

An implementation described in [Wil88, Röh96] holds a counter indicating the number of readers currently working inside the critical section, and a fair lock for the writers.

[7] Additionally, one should here take care about handling overflow if the size of an unsigned integer is exceeded. On the SB-PRAM, this is not necessary, since `mpadd` computes correctly modulo 2^{32} [Röh96]. Only if the total number of processes exceeded 2^{32}, explicit range checking would be required, but such a large PRAM machine is very unlikely to exist in the foreseeable future.

1.3.3 Semaphores

1.3.3.1 User–defined barrier synchronization

A barrier synchronization point, as already introduced in Section 1.1.2, is a point of the program where processors must wait until the last processor has reached it. The `barrier` statement provides an exact barrier synchronization which additionally requires that all processors continue with program execution at exactly the same clock cycle. An inexact barrier does not require this additional condition.

Such an inexact barrier can be realized by an integer semaphore. The initializiation is again to be executed before any processor uses the semaphore:

```
sh unsigned int my_barrier = 0;
```

A processor reaching the barrier synchronization point atomically increments the semaphore and waits then until the semaphore's value reaches `__STARTED_PROCS__`:

```
syncadd( &my_barrier, 1 );                    /*atomically fetch&inc*/
while (my_barrier != __STARTED_PROCS__)  ;   /*wait*/
```

From this simple implementation it also becomes obvious that a processor jumping across this piece of code will cause a deadlock since the other processors remain forever waiting in the `while` loop.

If the semaphore `my_barrier` should be re–used for subsequent barrier synchronizations, then, after continuing, `my_barrier` must be reset to zero. However, this requires waiting for a implementation–dependent (short) time after exiting the while loop, because otherwise a processor leaving the while loop[8] early may reset the semaphore before a "late" processor has checked the exit condition and thus would read the new semaphore value zero — and wait forever. Also the other processors would be waiting forever at the next synchronization point; this is a deadlock.

To avoid this, we implement a *re-initializing barrier* [Röh96] where a processor determines whether it is the last one to arrive at the barrier. If yes, then it is responsible for resetting the semaphore:

```
if ( mpadd( &my_barrier, 1) != __STARTED_PROCS__ - 1 )
   while (my_barrier != __STARTED_PROCS__)  ;     /*wait*/
else {
   /* executed by the processor arriving last: */
   while (my_barrier != __STARTED_PROCS__)  ;     /*wait*/
   my_barrier = 0;
}
```

The `while` loop in the `else` branch is semantically redundant; it is only required to guarantee that the processor arriving last waits for at least one iteration, in order to give the other processors enough time to exit the loop before the semaphore is reset to zero[9].

[8] Even an empty while loop takes a few clock cycles per iteration for checking the exit condition. As long as we do not program explicitly in assembler, we cannot rely on specific execution times.

[9] For the execution of Fork95 programs the operating system guarantees that processors are not interrupted except for "lethal" reasons. A barrier mechanism handling the case that processors may be interrupted is described in [Röh96].

1.3.3.2 Self–Scheduling for independent loop iterations

In asynchronous mode, the `mpadd()` operator is very useful to program self–scheduling parallel loops. Self–scheduling (see e.g. [Pol88] for a survey) is a technique that dynamically assigns loop iterations to processors if the iterations are independent but take different execution times to complete, which are typically not known in advance. A shared loop iteration counter

```
sh int iteration = 0;
```

initialized to zero serves as a semaphore to indicate the next loop iteration to be processed. A processor that becomes idle applies a `mpadd()` to fetch its next iteration:

```
pr int i;                              /* dynamic scheduling: */
...
for ( i=mpadd(&iteration,1); i<N; i=mpadd(&iteration,1) )
    execute_iteration ( i );
```

Note that on the SB-PRAM, parallel accesses to the `loopcounter` by `mpadd()` are not sequentialized, and the additional overhead is marginal.

If the execution times of different iterations may vary considerably, the more flexible run–time scheduling method is often (but not always) advantageous compared to static scheduling, e.g.

```
pr int i;                               /* static scheduling: */
...
for ( i=__PROC_NR__; i<N; i+=__STARTED_PROCS__ )
    execute_iteration ( i );
```

Here, the loop iterations are cyclically distributed across the available p processors. If a processor k is unlucky, the sum of the execution times of the iterations k, $k+p$, $k+2p$,... assigned to processor k is considerably higher than the average load per processor.

The theory of self–scheduling [Pol88] also considers combining several neighbored iterations to chunks (chunk scheduling), maybe in decreasing size (guided self–scheduling) and to assign them as a whole to a processor, as in many parallel systems the overhead of accessing the shared iteration counter is not negligible. On a PRAM with unit–time multiprefix instructions such as the SB-PRAM, however, the accesses by `mpadd` to the loop iteration semaphore are not sequentialized, and the access time is not larger than accessing a local loop counter.

If the execution times of the iterations are explicitly known to the programmer, (s)he may reorder the iterations such that the load per processor is close to the average. However, this is an NP–complete optimization problem.

1.3.3.3 Self–Scheduling for dependent loop iterations

Data dependencies often imply a partial order among the loop iterations. Parallel execution is legal only as long as the data dependences are not violated. This can be guaranteed using semaphores.

For instance, a computation which is specified in sequential by the loop

```
for (i=0; i<N; i++)
   A[i] = B[i] * A[i-D[i]];
```

with $0 \leq$`D[`i`]`$\leq i$ for all i, can be parallelized only if it is guaranteed for all $i \in \{0, ..., N-1\}$ that the point of time $T(i)$ when iteration i is executed comes always later than $T(i-D[i])$, as there is a data dependency from iteration `i-D[i]` (where an element of array `A` is written) to iteration `i` (where the same element is used). Static scheduling is not possible here because the dependence distance `D[i]` is only known at run time.

A parallelized version of this loop again uses a semaphore for the loop iteration counter. Additionally, a flag array

```
sh int DONE[N];
```

is required. Before executing the loop, the counter and the flags are initialized to zeroes. Then, execution of each iteration `i` is delayed until the value it attempts to read has been written by the processor executing iteration `i-D[i]`:

```
for (i=__PROC_NR__; i<N; i+=__STARTED_PROCS__)  DONE[i] = 0;
iteration = 0;
barrier;
for ( i=mpadd(&iteration,1); i<N; i=mpadd(&iteration,1) ) {
   if (D[i] > 0)
     while (!DONE[i-D[i]]) ;  /*wait for iter. i-D[i] to complete*/
   A[i] = B[i] * A[i-D[i]];
   DONE[i] = 1;
}
barrier;
```

Note that a separate prepass for the inspection of the data dependency structure, as in the inspector–executor method [KMR90, SCMB90], is not necessary here.

1.3.4 Parallel Queues

A *queue* is a data structure containing linearly ordered items with two basic operations: (1) a *put* operation that appends an item at the end of the queue, (2) a *get* operation that removes and returns the first element of the queue, if it is nonempty, and returns `NULL` otherwise. The elements obtained by subsequent calls to *get* are returned in exactly the same order as they have been *put* into the queue before (FIFO = first in, first out).

A straightforward (sequential) implementation of a queue is a doubly linked list, as shown in Figure 1.3(a). This offers, in contrast to an array, the advantage of unlimited size. In a parallel environment, however, this data structure has a severe disadvantage because it is inherently sequential; accessing the queue is a critical section. A solution to this problem is to split the linked list into $k > 1$ sublists which are accessible by an array of k list heads. The array elements are distributed across the k sublists in a round–robbin manner, see Figure 1.3(b). In order to maintain the total ordering of the elements, the next sublist to *get* from and the next sublist to *put* to have to be marked by a pointer. Inside each sublist, the *put* position is at the end and the *get* position at the first element again, as in the sequential case.

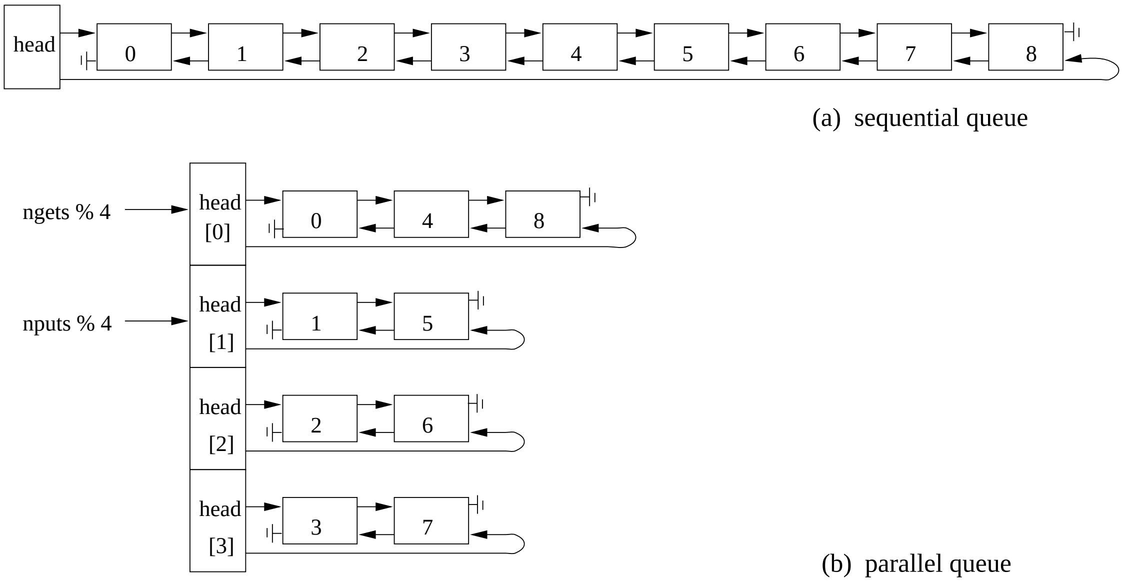


Figure 1.3 A sequential queue (a) and a parallel queue with 4 heads (b). In a parallel queue with k heads the elements of the corresponding linear sequential list are distributed among the k sublists in a round–robbin manner. In the sequential queue, the next element to get is the first one and the next element to put is the last one, reachable via pointers from the head. The parallel queue holds two pointers to denote the sublists where to *get* resp. *put* the next element. Inside a sublist, the *get* and *put* positions are at the head and the end, as in the sequential case.

A queue element consists of references to its successor and predecessor element and contains some user data:

```
typedef struct el {
   struct el *next, *prev;
   int data;          /*or whatever should be stored in the queue*/
} element;
```

The head of a sublist holds references to the first and the last list element. In particular if k is small, it may happen that several processors try to access the same sublist at the same time. For this reason, these accesses to the same sublist must be sequentialized by a lock. Furthermore, the head holds local counters for *put* (`nputs`) and *get* (`ngets`) accesses.

```
typedef struct {
   element *first, *last;
   fair_lock lock;
   int nputs, ngets;
} qhead;
```

The parallel queue data structure contains an array of k sublist heads, global counters to indicate the next *get* resp. *put* position, and the total number `size` of elements stored in the queue.

```
#define k 4          /*number of sublists*/

typedef struct {
   qhead head[k];   /*dynamic array of sublist heads*/
   int ngets;
   int nputs;
   int size;
} pqueue;

sh pqueue *pq;
```

Initialization of a parallel queue q with k heads is done by calling `init_pqueue( &pq )`. Since pq is shared, the value of ppq in the callee is equal for all processors.

```
void init_pqueue( pqueue **ppq )
{
   pr int i;
   if (__PROC_NR__==0) /*mask out all but one processor*/
      *ppq = (pqueue *) shmalloc( sizeof( pqueue ));
   barrier;
   for (i=__PROC_NR__; i<k; i+=__STARTED_PROCS__) {
      (*ppq)->head[i].first = (*ppq)->head[i].last = NULL;
      (*ppq)->head[i].nputs = (*ppq)->head[i].ngets = i;
      fair_lock_init( &((*ppq)->head[i].lock) );
   }
   (*ppq)->ngets = (*ppq)->nputs = (*ppq)->size = 0;
   barrier;
}
```

The routine `shmalloc()` allocates shared heap memory; it will be discussed in detail in Section 1.6.2. The first `barrier` is required to ensure that the space for the `pqueue` data structure has been allocated before any processor writes to it. Concurrent writes to `(*ppq)->ngets` etc. are not harmful since all processors write the same value before the second `barrier` is reached. After the second `barrier`, the initialization of `pq` is completed.

Up to k processors may access the parallel queue at the same time if each of them is working on a different sublist. The optimal value of k is a question of fine–tuning. Clearly, a large k in the same order of magnitude as the number of all available processors will avoid most serialization. A smaller value of k will save some shared memory space and initialization time, at the expense of partial serialization if there arise massive bursts of queries for accessing the parallel queue. [Röh96] proposes $k = 32$ for the SB-PRAM prototype with $p = 4096$ processors.

