Language Support for Synchronous Parallel Critical Sections

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

Abstract

We introduce a new parallel programming paradigm, namely synchronous parallel critical sections. *Such parallel critical sections must be seen in the context of switching between synchronous and asynchronous modes of computation.* Thread farming *allows to generate bunches of threads to solve independent subproblems asynchronously and in parallel. Opposed to that, synchronous parallel critical sections allow to organize bunches of asynchronous parallel threads to execute certain tasks jointly and synchronously. We show how the PRAM language Fork95 can be extended by a construct* `join` *supporting parallel critical sections. We explain its semantics and implementation, and discuss possible applications.*

1. Introduction

In a parallel environment, critical sections (for a survey, see e.g. [4]) are segments of code accessing data which are visible to more than one parallel thread. Their implementation is one of the key problems, e.g., of global resource management or consistency in parallel databases. Classically, semaphores are used to avoid more than one thread at a time to execute the critical section. The remaining threads also aiming to enter the section are kept in a (priority) queue where they wait until the presently executing thread has left the critical section. In the sequel, we will call such a mechanism *sequential critical section*.

The performance of sequential critical sections is acceptable as long as the critical section is short and the frequency by which threads demand to enter is low. In a massively parallel surroundings, however, with several thousand processors a sequential critical section can very soon become the bottleneck for the overall performance of the system.

One solution to this problem is given by *synchronous parallel critical sections*. In a synchronous parallel critical section (PCS for short) several threads are allowed to enter simultaneously. Inside the critical section and as soon as the entering phase has been finished they jointly execute a synchronous parallel algorithm. Having terminated, the threads return to their original mode of computation, the critical section gets unlocked, and a new bunch of threads is allowed to enter.

To make this idea work, several questions have to be answered: What are possible conditions under which threads are allowed to enter? When should the enter procedure be terminated? What happens with threads not allowed to enter? Should they be blocked? Should they be allowed to continue?

In order to investigate possible answers to these questions and to study its implications on the semantics and efficiency of programs we extend the parallel language Fork95 by a new language construct `join`.

Fork95 is an experimental parallel programming language which has been designed to write elegant and efficient programs for synchronous shared memory MIMD machines (also known as PRAM's). PRAM's are particularly well suited for the implementation of irregular numerical computations, non–numerical algorithms, and database applications. One such machine currently under construction at Saarbrücken University is the SB-PRAM [1, 2]. The SB-PRAM is a lock-step-synchronous, massively parallel multiprocessor with up to 4096 RISC-style processing elements and with a (from the programmer's view) physically shared memory of up to 2GByte with uniform memory access time.

In Fork95, processors are organized in *groups*. Groups may be temporarily subdivided into subgroups; this may also be applied recursively. Thus, at any point of program execution, the groups form a tree–like hierarchy, with the group consisting of all started processors as root, and the *leaf groups* being currently active. A synchronous mode of computation guarantees exact synchronicity for the leaf groups. Furthermore, Fork95 provides the possibility for *thread farming*, i.e., for locally switching from synchronous mode into an asynchronous mode of computation where desired by the programmer. This facility is crucial when tuning programs for efficiency. In some respect, the new construct `join` turns out to be complementary of farming.

The rest of the paper is organized as follows. The next section gives a short overview over the language Fork95 as it is. Section 3 presents the `join` construct together with its semantics. Section 4 explains how this construct can be implemented efficiently. Section 5 contains some examples together with explications of areas of possible applications. Section 6 contains measurements. Section 7 discusses further generalizations and concludes.

2. A short introduction to Fork95

For reasons of selfcontainment let us briefly recall the basic concepts of the programming language Fork95. A more detailed description can be found e.g. in [10] or [8]. Fork95 is a redesign of the PRAM language FORK [6]. In order to enable reuse of existing C code, Fork95 is based on ANSI C [3]. Additionally, it offers constructs that manage shared and private address subspaces, control synchronicity, and hierarchically divide groups of processors into subgroups. Fork95 makes the instruction–level synchronicity of the underlying hardware available to the programmer. It further enables direct access to hardware-supplied multiprefix operations.

2.1. Shared and private variables

The entire shared memory of the PRAM is partitioned into private address subspaces (one for each processor) and a shared address subspace. Accordingly, variables are classified as either private (`pr`, this is the default) or shared (`sh`), where "shared" always relates to the processor group that defined that variable.

There is a special private variable `$` meant to hold the current *group–relative processor ID* which is initially set to the physical processor ID `__PROC_NR__`. The special shared variable `@` is meant to hold the current *leaf group ID*. `@` and `$` are automatically saved and restored when subgroups are entered resp. left. However, the user is responsible to assign reasonable values to them (e.g., at the `fork()` instruction).

We consider an expression to be private if it is not guaranteed to evaluate to the same value on each processor, e.g. if a private variable occurs in it.

Fork95 inherits the concurrent write conflict resolution scheme from the target hardware. On the SB-PRAM, if several processors write the same (shared) memory location in the same cycle, the processor with maximal `__PROC_NR__` will win and write its value (PRIORITY–CRCW-PRAM). However, as several other write conflict resolution schemes (like ARBITRARY) are also used in theory, meaningful Fork95 programs should not be dependent on such specific conflict resolution schemes; there are better language elements (multiprefix instructions, see below) that cover practically relevant applications for concurrent write.

2.2. Synchronous and asynchronous mode

Fork95 offers two modes of program execution that are statically associated with program regions. In *synchronous mode*, Fork95 maintains at each point of program execution the *synchronicity invariant* (SI) which says that all processors belonging to the same leaf group are operating strictly synchronously, i.e., they follow the same path of control flow and execute the same instruction at the same time. Also, all processors within the same group have access to a common shared address subspace. Thus, newly allocated "shared" objects exist once for each group allocating them. — In *asynchronous mode* the SI is not preserved, and no shared variables can be allocated. The processors start the `main()` program in asynchronous mode.

Switching to synchronous mode of execution for any `<statement>` could be simply expressed by

```
start <statement>
```

where all processors form, after exact barrier synchronization [10], one single processor group and execute `<statement>` in synchronous mode. Up to now, `start` (its name is due to historical reasons) is only supported at the top level of program control, and nesting (also dynamic) of `start` statements is forbidden; a weakness of the language that we would like to overcome. Providing a more general construct to switch from asynchronous to synchronous