The *put* operation may be implemented as follows:

```
void put( pqueue *pq, element *e )
{
 pr int myputidx;
 pr qhead *mylist;
 syncadd( &(pq->size), 1);
```

```
 myputidx = mpadd( &(pq->nputs), 1);
 mylist = &(pq->head[myputidx % k]);
 e->next = NULL;
 while (mylist->nputs != myputidx)  ;  /*wait until it's my turn*/
 fair_lockup( &(mylist->lock) );
   e->prev = mylist->last;
   if (mylist->last) mylist->last->next = e;
   else              mylist->first = e;
   mylist->last = e;
 fair_unlock( &(mylist->lock) );
 syncadd( &(mylist->nputs), k );
}
```

The put accesses are consecutively numbered as myputidx, starting with 0. A put access i is routed to sublist $i\%k$. The while loop prohibits that another processor executes a subsequent put operation (e.g. $i+k$) before put operation i is executed. This already sequentializes all put accesses to the same sublist; the locking is only required to avoid conflicts with simultaneously occurring get operations on the same sublists.

```
element *get ( pqueue *pq )
{
 pr int mygetidx;
 pr qhead *mylist;
 pr element *e;
 if (tdr(&(pq->size),1,0)) { /*test-decrement-retest; enter if nonempty*/
   mygetidx = mpadd( &(pq->ngets), 1);
   mylist = &(pq->head[mygetidx % k]);
   while (mylist->ngets != mygetidx) ;  /*wait until it's my turn*/
   while (mylist->nputs == mygetidx) ;  /*wait until put is ready*/
   fair_lockup( &(mylist->lock) );
     e = mylist->first;
     mylist->first = e->next;
     if (e->next) e->next->prev = NULL;
     else         mylist->last = NULL;
   fair_unlock( &(mylist->lock) );
   syncadd( &(mylist->ngets), k );
   return e;
 }
 else return NULL;
}
```

The get accesses to the same sublist are kept in the correct order by the same mechanism as used in put(). Additionally, the processor executing get on sublist l must wait until another processor potentially executing the corresponding put operation has finished its transaction in sublist l.

The only remaining problem is how to determine whether the parallel queue is nonempty, and, if yes, immediately decrementing the queue size in order to prevent other get operations from execution on a possibly empty queue. A naive approach

```
if (pq->size >= 1) {
  mpadd( &(pq->size), -1 )
  ... /* get - operation */
}
```

would not be sufficient: Assume that `pq->size` has a value of 1, and $p > 1$ processors execute the comparison `(pq->size >= 1)` at exactly the same cycle. Then `pq->size` will get a negative value, as the following `mpadd` operation is executed by more than one processor. Thus, this composed operation must be atomic. An implementation with locks would be straightforward but leads to sequentialization of accesses to `pq->size`.

A serialization–free mechanism for decrementing a shared variable with testing for a lower bound for its value is provided by the following routine `tdr` (*test–decrement–retest*). `tdr(pc,d,b)` returns 1 (true), if the shared integer variable `*pc` can be decremented by a difference `d` without violating the condition `*pc`$\geq$`b`, and 0 (false) otherwise. If 1 is returned, `tdr` actually decrements `pc` atomically by `d`, otherwise not. `tdr` may be executed by arbitrarily many processors in parallel.

```
int tdr ( int *pc, int d, int b )
{
 if (*pc - d >= b)              /*test*/
    if (mpadd(pc,-d) - d >= b)  /*decrement, retest*/
       return 1;
    else {
       mpadd( pc, d );          /*failed: restore *pc*/
       return 0;
    }
 else return 0;                 /*failed*/
}
```

The decrement–retest operation in the second line of `tdr()` is speculative: a processor executing it assumes that no other processor has tested the condition in the first line recently. If the processor finds out by the retest operation that the bound has been violated (and thus the assumption was false), it increments again to restore the correct value, and returns 0; otherwise it accepts [10].

The first test operation (in the first line of `tdr()`) is redundant in our case. But it becomes necessary e.g. when using `tdr` as a locking mechanism with an integer semaphore variable, such as

```
while (!tdr( &semaphore, 1, 1 )) ;    /*lockup operation*/
critical_section();
mpadd( &semaphore, 1 );               /*unlock operation*/
```

in order to avoid race conditions among processors spinning on the `tdr` call[11]. This is discussed in detail in [GLR83].

[10] The test in the first line of routine `tdr` relies on the fact that the `ldg` load operation in `*pc` and the `mpadd` operations in the second and fifth line are executed at the same value of the `modulo` flag and at a different value than the `syncadd` access to the queue size in `put()`. This is guaranteed by the compiler: `ldg` and `mpadd` are both scheduled only when `modulo` $= 1$ while `stg` and `syncadd` are scheduled only when `modulo`$= 0$.

[11] This problem mainly occurs if the operating system can interrupt (and thus delay) the processor during program execution, which is not possible at the SB-PRAM (except for fatal reasons).

In the same way also the routine `tir(pc,d,b)` (*test–increment–retest*) can be implemented.

In order to make `tdr`(pc, d, b) resp. `tir`(pc, d, b) work properly for simultaneous execution by p processors, the programmer must guarantee that no range overflows may occur. If *MININT* denotes the minimal signed integer number and *MAXINT* the maximal signed integer number which can be represented on the given machine, then for `tdr`

$$b \geq MININT + (p-1)d_{max}$$

respectively for `tir`

$$b \leq MAXINT - (p-1)d_{max}$$

impose an upper bound d_{max} on d. On the SB-PRAM, integers range from $MININT = -2^{31}$ to $MAXINT = 2^{31} - 1$ and at most $p = 4096$ processors may execute `tdr` resp. `tir` at the same time; thus d should not be larger than 2^{19}.

Similar implementations of a parallel queue are given e.g. in [GLR83], [Wil88] and [Röh96]. [Wil88] and [Röh96] also give an implementation of a general `remove` operation that allows to remove elements from *any* position in the queue, at the expense of an additional counter in each list element and slightly more complicated `put` and `get` implementations.

[Röh96] also provides a parallel broadcast queue with a modified `get` operation where an element must be read by every processor before being removed from the queue. Furthermore, [Röh96] implements a parallel stack on the SB-PRAM.

1.3.5 Exercises

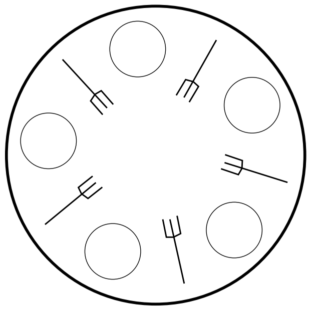
(1) *desynchronization* Before entering the `main()` function of a Fork95 program, all available processors execute the startup code of the program, and then perform an exact barrier synchronization in order to execute initializers for shared variables in exactly synchronous mode. For this reason, all processors happen to enter the (asynchronous) `main()` routine at exactly the same cycle, although the programmer should not rely on this. In order to test your exercise programs, write a short routine `desynchronize()` which, executed at the beginning of `main()`, will cause different delays for different processors.

(2) Write a Fork95 routine `pprintf()` which processes and prints its arguments as `printf()` but writes the resulting string atomically to the screen, similar to `prS()`. Use a fair lock. Furthermore, the print-out should be preceded by the processor ID as follows:

```
#0001# Hello world from P1
```

An implementation of `pprintf()` is supplied with the Fork95 library.

(3) A classical example for the problems of deadlock and starvation is the *Dining Philosophers* analogon. P philosophers spend their life eating and thinking. They sit on P chairs around a round table with P plates. Between any two neighbored plates, one fork is located (see figure). Clearly, a fork can only be used by one philosopher at the same time.



In order to eat, a philosopher needs to take up the fork to the left *and* the fork to the right of his plate. After eating, the philosopher again puts down the forks and thinks for some time θ before getting hungry again. If at least one of the two forks is currently occupied, the philosopher must wait; he may retry after δ time units. Each philosopher should eat T times.

Write a Fork95 program that models this scenario. Experiment with the parameters P, T, θ, δ. How can you avoid deadlocks?

Assume that after repeatedly failing t times to access the forks, a philosopher must succeed in order to survive. Extend your program to detect starvation, and try out for different values of t. How can starvation be avoided?

Can you figure out an elegant way of accessing *both* forks at once by a suitable *fetch&op* instruction for the special case $P \leq 32$? (*Hint:* model each fork as one bit in a shared bitvector (integer) and use bitwise multiprefix operators with suitable bit masks to access them atomically.)

(4) Show that access to a shared sequential queue in a parallel system is a critical section.

(5a) Write the routine `int tir( int *pc, int d, int b)` in Fork95.

(5b) Try to imagine a situation where the first test in `tdr()` resp. `tir()` is necessary to avoid an infinite loop.

(6) Write routines `init_pstack( pstack **)`, `void push( pstack *, element *)` and `element *pop( pstack * )` for a parallel stack.

(7) Extend the parallel queue implementation such that by `void remove( element e )` an element `e` can be removed at any position of the queue. (Hint: In each list element keep a counter `nrem` of already removed elements at the right of each element, and in each sublist head keep a counter `nremfirst` for elements removed to the left of the `first` element in the sublist. Which modifications are required for `put()` and `get()`?)

1.4 Synchronous and asynchronous regions

Fork95 offers two different programming modes that are statically associated with source code regions: *synchronous mode* and *asynchronous mode* (see Figure 1.4).

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

In asynchronous mode, which we know already from the previous sections, the synchronicity invariant is not enforced. The group structure is read–only; shared variables

```
sync int *sort( sh int *a, sh int n )
{
  pr int i;
  extern sync int compute_rank( sh int *, sh int);
  if ( n>0 ) {
     pr int myrank = compute_rank( a, n );
     for (i=0; i<n; i+=p)
        a[myrank] = a[i];
     return a;
  }
  else
     farm {
        printf("Error: n=%d\n", n);
        return NULL;
     }
}

extern async int *read_array( int * );
sh int *a, n;

async main()
{
  a = read_array( &n );
  start {
     a = sort( a, n );
     seq  if (a) printf("a[%d]=%d\n", $, a[$]);
  }
}
```

Figure 1.4 A Fork95 program is statically partitioned into synchronous (dark dashed boxes) and asynchronous (light solid boxes) regions.

and automatic shared heap objects cannot be allocated. There are no implicit synchronization points. Synchronization of the current group can, though, be explicitly enforced using the `barrier` statement.

The importance of being synchronous. The synchronous mode overcomes the need for protecting shared variables by locks because they are accessed in a deterministic way: The programmer can rely on a fixed execution time for each operation which is the same for all processors at any time during program execution. Thus no special care has to be taken to avoid race conditions. The synchronous mode saves the time and space overhead for locking and unlocking but incurs some overhead to maintain the synchronicity invariant during synchronous program execution.

As an example, consider the following synchronous code:

```
sh int a, x;
pr int y;
...
  if (__PROC_NR__ % 2)  a = x;
  else                  y = a;
...
```

Half of the processors execute the *then* branch of the `if` statement, the other half executes the *else* branch. While it is not determined whether `y` will finally have the old or the new value of `a`, it is nevertheless guaranteed by the synchronous mode that all processors writing their private variable `y` in the *else* branch assign the *same* value of `a`. A lock to guarantee this is not required.

The importance of being asynchronous. In asynchronous program regions there are no implicit synchronization points. Maintaining the synchronicity invariant requires a certain overhead (exact barrier synchronization) also for the cases where each group consists of only one processor, or when exact synchronicity is not required for consistency because of the absence of data dependencies. Hence marking such regions as asynchronous can lead to substantial savings. During the implementation of the (mostly synchronous) routines of the PAD library described later, we found considerate usage of the `farm` statement and asynchronous functions to pay off in significant performance improvements.

1.4.1 Declaring synchronicity of a function

Functions and pointers to functions are classified as either synchronous (declared with type qualifier `sync`) or asynchronous (`async`, this is the default), e.g.

```
sync void foo (void) { ... }
extern sync int sort ( sh int, sh int [], sh sync int (*cmp)(int,int) );
pr async void (*pf)(void);
```

The order of qualifiers for sharity and synchronicity in a declarator is irrelevant. Note that the sharity of a pointer to a function relates only to the pointer, not to the function being called. A synchronicity qualifier is only applicable to declarations and definitions of functions or pointers to functions; a sharity qualifier is not applicable to the declaration or definition of a function.

`main()` is asynchronous by default. Initially, all processors on which the program has been started by the user execute the startup code in parallel. After that, these processors start execution of the program in asynchronous mode by calling function `main()`.

1.4.2 The `farm` statement

A synchronous function is executed in synchronous mode, except from blocks starting with a `farm` statement[12]

```
farm  <stmt>
```

which enters asynchronous mode during execution of the asynchronous region `<stmt>`, and re-installs synchronous mode after all processors have finished their execution of `<stmt>` by a group–local exact barrier synchronization.

`farm` statements make only sense in synchronous regions. The compiler emits a warning if it encounters a `farm` statement in an asynchronous region.

[12] The keyword `farm` is due to historical reasons [KS95b]; it was originally motivated by the intended use for programming independent tasks (*task farming*). From today's point of view, a better choice would be something like `relax` or `sloppy` (*relaxed synchronicity*). Anyway, what's in a name? Finally, if you really like another keyword replacing `farm`, you may just use the C preprocessor...

1.4.3 The seq statement

The `seq` statement

```
seq  <stmt>
```

works similar to the `farm` statement. Its body `<stmt>` is an asynchronous region and is thus executed in asynchronous mode, but, in contrast to `farm`, by only *one* processor, which is arbitrarily chosen from the current group of processors executing the `seq` statement.

At the end of the `seq`, synchronous mode of execution is re-installed by a group–local exact barrier synchronization.

The `seq` statement is semantically equivalent to

```
sh int p = 0;                 /*shared by the current group*/
pr int me = mpadd( &p, 1 );   /*renumber the processors*/
farm if (me==0) <stmt>
```

but in contrast to this variant which just masks out the remaining $p-1$ processors of the group, `seq` allows a clever compiler back-end to keep, if possible, all p processors busy by exploiting instruction–level parallelism in the execution of `<stmt>`.

`seq` statements make only sense in synchronous regions. The compiler emits a warning if it encounters a `seq` statement in an asynchronous region.

1.4.4 The start statement

Asynchronous functions are executed in asynchronous mode, except from blocks starting with the `start` statement[13]

```
start <stmt>
```

The `start` statement, only permitted in asynchronous mode, switches to synchronous mode for its body `<stmt>`. It causes all available processors to perform an exact barrier synchronization and execute `<stmt>` simultaneously and in synchronous mode, with unique processor IDs `$` numbered subsequently from 0 to `__STARTED_PROCS__`-1. `<stmt>` is a synchronous region.

`start` statements make only sense at the top level of the program; there should not be more than one `start` active at the same time.

[13] Again, the keyword `start` is not the best choice from today's point of view; something like `strict` may be more suitable. In the old FORK proposal [HSS92], the `start` statement was used with a shared parameter expression, `start(`e`)`, to start a sub–PRAM of e processors while freezing the current configuration. However this did not really add more expressiveness to the language design. During the redesign phase [KS95b] we dropped the argument of `start`.

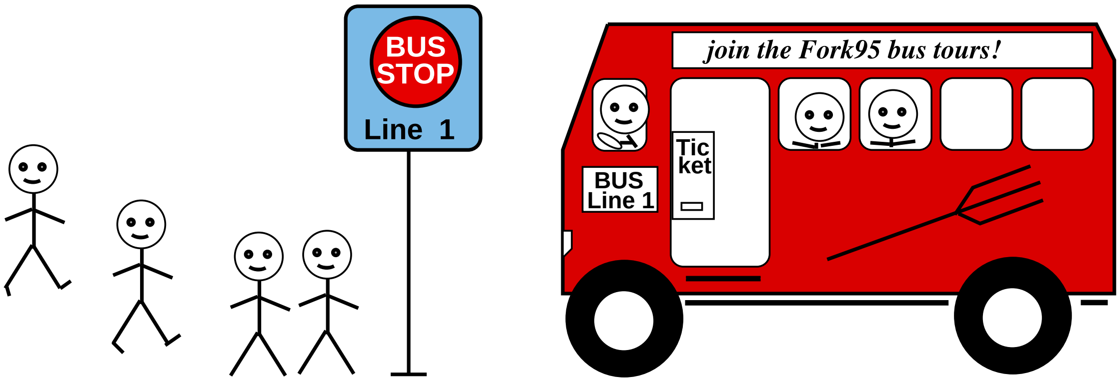


Figure 1.5 The bus stop analogy for the semantics of the `join` statement

1.4.5 Calling synchronous and asynchronous functions

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

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

1.4.6 The `join` statement

The `join` statement recently introduced into Fork95 [KS97b] is a generalization of the `start` statement which allows more flexibly changing from asynchronous to synchronous mode and permits arbitrary nesting of synchronous and asynchronous regions

A useful analogy to understand the behaviour of the new `join` operator is a *bus stop* (see Figure 1.5). Imagine a city with several excursion *bus lines*, each of which has one unique bus stop. One excursion bus circulates on each bus line. At the door of each bus there is a *ticket* automaton that sells tickets when the bus is waiting. Tickets are numbered consecutively from 0 upwards. All passengers inside a bus form a group and behave synchronously. They can be distinguished by an ID number `$` which is initialized to their ticket number. Each bus has a *bus driver*, namely the passenger that obtained ticket number zero.

What happens at the bus stop? Passengers come by asynchronously and look for the bus to join the excursion. If the bus is *gone*, they have the choice to either *retry* and wait for the next bus tour of this line (if there is one), perhaps by doing some *other useful work* meanwhile, or to resign and continue with the next statement. If the bus is not gone, it is waiting and its door is not locked, thus the passenger can get a ticket at the ticket automaton and enter. The first passenger entering the bus receives ticket number 0; he becomes the bus driver and does some *initialization* work at the bus. He waits according to a certain *delay* strategy. Then he signals that he will start the bus and switches off the ticket automaton — thus no one can enter this bus any more. At this point, some passengers inside are still allowed to immediately *spring off* the starting bus if they desire. After that, the door is definitely locked. The passengers inside form a *group* and behave synchronously during the *bus tour*. They can allocate shared objects that are then accessible to all bus passengers during the tour. After the tour, all passengers leave the bus at the bus stop and continue, again asynchronously, with their next work.

The behaviour described in the bus analogy is supplied in Fork95 by a language construct

```
join ( delaystmt; springoffcond; SMsize )
       statement
  else useful_work
```

The passengers are the processors. Each `join` instruction installs a unique bus line with a bus stop. The delay statement `delaystmt` specifies a statement that is executed by the bus driver and models a time interval or a condition that must be met to start the bus. The spring–off condition `springoffcond` is a boolean expression supplied by the programmer; it may be different for different processors[14]. The delay statement as well as the spring–off condition may read the current value of the ticket counter, `__TICKET__`. The bus driver allocates a chunk of `SMsize` words of shared memory [15] in order to install a new shared stack and heap for the bus. This chunk of memory is released when the bus tour terminates. The bus tour itself corresponds to the proper body `statement` of the join instruction and is a synchronous region.

The `else` part is optional and specifies an asynchronous statement `useful_work` that is executed by processors that miss the bus and by those that spring off. A `retry` statement occurring inside `useful_work` causes these processors to go back to the bus stop and try again to get the bus, very similar to `continue` in loops.

Note that a bus cannot have more than one entry point (`join` instruction) within the program. If this is desired by the programmer, (s)he can encapsulate the `join` into a function and call that function from several sites.

The implementation is described in detail in [KS97b]; for now we only sketch the most important ideas. Internally, there is for each bus a a shared variable `gone` that guards access to the ticket automaton, which, in turn, is an integer semaphore `__TICKET__` that is accessed by a multiprefix increment operation.

Buses of different bus lines can be nested. Recursion (directly or indirectly) to the *same* bus line will generally introduce a deadlock because a processor would wait forever for a bus whose return it is blocking itself just by its waiting.

Note that the passengers inside a bus will generally have their origin in different former leaf groups. The old group structure, as well as all locally defined shared objects of that old group structure, are *not* visible during the bus tour. Global shared objects are always visible.

The *current implementation* provides a slightly restricted version of the `join` statement, as it is implemented as a macro:

```
join ( joinID, delaystmt, springoffcond, SMsize, is_retry,
       statement, useful_work );
```

[14]If the consecutive numbering of the `$` ID's is destroyed by some processors springing off, the programmer can re–install this by simply recomputing `$` with a multiprefix incrementation at the beginning of the bus tour.

[15]This chunk of memory is allocated on the permanent shared heap (see Section 1.6.2).

where is_retry is an integer value indicating whether the processors missing the bus or springing off should retry (nonzero) or not (zero). Again, statement denotes the synchronous bus tour. Note that the useful_work field may be empty. The joinID (i.e., the bus line ID) must be an integer constant unique to each join macro. This is required by the system to guarantee that, in terms of our analogy, only one bus circulates on each bus line.

1.4.6.1 Example

As an example, consider the semantics of the start statement. start is a special case of join: it collects *all* __STARTED_PROCS__ available processors. Thus, we may rewrite

```
start
   statement;
```

as

```
join( while(__TICKET__ < __STARTED_PROCS__); 0; 100000 )
   statement;
else
   retry;
```

where the bus is equipped with 100000 words of shared memory taken from the permanent shared heap (see Section 1.6.2). Since the delay statement depends on the current value of __TICKET__ and the else clause contains a retry statement, the bus tour statement starts only when all __STARTED_PROCS__ processors have entered the bus.

1.4.6.2 Synchronous parallel critical sections

The join statement allows the implementation of a generalization of the (sequential) critical section introduced in Section 1.3.2, namely *synchronous parallel critical sections*. Using join, blocks of statements can be executed in parallel and synchronously by a set of processors that has been specified by the programmer. The programmer may specify an exact number of passengers or only a maximum or minimum number for these – or some other shared criterion esteemed relevant: any desirable constellation can easily be programmed using delaystmt and springoffcond appropriately. The implementation of the bus guarantees that no other processor can join a bus tour as long as the door is locked. They have to wait (at least) until the processors inside the bus have finished their excursion, i.e. have left the parallel critical section.

1.4.6.3 Synchronous parallel versus asynchronous sequential critical sections

The question arises in which cases synchronous parallel critical sections may be the better choice, compared to traditional sequentializing critical sections. The prerequisite for exploiting the potential benefits of parallel critical sections — parallel speedup and/or

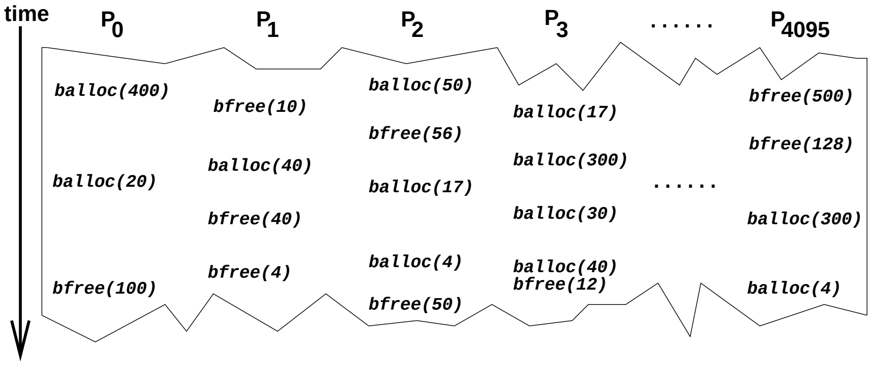


Figure 1.6 Nearly simultaneous queries to the block heap memory allocator may be collected and processed in parallel using the `join` statement.

---

the possibility to apply more sophisticated and expensive algorithms — is the availability of a suitable parallel algorithm for the problem to be solved in the critical section.

As an example let us consider a very simple storage allocation scheme where the available heap memory is divided into N equally sized blocks. Pointers to free blocks are maintained by a queue `avail`. Since N is a known constant, the queue is implemented by a shared array together with two integers `low` and `high`. Integer `low % N` points at the first occupied cell in `avail` whereas integer `high % N` points to the first free cell.

```
sh char *avail[N];  /*array of N pointers*/
sh int high, low;
```

Each processor works on its local task and sometimes issues a query `balloc()` to allocate or `bfree` to release a block of memory (see Figure 1.6; ignore the block sizes for now). To implement operations `void bfree(char *p)` and `char *balloc()` we introduce an auxiliary function `char *pcs(char *ptr, int mode)` with an extra argument `mode` to distinguish between the two usages.

```
/* modes of pcs: */
#define ALLOC 1
#define FREE 0

void bfree(char *p) {
   pcs(p, FREE);
}

char *balloc() {
   return pcs(NULL, ALLOC);
}
```

Now, function `pcs()` is implemented using the `join` construct. Essentially, it consists in applying an `mpadd`–operation to variable `high` (in case `mode == FREE`) resp. to variable `low`. It only has to be taken care of that no block can be delivered from an empty queue.

procs	asynchronous variant			sync. variant using `join`		
1	5390 cc	=	21 msec	6608 cc	=	25 msec
2	5390 cc	=	21 msec	7076 cc	=	27 msec
4	5420 cc	=	21 msec	8764 cc	=	34 msec
8	5666 cc	=	22 msec	9522 cc	=	37 msec
16	5698 cc	=	22 msec	10034 cc	=	39 msec
32	7368 cc	=	28 msec	11538 cc	=	45 msec
64	7712 cc	=	30 msec	11678 cc	=	45 msec
128	11216 cc	=	43 msec	11462 cc	=	44 msec
256	20332 cc	=	79 msec	11432 cc	=	44 msec
512	38406 cc	=	150 msec	11556 cc	=	45 msec
1024	75410 cc	=	294 msec	11636 cc	=	45 msec
2048	149300 cc	=	583 msec	11736 cc	=	45 msec
4096	300500 cc	=	1173 msec	13380 cc	=	52 msec

Figure 1.7 Timings for the parallel shared heap memory allocator, submitted to `balloc()` resp. `bfree` queries. distributed randomly over time. We compare an asynchronous implementation using locks (second column) with a synchronous one using `join` (third column). The measurements are taken on the SB-PRAM simulator.

```
char *pcs( pr char *ptr, pr int mode ) {
  pr int t, my_index, *result;
  result = NULL;

  join(for(t=0;t<2;t++); 0; 1000)  {
    if (mode == FREE) {
      my_index = mpadd( &high, 1 );
      avail[my_index % N] = ptr;
      /*insert ptr into queue of free blocks*/
    }
    if (mode == ALLOC)  {
      my_index = mpadd( &low, 1)
      if (my_index  >= high) {
        /*sorry, cannot get block from queue*/
        result = NULL;
        mpadd( &low, -1 );
        /*value of low must be corrected.*/
      }
      else result = avail[my_index % N];
  } }
  else retry;
  return result;
}
```

This implementation should be contrasted to a conventional one protecting access to variables `low` and `high` using locks (see Section 1.3.2). Note that the simplicity of our problem is rather favouring the asynchronous implementation. The run time figures for both implementations run with a varying number of processors are given in Figure 1.7. The break–even point is here approximately at 128 processors. For smaller numbers of processors the overhead spent in `join` and for maintaining exact synchronicity outweighs most of the parallel speed-up, and the density of queries over time is not dense enough to outperform the asynchronous implementation. For large numbers of processors the run time for the asynchronous variant grows linearly in the number of processors since the sequentialization due to the sequential critical section dominates the execution time.

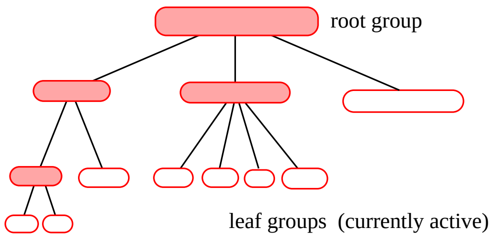


Figure 1.8 The group hierarchy in Fork95 forms a logical tree.

1.4.7 Exercises

(1a) Implement a *sequential* critical section using `join`.

(1b) Implement the functionality of the `seq` statement using `join`.

(2) Write an own implementation of the `join()` macro.

(3a) Improve the `pcs` program in Section 1.4.6.3 by variable block sizes and a more sophisticated block allocation method (*best fit*): Keep the list of free blocks ordered by increasing size and use a data structure (e.g. a search tree) to enable fast access to the smallest block whose size is still greater than or equal to the one demanded for by `balloc()`. Figure out an algorithm that allows to process several `balloc`/ `bfree` queries (see Figure 1.6) synchronously in parallel.

(3b) Write an asynchronous implementation for the problem (3a) and compare its performance to your synchronous implementation for (3a).

(4) Write a synchronous implementation of a parallel queue, i.e., pack the `put` and `get` accesses in a `join` statement with appropriate parameters. Compare the results with the asynchronous implementation of the parallel queue described in Section 1.3.4 for different numbers of parallel list heads.

1.5 Synchronous program execution and the group concept

In synchronous mode, Fork95 programs are executed by *groups* of processors, rather than by individual processors. Initially, there is just one group containing all available processors. Groups may be recursively subdivided. Thus, at any point of program execution, all presently existing groups form a tree–like[16] *group hierarchy*. Only the *leaf groups* of the group hierarchy are active (see Figure 1.8).

1.5.1 Group ID `@` and group–local processor ID `$`

Each group has a (shared) group ID. A processor can inspect the group ID of its current group by the shared integer variable `@`. `@` is primarily intended to distinguish the subgroups of a group. For this reason, the value of `@` can be modified by the programmer. Initially, at `start`, `join`, and at the beginning of program execution, `@` is set to 0 but may be redefined appropriately during program execution.

[16]If `join` statements are executed, the group hierarchy rather forms a *forest* since each "bus tour" due to a `join` statement installs a new root group. As this difference is not important for the following discussions, we omit it for simplicity of presentation.

The special private variable `$`[17] is intended to hold the current *group–relative processor ID*. `$` is initially set to the processor ID `__PROC_NR__` but may be redefined during program execution. `start` and `join` renumber `$` subsequently from 0 to the group size minus one.

Both the group–relative processor ID `$` and the group ID `@` are automatically saved when splitting the current group, and restored when reactivating it. However, the user is responsible to assign reasonable values to them.

1.5.2 Synchronicity invariant

At each point of program execution in synchronous mode, Fork95 maintains the synchronicity invariant which requires that *all processors belonging to the same active processor group are operating strictly synchronously, i.e., they follow the same path of control flow and execute the same instruction at the same time.* Also, all processors within the same group have access to a common shared address subspace. Thus, newly allocated shared variables exist once for each group allocating them.

A processor can inspect the number of processors belonging to its current group using the routine

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

At the entry into a synchronous region, the processors form one single processor group. However, it may be possible that control flow diverges at conditional branches whose condition depends on private values. To guarantee synchronicity within each active subgroup, the current leaf group must then be split into subgroups.

We consider an *expression* to be *private* if it is not guaranteed to evaluate to the same value on each processor, i.e. if it contains a private variable like `$`, a function call, or a multiprefix operator (see Section 1.3.1). Otherwise it is shared which means that it evaluates to the same value on each processor of the group.

1.5.3 Synchronous `if` statements with private condition

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

At an `if` statement with a private condition

```
if ( $ % 3 )
   statement1;          /* then-part (condition was true) */
else
   statement0;          /* else-part (condition was false) */
```

maintaining the synchronicity invariant causes the current processor group G to be split into two subgroups G_1 and G_0 (see Figure 1.9): the processors for which the condition evaluates to a non–zero value form the first subgroup G_1 and execute the `then` part while the remaining processors form subgroup G_0 and execute the `else` part. The parent group G is deactivated; its available shared address subspace is subdivided among the new subgroups.

[17] In the old FORK proposal [HSS92], the group–relative processor ID was denoted by `#`.

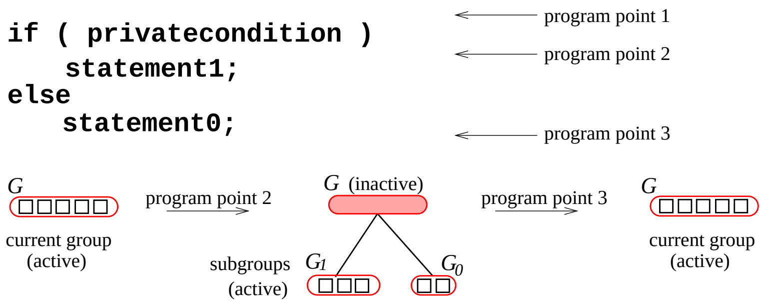


Figure 1.9 In synchronous mode, an `if` statement with a private condition causes the current group to be split into two subgroups. Processors are visualized by small rectangular boxes.

Each subgroup can now declare and allocate shared objects relating only to that subgroup itself. Also, the subgroups inherit their group IDs `@` and the group-relative processor IDs `$` from the parent group but may redefine them locally.

When both subgroups G_1 and G_0 have finished the execution of their respective branch, they are released and the parent group G is reactivated by an explicit group–wide barrier synchronization of all its processors.

Nested synchronous `if` statement with private conditions result in nested subgroup splitting, as can be seen for the following example program:

```
#include <fork.h>
#include <io.h>

main() {
  pr int i = $;
  start {
    if ($<2) {
      @ = 17;
      $ = 33-$;
      if (i==0)
        farm pprintf("then-branch, $=%d @=%d\n", $, @);
      else
        farm pprintf("else-branch, $=%d @=%d\n", $, @);
    }
    else farm pprintf("do nothing\n");
  }
}
```

which, started with four processors, produces the following output:

```
PRAM P0 = (p0, v0)> g
#0002# do nothing
#0003# do nothing
#0000# then-branch, $=33 @=17
#0001# else-branch, $=32 @=17
EXIT: vp=#0, pc=$000002a3
```

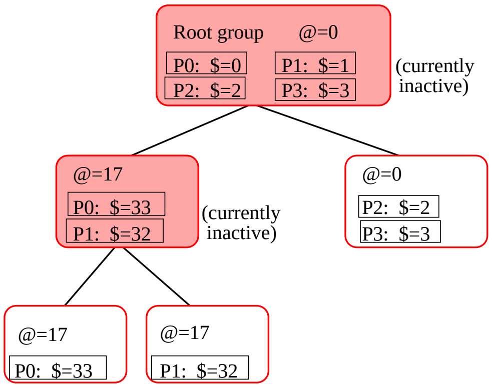


Figure 1.10 Group hierarchy for the example program

```
EXIT: vp=#1, pc=$000002a3
EXIT: vp=#2, pc=$000002a3
EXIT: vp=#3, pc=$000002a3
Stop nach 21725 Runden, 869.000 kIps
```

Figure 1.10 shows the group hierarchy at the point of time when the branches of the innermost `if` statement are processed.

Typical applications for the private synchronous `if` statement are parallel divide–and–conquer algorithms which split a given problem into two subproblems (divide step) and recursively solve these before composing the subproblem solutions to a solution of the given problem (conquer step).

A classical example is the Quickhull algorithm [PS95] which computes the convex hull of n points in the Euclidean plane (see Figure 1.11, left hand side). The points are given as an array `pt` of indices into arrays `x` and `y` giving their x and y coordinates. The main idea is as follows: We start by determining the points u, v with minimal resp. maximal x coordinate. Clearly, u and v belong to the convex hull (see Figure 1.11, right hand side).

Then we split the set of remaining points into a set S_{upper} of points located to the left of the line (u, v), and S_{lower} of points located to the right of (u, v). For both S_{upper} and S_{lower}, we have now reduced the problem to determining the upper hull of a set S of points located above a line segment, including the two extremal points p_1 (left) and p_2 (right) of the line. In particular, for S_{upper} we have $S = S_{upper} \cup \{u, v\}$ with $p_1 = u$ and $p_2 = v$; for S_{lower} we set $S = S_{lower} \cup \{v, u\}$ with $p_1 = v$ and $p_2 = u$.

Now we apply the following recursive method to S and (p_1, p_2): We determine the point $pivot \in S$ (called the pivot point) with the largest distance from line (p_1, p_2) (see Figure 1.12, left hand side), i.e. which maximizes the cross product

```
#define cross(pivot,p1,p2) \
 ((x[p1]-x[pivot])*(y[p2]-y[pivot]) - (y[p1]-y[pivot])*(x[p2]-x[pivot]))
```

Obviously, $pivot$ belongs to the convex hull over (p_1, p_2), while the points located in the interior of the triangle $(p_1, pivot, p_2)$ cannot belong to the convex hull and are eliminated from S. Among the remaining points, those located to the left of line $(p_1, pivot)$ may still belong to the upper hull over $(p_1, pivot)$ and are thus collected in a set S_1. The other points, located to the left of $(pivot, p_2)$, are candidates for the upper hull over $(pivot, p_2)$ and are collected in a set S_2 (see Figure 1.12, right hand side). Then we apply the method recursively to S_1 with $(p_1, pivot)$ and to S_2 with $(pivot, p_2)$ until the subproblems become trivial. The points belonging to the convex hull are marked.

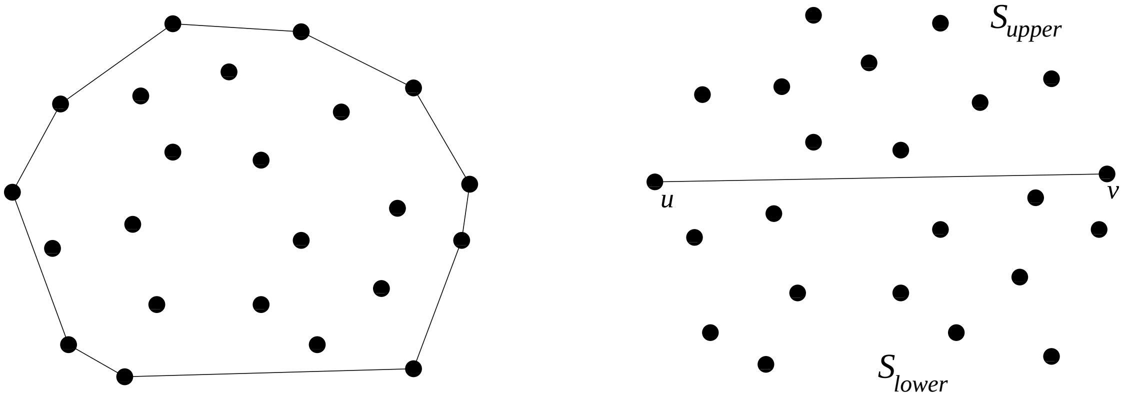


Figure 1.11 Left hand side: The points belonging to the convex hull are connected by a closed convex polygon. Right hand side: The first step of the Quickhull algorithm divides the problem into two subproblems: computing the upper resp. the lower hull.

The recursive core routine may look as follows:

```
sync void qh( sh int *pt, sh int n )
{
 /* We have p1==pt[0],p2==pt[1] */
 sh int pivot;
 sh int p = 0;                /* number of available procs */
 sh int p1=pt[0], p2=pt[1];   /* the line (p1,p2) */
 sh int *S1, *S2, n1, n2;     /* to hold subproblems */
 sh int firstrightproc;

 $ = mpadd( &p, 1 );   /* renumber $ and determine p */
 if (n==2) return;
 if (n==3) {  /* one point (beyond old p1,p2) must belong to hull */
    belongs_to_hull[pt[2]] = 1;
    return;
 }
 if (p==1) { seq_qh( pt, n ); return; }

 pivot = search_pivot( pt, n );
 belongs_to_hull[pivot] = 1;

 /* determine subproblems: */
 n1 = n2 = n;
 S1 = delete_right( pt, &n1, p1, pivot );
 S2 = delete_right( pt, &n2, pivot, p2 );

 /* determine amount of processors dedicated to each sub-problem:*/
 firstrightproc = (int)floor( (float)p * est_work(n1) / est_work(n) );
 if (firstrightproc == 0 && n1 > 0) firstrightproc = 1;

 if ($ < firstrightproc)   /*split group of processors: */
    qh( S1, n1 );   else     qh( S2, n2 );
}
```

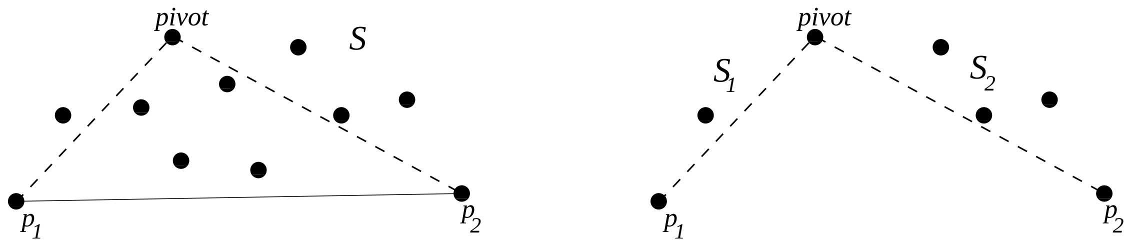


Figure 1.12 The divide step of the Quickhull algorithm.

The splitting of the group into two subgroups which operate in parallel on the two subproblems is done in such a way that the number of processors dedicated to each subgroup is approximately proportional to the work estimated for solving the corresponding subproblem. For the Quickhull algorithm, we know that the expected work is $O(n \log n)$, thus we set

```
#define est_work(n) ((float)((n) * (ilog2(n)+1)))
```

where the `ilog2()` function[18], applicable to integer operands, returns the floor of the base–2 logarithm of its argument.

1.5.4 Synchronous loops with private exit condition

A similar subgroup construction is required also at loops with a private exit condition. Assume that the processors of a leaf group G arrive in synchronous mode at a loop statement. All processors that will execute the first iteration of the loop form a subgroup G' and stay therein as long as they iterate (see Figure 1.13).

Once the loop iteration condition has been evaluated to zero for a processor in the iterating group G', it leaves G' and waits at the end of the loop to resynchronize with all processors of the parent group G. However, at loops it is not necessary to split the shared memory subspace of G, since processors that leave the loop body are just waiting for the last processors of G to complete loop execution.

As an illustration, consider the dataparallel loop

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

This causes the private variable `i` of processor k, $0 \leq k \leq p-1$, to loop through the values k, $k+p$, $k+2p$ etc. The whole index range from 0 to $\mathtt{n}-1$ is covered provided that the involved processors have ID's `$` consecutively ranging from 0 to $p-1$. Note that this can also be applied in asynchronous context as long as the programmer guarantees for all required processors to arrive at that statement.

In order to enhance program understandability, one may use the macro

[18] `ilog2()` is available as an *expression operator* in Fork95 and executes on the SB-PRAM by the one–cycle `rm` instruction.

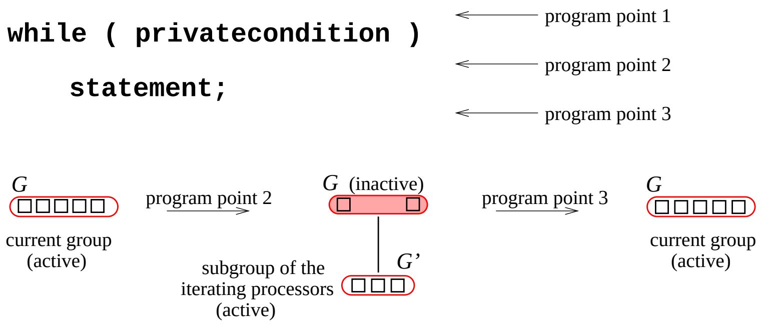


Figure 1.13 In synchronous mode, a loop (here: `while`) with a private exit condition causes the current group G to form a subgroup G' for the iterating processors.

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

Then, the above loop may be just written as

```
forall (i, 0, n, p)  a[i] += c * b[i];
```

1.5.5 The `fork` statement

Splitting into subgroups can, in contrast to the implicit subdivision at private branch conditions, also be done explicitly, by the `fork` statement[19]. The semantics of

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

is the following: First, the shared expression e_1 is evaluated to the number (say, g) of subgroups to be created. Then the current leaf group is split into that many subgroups, whose group ID's `@` are indexed $0, ..., g-1$ (see Figure 1.14). Evaluating the private expression e_2, every processor determines the index of the newly created subgroup it wants to become a member of. Clearly, the value of e_2 must be within the range $0, ..., g-1$. Finally, by evaluating the private expression e_3, a processor can renumber its current group–relative processor ID within the new leaf group. The assignment to `$`, though, can also be omitted if renumbering `$` is not required. Note that empty subgroups (with no processors) are possible; an empty subgroup's work is immediately finished, though.

Continuing, the parent group is now deactivated. Its shared memory subspace is partitioned into g slices, each of which is assigned to one subgroup. Thus, each new subgroup has now its own shared memory subspace.

Then, each subgroup continues by executing `<stmt>`; the processors within each subgroup work synchronously, but different subgroups can choose different control flow paths.

[19] Similar to the discussion in footnote 12 on page 25, the keyword `fork` was chosen due to historical reasons [HSS92]. From today's point of view, something like `split` would be better suited because this avoids confusion with the UNIX `fork` mechanism which denotes spawning several child processes from a single process, while in Fork95 the number of processes remains constant and only the organization of them into groups is rearranged. If you like a different keyword, just apply the C preprocessor ...

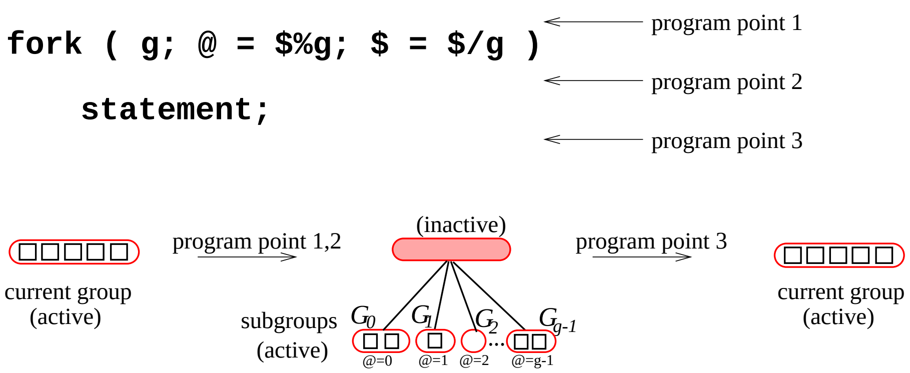


Figure 1.14 The `fork` statement splits group G into g subgroups G_0, G_1, ..., G_{g-1}.

After a subgroup has completed the execution of the body `<stmt>`, it is abandoned. All processors of the parent group are synchronized by an exact barrier; the parent group is then reactivated as the current leaf group. The shared memory subspaces of the former subgroups are re–merged and can now be overwritten by the parent group.

Finally, the statement following the `fork` statement is executed again synchronously by all processors of the parent group.

Another example for a parallel divide–and–conquer algorithm, here implemented using the recursive application of the `fork` statement, is the parallel drawing of fractals known as *Koch curves* (see e.g. chapter 10-18 of Hearn/Baker [HB94] for an introduction to fractals in computer graphics). A Koch curve of degree d is obtained recursively by replacing a straight line segment in a curve of degree $d-1$ by a so–called generator pattern. The procedure is initialized by an initiator figure as a Koch curve of degree zero (see Figure 1.15).

Drawing a Koch curve of degree `DEGREE` in parallel is straightforward using the following recursive Fork95 function which refines a straight line segment connecting two points `(startx,starty)` and `(stopx,stopy)`. The shared parameter `level` indicates the current level of recursive refinement.

```
sync void Koch ( sh int startx, sh int starty,
                 sh int stopx,  sh int stopy,  sh int level )
{
 sh int p;
 if (level >= DEGREE) {   /*reach the limit of recursion*/
    line( startx, starty, stopx, stopy, level+2, 1 );
    return;
 } /* else: another recursion step */
 p = groupsize();
 if (p<4)
    seq  seq_Koch( startx, starty, stopx, stopy, level );
 else {
    sh int x[5], y[5], dx, dy;
    pr int j;
    seq {         /* linear interpolation: */
```

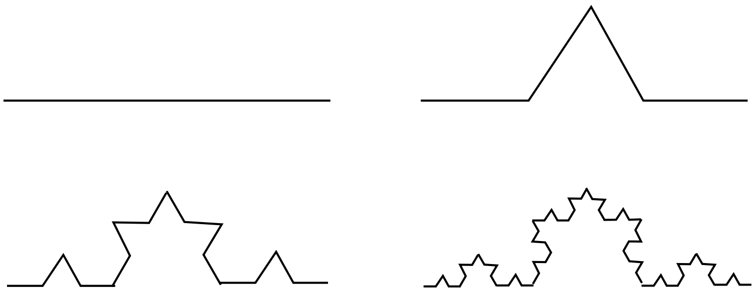


Figure 1.15 Koch curves of degree 0, 1, 2, and 3 for a straight line segment initiator (left upper figure). The generator pattern is here identical to the degree 1 curve (right upper figure).

```
      dx = stopx - startx;       dy = stopy - starty;
      x[0] = startx;             y[0] = starty;
      x[1] = startx + (dx/3);    y[1] = starty + (dy/3);
      x[2] = startx + dx/2 - (int)(factor * (float)dy);
      y[2] = starty + dy/2 + (int)(factor * (float)dx);
      x[3] = startx + (2*dx/3); y[3] = starty + (2*dy/3);
      x[4] = stopx;              y[4] = stopy;
   }
   fork ( 4; @=$%4; $=$/4 )
      Koch( x[@], y[@], x[@+1], y[@+1], level + 1 );
 }
}
```

The routine `line`(x_1, y_1, x_2, y_2, c) plots in parallel a line connecting two points (x_1, y_1) and (x_2, y_2) with color c. This and the other required graphics routines are declared in `graphic.h` and summarized in Table 1.3 in the Appendix, page 59. The constant `factor` may be used to distort the curve; its default value is `0.33`.

Future development of the `fork` statement A future implementation of Fork95 may extend the `fork` statement as follows: As an additional (fourth) parameter, the user specifies a *minimal subgroup size* m. If, during splitting the current group, one of the subgroups obtains less than m processors, the subgroups are executed in sequential instead of in parallel, i.e. each subgroup is executed by all processors of the parent group. Note that this scheme provides a straightforward way of virtual processing for parallel divide–and–conquer computations.

1.5.6 Accessing objects shared by different groups

While all processors inside a leaf group operate strictly synchronously, different leaf groups will, in general, work asynchronously with respect to each other. This may produce problems when two different leaf groups want to access the same shared object. Such a shared object may be either global or be allocated by a common predecessor group of these groups in the group hierarchy tree. If the accesses are not atomic, and at least one of the groups wants to *write* to such an object, the accesses to it become a critical section.

For instance, look at the following test program:

```
sh struct cmplx_int {
  int re;
  int im;
} common;

main()
{
 start
   if ($ % 2) {
     common.re = 45;
     common.im = 2 * common.re;
   }
   else {
     common.im = 17;
     common.re = 23;
   }
 if (__PROC_NR__==0)
   printf("common.re = %d, common.im = %d\n", common.re, common.im);
 barrier;
}
```

The `if` statement causes the root group to be split into two subgroups; these execute their branches in parallel. Executing this program with four processors, we obtain:

```
PRAM P0 = (p0, v0)> g
common.re = 23, common.im = 90
Stop nach 12779 Runden, 851.933 kIps
```

This is probably not what the programmer expected; (s)he would rather like to have access to `common` being atomic, which means that `common` finally has either the value {`45,90`} or {`23,17`}, depending on which of the groups did its assignment last.

Such conflicts can again be managed using locks. Nevertheless, the implementation introduced in Section 1.3.2.1 cannot be reused without modification, as all processors of a group should get access to the critical section simultaneously and as a whole. Thus we need a *group lock* mechanism.

A simple solution using the existing lock mechanisms may be to select one processor for each group who should be responsible for the locking/unlocking mechanism [20]. This is done by masking the locking/unlocking operations by `seq` statements:

```
sh struct cmplx_int {
  int re;
  int im;
  fair_lock fglock;   /* used as a fair group lock */
} common;
```

[20] Note that the compiler casts the `fair_lockup` and `fair_unlock` calls to asynchronous mode automatically.

```
main()
{
 start {
   fair_lock_init(&(common.fglock));
   if ($ % 2) {
      seq  fair_lockup( &(common.fglock));
      common.re = 45;
      common.im = 2 * common.re;
      seq  fair_unlock( &(common.fglock));
   }
   else {
      seq  fair_lockup( &(common.fglock));
      common.im = 17;
      common.re = 23;
      seq  fair_unlock( &(common.fglock));
   }
 }
 if (__PROC_NR__==0)
   printf("common.re = %d, common.im = %d\n", common.re, common.im);
 barrier;
}
```

Now we get the desired output:

```
PRAM P0 = (p0, v0)> g
common.a = 45, common.b = 90
Stop nach 13179 Runden, 753.086 kIps
```

1.5.7 Exercises

(1) Write a synchronous recursive implementation of Strassen's Matrix–Matrix multiplication (see [Str69] or e.g. [PTVF92]) for floatingpoint numbers using the `fork` statement. Strassen's algorithm is asymptotically faster ($O(n^{\log_2 7})$) than the straightforward method ($O(n^3)$) for Matrix–Matrix multiplication. Compare the performance of your implementation on $7^4 = 2401$ processors, given a sufficiently large matrix size, (e.g. 256×256) with a straightforward implementation of Matrix–Matrix multiplication.

(2a) Complete the implementation of the parallel *Quick-Hull* algorithm sketched in subsection 1.5.3. You may use the routines in `graphic.h` (see also Table 1.3, page 59 in the Appendix) to verify the results.

(2b) In the `qh` routine, splitting the current group is not necessary if one of the subproblems is empty, i.e. if `n1`$= 0$ or `n2`$= 0$. Modify the routine according to this optimization.

(3) Complete the implementation for drawing Koch curves in parallel, as sketched in subsection 1.5.5. Apply it to a regular–sided triangle as initiator — then you should obtain the famous "Koch snow-flake".

1.6 Pointers and heaps

1.6.1 Pointers

Fork95 offers pointers, as opposed to its predecessor FORK. The usage of pointers in Fork95 is as flexible as in C, since all private address subspaces have been embedded into the global shared memory. In particular, one does not have to distinguish between pointers to shared and pointers to private objects, in contrast to some other parallel programming languages, for example C^* [RS87]. Shared pointer variables may point to private objects and vice versa; the programmer is responsible for such assignments making sense. Thus,

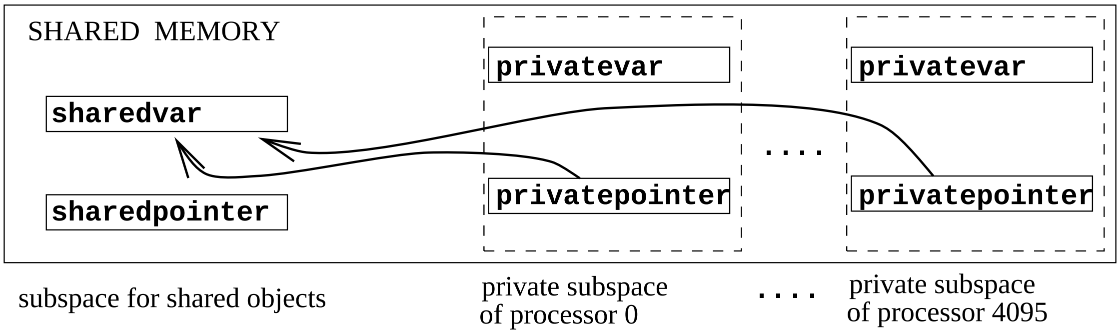
```
sh int *sharedpointer;
```

declares a shared pointer which may point either to a private or to a shared location.

For instance, the following program fragment

```
pr int privatevar, *privatepointer;
sh int sharedvar;
privatepointer = &sharedvar;
```

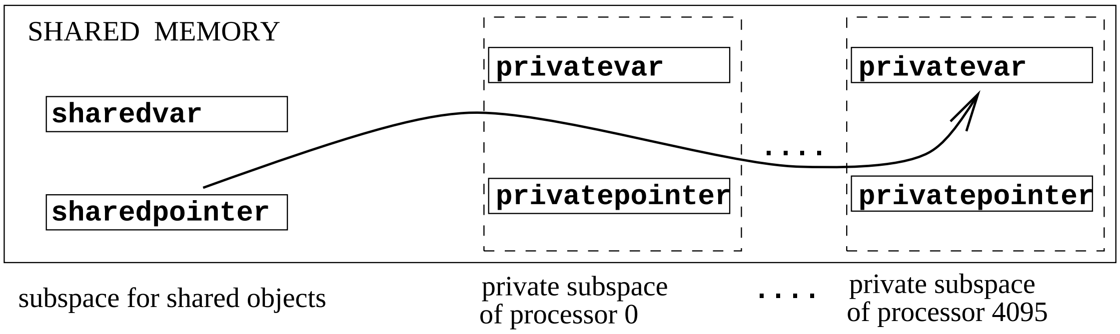
causes the private pointer variable `privatepointer` of each processor participating in this assignment to point to the shared variable `sharedvar`.



Accordingly, if all processors execute the following statement simultaneously

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

the shared pointer variable `sharedpointer` is made point to the private variable `privatevar` of the processor with the highest processor ID participating in this assignment.



Thus, a shared pointer variable pointing to a private location nevertheless represents a shared *value*; all processors in the scope of the group that declared that shared pointer see the *same* private object `privatevar` through that pointer. In this way, private objects can be made globally accessible.

1.6.2 Heaps

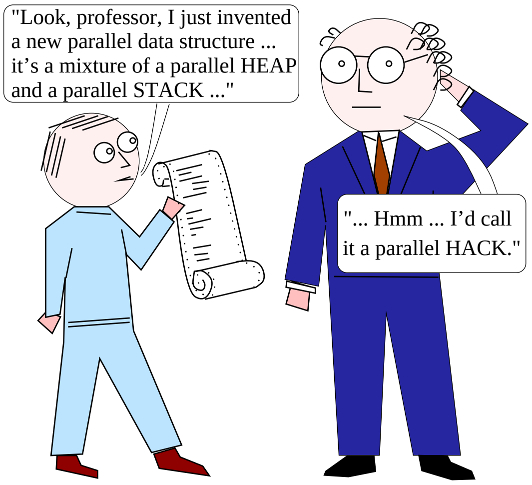
Fork95 supplies three kinds of heaps: one permanent, private heap for each processor, one automatic shared heap for each group, and a global, permanent shared heap.

Space on the private heaps can be allocated and freed by the asynchronous functions `malloc()` and `free()` known from C.

Space on the permanent shared heap is allocated and freed accordingly using the asynchronous functions `shmalloc()` and `shfree()`.

The automatic shared heap is intended to provide fast temporary storage blocks which are local to a group.

Consequently, the life range of objects allocated on the automatic shared heap by the synchronous `shalloc()` function is limited to the life range of the group by which that `shalloc()` was executed. Thus, such objects are automatically removed if the group allocating them is released, just in the same way as the local variables of a function allocated on the run time stack cease to exist when the function terminates. In this way, the automatic shared heap is really something in between a parallel stack and a parallel heap.



For better space economy, there is a synchronous function `shallfree()` which frees *all* objects `shalloc`ed so far in the current (synchronous) function.

Note the following important difference: If p processors call `shalloc()` concurrently, each processor obtains a copy of the address of *one* allocated block. In contrast, if p processors call `shmalloc()` concurrently, each processor obtains an individual pointer to an individually allocated block, i.e. p blocks are allocated altogether.

1.6.3 Pointers to functions

Pointers to functions are also supported. However, special attention must be paid when using private pointers to functions in synchronous mode:

```
sync int fp_example ( ..., pr void (*fp)(void), ... )
{ ...
  *fp();    /*! this may call several different functions !*/
  ...       /*! call to *fp is casted to asynchronous mode */
}
```

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

In contrast, shared function pointers may also point to synchronous functions, as control flow is not affected:

```
sync int fp_example ( ..., sh sync void (*fp)(void), ... )
{ ...
  *fp();   /* all processors of the group call the same function */
  ...
}
```

1.6.4 Exercises

(1) Write a generic synchronous recursive Quicksort routine. Pass to this function as a parameter the pointer to the compare function.

1.7 Caveats in Fork95 programming

In general, Fork95 offers a lot of comfort to the programmer. Nevertheless, there are some problems and common pitfalls the programmer should take care of. These are partly inherent to the language itself (Subsections 1.7.1, 1.7.2, 1.7.3) and motivated by the design goals of performance and generality, and partly they are due to weaknesses of the current implementation (Subsection 1.7.4) which are likely to disappear in a future version of the compiler.

1.7.1 Spaghetti jumping

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

For jumps of type `break`, `continue`, and `return`, the target group is statically known; it is a predecessor of the current leaf group in the group hierarchy tree. In this case, the compiler provides a safe implementation even for the synchronous mode.

For a `goto` jump, however, the target group may not yet have been created at the time of executing the jump. Even worse, the target group may not be known at compile time. Jumps across synchronization points usually will introduce a deadlock. For this reason, `goto` jumps are under the programmer's responsibility. However, as long as source and destination of a `goto` are within the same asynchronous region, there is no danger of deadlock.

1.7.2 Shared memory fragmentation

The reader may already have noticed that it is not wise to have more `fork` or private `if` statements on the recursive branch of a recursive procedure (like parallel depth–first–search, for instance) than absolutely necessary. Otherwise, after only very few recursion steps, the remaining shared memory fraction of each subgroup has reached an impracticably small size thus resulting in early stack overflow. As static program analysis for C (and thus also for Fork95) is generally hard, even a clever compiler will be often unable to analyze the space requirements of the subgroups and thus, following a worst–case strategy, assign equally sized slices of the parent group's shared memory subsection to the child groups.

If the programmer statically knows a group to consist of only one processor, (s)he should better switch to asynchronous mode because this avoids the expensive time penalty of continued subgroup formation and throttling of computation by continued shared memory space fragmentation.

1.7.3 The `asm` statement

As some older C compilers, the current Fork95 implementation offers the `asm` statement which allows to include inline–assembler sections directly into the C source.

`asm()` takes a constant string as argument; the string is directly copied to the assembler file produced by the compiler. Errors in the assembler string will be reported by the assembler `prass`.

Additionally, accesses to a local variable [21] or function parameter `var` can be expressed in this string as `%var`, as in the following example:

```
unsigned int rc = lock->readercounter;
asm("bmc    0\n\
     gethi  0x40000000,r31 /*__RW_FLAG__*/\n\
     syncor r31,%rc\n\
     nop    /*delay*/");
```

Fork95 admits this low–level programming with inline–assembler for efficiency reasons, but it should only be used as a last resort, as it makes programs less portable and less understandable. Moreover, it disables any compiler optimizations on that part of the program, since the compiler does not really "understand" what happens inside the `asm` code — it just trusts the programmer.

1.7.4 Short–circuit evaluation of `&&` and `||`

We decided to follow C's semantics of evaluating the binary logical operators `&&` and `||` in a short–circuit way. This means that e.g. in

[21] In the current implementation this replacement mechanism works only for shared formal parameters and for shared local variables defined at the top level of a synchronous function. Furthermore, global variables (private and shared) cannot be resolved by this method. These restrictions are caused by the fact that block–local shared variables not defined at top–level do not have a unique addressing scheme, and that addresses of global variables cannot be generated within one instruction.

```
sync void crash ( pr int a )
{
    pr int c = a && foo(...);
    ...
}
```

the call to `foo()` is not executed by the processors which evaluated `a` to a zero value. Already in sequential C programming this may be a pitfall for programmers if side effects in the argument's evaluation or in the execution of `foo()` are not foreseen. In the synchronous mode of Fork95, this may additionally lead to unintended asynchronity and thus to race conditions or even to a deadlock. As there exist obvious workarounds to avoid such situations, the current implementation renounces on an explicit splitting of the current group into subgroups to maintain synchronicity. Rather, the compiler emits a warning in this case.

A straightforward workaround would be to rewrite the logical expression above as

```
sync void no_crash ( pr int a )
{
   pr int c;
   if (a)                /*group splitting*/
          c = foo(...);
   else   c = 0;
   ...
}
```

As an alternative, one may, if the side–effects of short–circuit evaluation are not explicitly desired, replace a logical operation `A && B` by the expression `((A?1:0)*(B?1:0))`, and the logical operation `A || B` by the bitwise operation `(A | B)?1:0)` which are free of race conditions.

Furthermore, one may, if possible, switch to asynchronous mode during the evaluation of the logical expression if `foo()` could be rewritten as an asynchronous function.

1.8 Other parallel programming languages

In this survey (which is surely not exhaustive), we mainly focus on languages which support the shared memory programming model. Our classification is summarized in Table 1.1. For a more general survey on parallel programming languages, we refer to [BST89], [SE95] and [GG96].

1.8.1 Asynchronous parallel programming languages

1.8.1.1 The library approach

One approach of introducing parallelism into languages consists of 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

	shared memory model			message passing
	SIMD–oriented	SPMD–oriented	MIMD–oriented	(MIMD)
support exactly synchronous execution	NESL, V, C^*, Dataparallel C, Fortran 90, Vector-C, APL, CM-Fortran	HPF, Fortran-D, Vienna-Fortran, MPP-Fortran	FORK, Fork95, ll, pm2, Modula-2* (*PRAM–oriented*)	—
no exactly synchronous execution	—		ParC, Cilk 3.0, Sequent-C, LINDA	CSP, occam, PVM, MPI

Table 1.1 Classification of programming languages

extending C or FORTRAN have been proposed and implemented on a broad variety of parallel machines.

The currently probably most widely used library for distributed memory systems is PVM [GBD$^+$94], which is based on CSP [Hoa78, Hoa85]. It offers various routines for spawning additional tasks from an existing task, aborting tasks, sending and receiving messages, more complex communication patterns like gather, combine and broadcast. PVM gained much popularity because it is freely available on many different parallel hardware platforms; because of compatibility reasons, PVM often is even used for programming shared memory systems although this usually degrades performance compared with direct shared memory programming.

A more recent approach with similar and slightly extended functionality is the message passing interface library MPI [SDB93, CGH94]. There exist free and commercial implementations of MPI for many different parallel hardware platforms.

The P4 library[22] and its relatives support the shared memory programming model as well. The basic primitives provided for shared memory are semaphores and locks. Moreover, it provides shared storage allocation and a flexible monitor mechanism including barrier synchronization [BL92, BL94].

LINDA [CG89, CGMS94] is another library approach for asynchronous MIMD programming. It models the shared memory as an associative memory, called the *tuple space*, which is accessed by the processors asynchronously by atomic put and get operations.

The asynchronous MIMD 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.

1.8.1.2 Process–oriented parallel programming languages

Most process–oriented languages do not offer a mode of strictly synchronous program execution.

ParC [BAFR96] provides asynchronous processes. Parallel processes can be spawned from a single process implicitly by a `forall` loop (for concurrent execution of loop iterations) or explicitly by the `par{ :: }` statement. This results in a tree of processes where child

[22] The P4 library has been implemented for the SB-PRAM [Röh96].

processes share the stack of the parent process (*cactus stack*); in this way there is also an implicit sharing of variables by scoping.[23] Nevertheless there are no global private variables. Processes are mapped automatically to processors; the programmer can optionally specify a directive to explicitly map `forall` loop iterations to processors. ParC also offers an atomic fetch&add primitive. A nice feature is that a process is able to kill all its siblings and their descendants in the process tree. On the other hand, control flow must not exit a block of statements in an abnormal way (such as `return`, `continue`, `break`).

Cilk [BJK$^+$95, BFJ$^+$96] is a multithreaded language extending C. It is primarily intended to compile to common distributed memory platforms. There are language constructs to define (light–weight) threads, to spawn new threads, and for message passing. Version 3.0 offers a distributed shared memory model. Processors are not explicit in Cilk's programming model. The scheduling of the threads (with the goal of load balancing) is done automatically by the run-time system.

1.8.2 Synchronous parallel programming languages

Attempts to design synchronous languages have been made mainly 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 pipelined vector processors or processor arrays such as the CM2. Examples of such languages are MODULA-2* [PM94], C^* [RS87] or its relatives DATAPARALLEL C [HQ91] and DBC [SG92].

The data–parallel Fortran dialects partially allow to exploit also MIMD parallelism in a very restricted way; proper task parallelism is not provided. Thus we summarize them as *SPMD–oriented* languages.

However, these languages support only one global name space. Other parallel computation paradigms like a parallel recursive divide–and–conquer style as suggested in [BDH$^+$91], [Col89], [dlTK92], [HR92a], [HR92b] are not supported. This is the domain of MIMD languages. By *PRAM–oriented* languages, we classify those MIMD languages which support exact synchronicity.

1.8.2.1 SIMD–oriented languages

SIMD–oriented languages maintain a sequential flow of program control. This is either naturally enforced by SIMD hardware (e.g., vector processors or array processors), or installed by the compiler when the target is a MIMD machine. To exploit SIMD parallelism, it seems quite natural to extend existing sequential programming languages by vector statements using an array syntax and suitable intrinsic functions like vector sum, dot product, or matrix transpose. Examples are *Fortran90* [MR90], *CM-Fortran* [AKLS88], *Vector-C* [LS85], or just *APL* [Ive62].

[23] Note that this is different from the UNIX `fork` command for spawning processes where a child process gets a *copy* of the parent's stack.

NESL [Ble96] is a functional data–parallel language partly based on ML. Its main data structure is the (multidimensional) list. All program objects are shared. Elementwise operations on lists are converted to vector instructions (*flattening*) and executed on SIMD machines. In contrast, the MIMD-oriented Fork95 also allows for asynchronous and task parallelism, low-level PRAM programming and direct access to shared memory locations. Recent work [CSS95, ACD+97] applies the hierarchical parallel list concept of NESL to the imperative languages C (resulting in a language called *V* [CSS95]) and Fortran90 (Fortran90V, [ACD+97]).

C^* [RS87, TPH92] is a SIMD–oriented language. It extracts parallelism from special program objects called *shapes* which are explicitly parallel arrays. A grid structure of the processors must be given; for a distributed memory machine the compiler performs the required mapping of shape sections to the local memory modules of the processors. A group concept or recursive parallelism are not supported. The concept of parallel pointers is extremely complicated in C^* since in addition to the sharity of the pointer itself, the sharity of the value pointed to is, in contrast to Fork95, a part of the pointer's type specification. ViC* [CC94] is a precompiler that extends $C*$ for out–of–core computations and thus offers a virtual memory mechanism on top of C^*.

Dataparallel C [QH90, HQ91, HLQA92] is a SIMD language which is synchronous at the expression level, i.e. a processor is either active and working synchronously with all other processors, or it is inactive. The language distinguishes between sequential and data–parallel sections of code. Data–parallel program regions are marked by a `par{}` statement. For obvious reasons, `goto` jumps across `par` boundaries are forbidden (this would have a similar effect as jumping over a synchronization point in Fork95). There are three different types of sharity for a variable: `mono`, corresponding to Fork95's `sh`, which means that the variable resides on the host; `poly`, which denotes a shared variable being replicated such that each processor holds a copy, and `local`, corresponding to Fork95's `pr`, which means that one copy of the variable resides on each processor. Pointer declarations are still more complicated than in Fork95; the sharity of the pointee must be statically known, and conversions are more restricted. A group concept does not exist.

CM-Fortran [AKLS88] extends Fortran 77 with array syntax and directives for alignment and data distribution. The virtual processing supported by the operating system of the CM hardware is transparent to the programmer.

1.8.2.2 SPMD–oriented data–parallel languages

Further data–parallel languages in this tradition (see [Qui94] for a summary) are the various data–parallel Fortran dialects such as *Vienna Fortran* [CMZ92], *Fortran-D* [HKT91], *MPP-Fortran* [MS94], *High Performance Fortran (HPF)* [HPF93].

These languages still offer SIMD–oriented semantics but are mainly targeted towards distributed memory MIMD machines and thus require data layout directives to perform efficiently. Since the hardware cannot offer exact synchronicity, this is emulated by the compiler by analyzing the data dependencies and enforcing these by suitably inserted message passing code. Virtual processing is done implicitly by specifying array decompositions.

1.8.2.3 PRAM–oriented languages

In contrast to SIMD–oriented languages, PRAM–oriented languages provide autonomous processors; program control is no longer centralized, each processor has its own program pointer and sometimes even its own program memory. Nevertheless, as there is a common clock, execution of an instruction starts and ends at the same point of time on all processors. Moreover, executing an instruction takes the same time on all processors. For instance, loading a (word) value from the shared memory takes the same time, regardless of which processor accesses it and where it is located in the shared memory. We may call this also a *synchronous MIMD* model of programming.

In addition, a PRAM–oriented language must be able to exploit this synchronous hardware feature and make it transparent to the programmer at the language level. Such a mode of exact synchronicity must be supported by the compiler which generates code to keep program counters equal and to handle the cases where exact synchronicity may be violated due to diverging control flow. A hierarchical group concept supports this task considerably, as in Fork95. However, some PRAM–oriented languages offer only a flat group concept, i.e. there is only one level of exact synchronicity, only one group containing all processors.

Data–parallel variants of Modula, namely *pm2* [Juv92b] and *Modula-2** [PT91, HHL$^+$93, PM94], support a subset of Fork95's functionality. The main constructs to express parallelism are synchronous and asynchronous parallel loops. Exact synchronicity is supported only in special synchronous `forall` loops; there is no explicit group concept. For asynchronous `forall` loops Modula-2* offers a signal–wait mechanism for partial synchronization. Directives for processor allocation help the compiler to find a suitable distribution of the loop iterations across the processors. Private variables do not exist in Modula-2*. pm2 compiles [Juv92a] to a PRAM simulator while Modula-2* offers back-ends for several existing machines.

The only attempt we are aware of which allows both parallely recursive and synchronous programming are the imperative parallel languages FORK [HSS92], the predecessor of Fork95, and *ll* [LSRG95].

Based on a subset of PASCAL (no jumps), *ll* controls parallelism by means of a parallel *do*–loop which allows a (virtual) processor to spawn new ones executing the loop body in parallel. Opposed to that, the philosophy of Fork95 is to take a certain set of processors and distribute them over the available tasks. Given fixed sized machines, the latter approach seems better suited to exploit the processor resources.

1.8.3 The re-design of FORK

The design of FORK [HSS92] was a rather theoretical one: Pointers, dynamic arrays, nontrivial data types and non–structured control flow were sacrificed in order to facilitate correctness proofs [Sch91, RS92]. In this way, however, the language became completely unusable. A compiler [Käp92, Wel92, Lil93, Sei93] was never completed.

In order to provide a full–fledged language for real use, we have added all the language features which are well–known from sequential programming. We decided to build upon an existing and well–known sequential language since this enables better acceptance of the language and facilitates the integration of existing sequential code into a parallel implementation. Thus, for the new FORK dialect Fork95 we decided to extend the ANSI-C syntax — instead of clinging to the original one which was an idiosyncratic mixture of

Pascal and Modula. This also meant that (for the sequential parts) we had to adopt C's philosophy.

We introduced the possibility of locally asynchronous computation to save synchronization points and to enable more freedom of choice for the programming model. First only existing inside the bodies of `farm` statements [KS95b] which were originally only thought as an instrument for performance tuning and encapsulating of I/O, the asynchronous mode soon became an equally important component of the language. Asynchronous and synchronous mode complement each other. Static association of synchronous resp. asynchronous mode of execution with program regions avoids confusion and eliminates a potential source of bugs. Switching forth and back is now highly flexible by the `join` statement.

A second important decision was to abandon the tremendous run time overhead of virtual processor emulation by limiting the number of processes to the hardware resources, resulting in a very lean code generation and run time system. As we have shown in [KT97], the combination of exact synchronous program execution with virtual processing, especially for a weakly typed base language such as C, is extremely expensive to realize. In pm2 the same decision was made.

Third, we restricted the language support for exact synchronicity to the leaf groups only. In contrast, the old FORK proposal also supported larger group hierarchies running exactly synchronous. At any point of the program and for each (leaf) group, the compiler kept track of the so–called maximally synchronous group, i.e. the root of the subtree in the group tree all of whose leaf groups currently run exactly synchronous with this group considered. Originally intended to avoid some unnecessary synchronization points, it appeared that the compile–time and run–time overhead of this theoretical flourish was not useful in practice, decreased performance, and confused the programmer.

Pointers were the fourth big evolution step. Usage of pointers is very flexible now. Unfortunately, with our decision for C we also inherited its unpleasant weak type system. This disables many compile-time analyses that were possible with the old proposal [Wel92, Käp92, Sei93]. Nevertheless we think that this is a price to be paid for practically useful parallel programming. A future change of the basis language from C to Java will overcome most of this problem.

1.8.4 BSP languages

Valiant's BSP model (see [Val90]) is an alternative to the PRAM model. A library extension for C (BSPlib) supporting BSP is currently being developed in Oxford [Lec95, GHL+96]. Nevertheless, BSP enforces a less comfortable programming style than the PRAM, such that BSP programming for non–regular problems becomes quite hard.

1.8.5 Skeletons

The dream of making parallel programming as simple as sequential programming produced a discussion of a minimal set of extensions to sequential programming languages which should allow to formulate a problem as a hierarchical composition of basic parallel algorithmic paradigms, such as data–parallel computations, parallel divide–and–conquer, parallel reduction, parallel prefix, pipelining, etc. while abstracting from explicit parallelism and/or locality. Such extensions go clearly beyond simple library routines. They are often referred to as *skeletons* [Col89, DFH+93] and regarded as higher order functions, typically in the context of functional programming languages. Also imperative languages have been extended by skeletons, e.g., P^3L [BDP94] or Skil [BK96].

Skeletons are not only a means for elegant and machine–independent expression of parallelism, but also provide a more powerful hint for the compiler to generate efficient parallel code, maybe lay out data appropriately, and predict parallel run time which is again required for code optimization. A skeleton may thus be seen as a high–level interface to transfer the programmer's knowledge about the intended structure of parallel computation to the compiler.

While skeletons are intended for a top–down organization of the program, a library is better suited for bottom–up organization. At least for regular numerical applications, also the bottom–up approach is suitable to pass additional knowledge to the compiler [Keß96]. Ideally, the two approaches should complement each other. With an object–oriented successor of Fork95 this goal may be reached.

References

[ACD+97] K.T. Au, M.M. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, M. Köhler, G. Keller, W. Pfannenstiel, and M. Simons. Enlarging the Scope of Vector-Based Computations: Extending Fortran 90 by Nested Data Parallelism. In *Proc. APDC'97 Int. Conf. on Advances in Parallel and Distributed Computing*, pages 66–73. Los Alamitos: IEEE Computer Society Press, 1997.

[AKLS88] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Compiling Fortran 8x Array Features for the Connection Machine Computer Systems. In *ACM Symposium on Parallel Programming (PPEALS)*, pages 42–56, 1988.

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

[BAFR96] Y. Ben-Asher, D.G. Feitelson, and L. Rudolph. ParC — An Extension of C for Shared Memory Parallel Processing. *Software – Practice and Experience*, 26(5):581–612, May 1996.

[BDH+91] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, and S. Saxena. Improved Deterministic Parallel Integer Sorting. *Information and Computation*, 94, 1991.

[BDP94] Bruno Bacci, Marco Danelutto, and Susanna Pelagatti. Resource Optimisation via Structured Parallel Programming. In K.M. Decker and R.M. Rehmann, editors, *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland.*, pages 1–12. Basel: Birkhäuser, April 1994.

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proc. 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multi-threaded run-time system. In *Proc. 5th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 207–216, 1995.

[BK96] George H. Botorog and Herbert Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proc. 5th Int. Symp. on High Performance Distributed Computing (HPDC)*, pages 243–252. Los Alamitos: IEEE Computer Society Press, 1996.

[BL92] Ralph Butler and Ewing L. Lusk. User's Guide to the P4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.

[BL94] R. Butler and E.L. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20(4):547–564, April 1994.

[Ble96] Guy Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[CC94] Thomas H. Cormen and Alex Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Technical Report PCS-TR94-243, Dartmouth College, Computer Science, Hanover, NH, 1994.

[CG89] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989.

[CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In K.M. Decker and R.M. Rehmann, editors, *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland.*, pages 213–218. Basel: Birkhäuser, April 1994.

[CGMS94] N.J. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–656, April 1994.

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

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

[CSS95] Manuel M.T. Chakravarty, Friedrich Wilhelm Schröer, and Martin Simons. V – Nested Parallelism in C. In W. Giloi, S. Jähnichen, and B. Shriver, editors, *Proc. 2nd Int. Conf. on Massively Parallel Programming Models*, pages 167–174. Los Alamitos: IEEE Computer Society Press, 1995.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.B. Kelly, D.W.N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *Proc. PARLE'93*, pages 146–160. Springer LNCS 694, 1993.

[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.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Sept. 1994.

[GG96] Wolfgang Gellerich and Michael M. Gutzmann. Massively Parallel Programming Languages – a Classification of Design Approaches. In *Proc. ISCA Int. Conf. on Parallel and Distributed Computing Systems*, volume I, pages 110–118. ISCA, 1996. (`http://www.informatik.uni-stuttgart.de/ifi/ps/Gellerich/parlang-s.ps`).

[GHL+96] Mark W. Goudreau, Jonathan M.D. Hill, Kevin Lang, Bill McColl, Satish B. Rao, Dan C. Stefanescu, Torsten Suel, and Thanasis Tsantilas. A proposal for the BSP Worldwide Standard Library (preliminary version). `http://www.bsp-worldwide.org/`, April 1996.

[GLR83] A. Gottlieb, B. Lubachevsky, and L. Rudolph. Basic Techniques for the Efficient Coordination of Large Numbers of Cooperating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983. (see also: Ultracomputer Note No. 16, Dec. 1980, New York University).

[HB94] Donald Hearn and M. Pauline Baker. *Computer Graphics, Second Edition*. Prentice Hall, 1994.

[HHL+93] Stefan U. Hänßgen, Ernst A. Heinz, Paul Lukowicz, Michael Philippsen, and Walter F. Tichy. The Modula-2* Environment for Parallel Programming. In *Proc. 1st Int. Conf. on Massively Parallel Programming Models*. Los Alamitos: IEEE Computer Society Press, 1993.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler–Support for Machine–Independent Parallel Programming in Fortran-D. Technical Report Rice COMP TR91-149, Rice University, March 1991.

[HLQA92] Philip J. Hatcher, Anthony J. Lapadula, Michael J. Quinn, and Ray J. Anderson. Compiling Data-Parallel Programs for MIMD Architectures. In Hans Zima, editor, *Proc. 3rd Workshop on Compilers for Parallel Computers*, pages 347–358. Technical report ACPC/TR 92-8 of the Austrian Center of Parallel Computation, July 1992.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):667–677, August 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[HPF93] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Scientific Programming*, 2, 1993.

[HQ91] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming in 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 Computing*, 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 Computing*, 16:233–249, 1992.

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

[Ive62] Kenneth E. Iverson. *A Programming Language.* New York: Wiley, 1962.

[Juv92a] S. Juvaste. An Implementation of the Programming Language pm2 for PRAM. Technical Report A-1992-1, Dept. of Computer Science, University of Joensuu, Finland, 1992. `ftp://cs.joensuu.fi`.

[Juv92b] S. Juvaste. The Programming Language pm2 for PRAM. Technical Report B-1992-1, Dept. of Computer Science, University of Joensuu, Finland, 1992. `ftp://cs.joensuu.fi`.

[Käp92] Karin Käppner. Analysen zur Übersetzung von FORK, Teil 1. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1992.

[Keß96] Christoph W. Keßler. Pattern-driven Automatic Parallelization. *Scientific Programming*, 5:251–274, 1996.

[KMR90] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. In *ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 177–186, 1990.

[KS95a] Christoph W. Keßler and Helmut Seidl. Fork95 Language and Compiler for the SB-PRAM. In E.L. Zapata, editor, *Proc. 5th Workshop on Compilers for Parallel Computers*, pages 409–421. Dept. of Computer Architecture, University of Malaga, Spain. Report No. UMA-DAC-95/09, June 28–30 1995. `http://www.informatik.uni-trier.de/~kessler/fork95.html`.

[KS95b] Christoph W. Keßler and Helmut Seidl. Making FORK Practical. Technical Report 01/95, Sonderforschungsbereich 124 VLSI–Entwurfsmethoden und Parallelität, Universität Saarbrücken, 1995.

[KS97a] Christoph W. Keßler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of Parallel Programming*, 25(1):17–50, Feb. 1997.

[KS97b] Christoph W. Keßler and Helmut Seidl. Language Support for Synchronous Parallel Critical Sections. In *Proc. APDC'97 Int. Conf. on Advances in Parallel and Distributed Computing, Shanghai, China.* Los Alamitos: IEEE Computer Society Press, March 1997.

[KT97] Christoph W. Keßler and Jesper L. Träff. Language and Library Support for Practical PRAM Programming. In *5th EUROMICRO Workshop on Parallel and Distributed Processing, London.* Los Alamitos: IEEE Computer Society Press, Jan. 1997.

[Lec95] David Lecomber. An object-oriented programming model for BSP computations. In *Proc. PPECC Workshop on Parallel and Distributed Computing*, March 1995.

[Lil93] Joachim Lillig. Ein Compiler für die Programmiersprache FORK. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1993.

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

[LSRG95] C. León, F. Sande, C. Rodríguez, and F. García. A PRAM Oriented Language. In *EUROMICRO PDP'95 Workshop on Parallel and Distributed Processing*, pages 182–191. Los Alamitos: IEEE Computer Society Press, Jan. 1995.

[MR90] Michael Metcalf and R. Reid. *Fortran90 Explained.* Oxford University Press, 1990.

[MS94] Tom MacDonald and Zdenek Sekera. The Cray Research MPP Fortran Programming Model. In K.M. Decker and R.M. Rehmann, editors, *IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland.*, pages 1–12. Basel: Birkhäuser, April 1994.

[PM94] Michael Philippsen and Markus U. Mock. Data and Process Alignment in Modula-2*. In *C.W. Keßler (Ed.): Automatic Parallelization — New Approaches to Code Generation, Data Distribution and Performance Prediction*, pages 177–191. Wiesbaden: Vieweg, 1994.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[PS95] F.P. Preparata and M.I. Shamos. *Computational Geometry — an Introduction.* Springer–Verlag, New York, 1995.

[PT91] Michael Philippsen and Walter F. Tichy. Compiling for Massively Parallel Machines. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 92–111. Springer Workshops in Computing Series, 1991.

[PTVF92] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C — The Art of Scientific Computing, second edition.* Cambridge University Press, 1992.

[QH90] Michael J. Quinn and Phil Hatcher. Data-Parallel Programming on Multicomputers. *IEEE Software*, pages 124–131, September 1990.

[Qui94] Michael J. Quinn. *Parallel Computing, Theory and Practice, 2nd edition.* McGraw-Hill, 1994.

[Röh96] Jochen Röhrig. Implementierung der P4-Laufzeitbibliothek auf der SB-PRAM. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1996.

[RS87] J. Rose and G. Steele. C*: an Extended C Language for Data Parallel Programming. Technical Report PL87-5, Thinking Machines Inc., Cambridge, MA, 1987.

[RS92] Gudula Rünger and Kurt Sieber. A Trace–Based Denotational Semantics for the `PRAM`–Language `FORK`. Technical Report C1-1/92, Sonderforschungsbereich 124 VLSI–Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken (Germany), 1992.

[Sch91] Arno Schmitt. A Formal Semantics of `FORK`. Technical Report C1-11/91, Sonderforschungsbereich 124 VLSI–Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken (Germany), 1991.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Runtime Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[SDB93] A. Skjellum, N. Dorr, and P. Bangalore. Writing Libraries in MPI. In *Scalable Parallel Libraries Conference, IEEE CS Press*, 1993.

[SE95] David B. Skillicorn and Domenico Talia (Eds.). *Programming Languages for Parallel Processing.* Los Alamitos: IEEE Computer Society Press, 1995.

[Sei93] H. Seidl. Equality of Instances of Variables in `FORK`. Technical Report C1-6/93, Sonderforschungsbereich 124 VLSI–Entwurfsmethoden und Parallelität, Universität des Saarlandes, Saarbrücken (Germany), 1993.

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

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[TPH92] Walter F. Tichy, Michael Philippsen, and Phil Hatcher. A Critique of the Programming Language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), Aug. 1990.

[Wel92] Markus Welter. Analysen zur Übersetzung von FORK, Teil 2. Diploma thesis, Universität des Saarlandes, Saarbrücken (Germany), 1992.

[Wil88] James M. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch–and–Add.* PhD thesis, New York University, 1988.

[ZYC96] Xiaodong Zhang, Yong Yan, and Robert Castaneda. Evaluating and Designing Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors. *IEEE Parallel & Distributed Technology*, 4(1):25–42, 1996.

Online software and documentation

The Fork95 compiler including all sources is available by anonymous ftp from

`ftp.informatik.uni-trier.de` in directory `/pub/users/Kessler`.

This distribution also contains documentation, example programs and a preliminary distribution of the SB-PRAM system software tools including assembler, linker, loader and simulator.

Fork95 is also accessible on the World-Wide-Web via the URL

`http://www.informatik.uni-trier.de/~kessler/fork95`

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

Fork95 function (pointer) type qualifiers			
name	meaning	remark	example for usage
`sync`	declare (pointer to) synchronous function		`sync int foo( sh int k ) {...}`
`async`	declare (pointer to) asynchronous function	default	`sh void (*task[Pmax])(void);`
Fork95 storage class qualifiers			
`sh`	declare a shared variable		`sh int k=1, *pk, a[32];`
`pr`	declare a private variable	default	`pr int i,*pi, b[100];`

Fork95 language constructs: statements				
name	meaning	example for usage	mode	body
`barrier`	group–local barrier synchronization	`barrier;`	async.	—
`farm`	enter asynchronous mode	`farm puts("Hello");`	synchr.	async.
`fork`	split current group in subgroups	`fork(3;@=$%3;$=$/3)...`	synchr.	synchr.
`join`	join a new root group	`join(wait();0;1000)...`	async.	synchr.
`retry`	jump back to the `join`	`else{ other(); retry;}`	async.	—
`seq`	one processor enters asynchronous mode	`seq puts("Hello");`	synchr.	async.
`start`	all processors enter synchronous mode	`start { a[$]=$; }`	async.	synchr.

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

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

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

Table 1.2 Fork95 at a glance. `uint` is an abbreviation for `unsigned int`.

Other important Fork95 library routines				
name	mode	type	parameters	meaning
`reltoabs`	`sync`	`int`	`void`	get absolute address
`getct`	`async`	`uint`	`void`	get clock cycle counter
`prS`	`async`	`void`	`char *`	print string to stdout

The locks (see 1.3.2.1 and 1.3.2.3)				
name	mode	type	parameters	meaning
`simple_lock_init`	`async`	`void`	`simple_lock`	initialize a simple lock
`simple_lockup`	`async`	`void`	`simple_lock *`	wait and lock a simple lock
`simple_unlock`	`async`	`void`	`simple_lock *`	unlock a simple lock
`fair_lock_init`	`async`	`void`	`fair_lock *`	initialize a simple lock
`fair_lockup`	`async`	`void`	`fair_lock *`	wait and lock a fair lock
`fair_unlock`	`async`	`void`	`fair_lock *`	unlock a fair lock

The `pqueue` **parallel queue data structure** (see 1.3)				
name	mode	type	parameters	meaning
`init_queue`	`async`	`void`	`pqueue **`	initialize a parallel queue
`put`	`async`	`void`	`pqueue *, element *`	append element to queue
`get`	`async`	`element *`	`pqueue *`	get first queue element
`tir`	`async`	`int`	`int *, int, int`	test–increment–retest
`tdr`	`async`	`int`	`int *, int, int`	test–decrement–retest

Graphics output routines (see `graphic.h`)				
name	mode	type	parameters	meaning
`init_pict`	`sync`	`pict *`	`int` H`, int` B	allocate pixel array $H \times B$
`switch_to_pict`	`sync`	`void`	`pict *`	switch to other pixel array
`clear_pixels`	`sync`	`void`	`int` c	set all pixels to color c
`set_pixel`	`async`	`void`	`int` x`, int` y`,` `int` c	assign color value c to a pixel (x, y)
`get_pixel`	`async`	`int`	`int` x`, int` y	get color value of a pixel

`line`	`sync`	`void`	`int` x_1`, int` y_1`,`
			`int` x_2`, int` y_2`,`
			`int` c`, int` w
				draw a line of width w
				and color c in parallel

`seq_line`	`async`	`void`	`int` x_1`, int` y_1`,`
			`int` x_2`, int` y_2`,`
			`int` c`, int` w
				draw a line of width w
				and color c in sequential

`write_pixmap`	`sync`	`void`	`const char *`*name*	write current picture to X- Pixmap file *name*`.xpm`

Table 1.3 Some important Fork95–specific library routines. `uint` is an abbreviation for `unsigned int`.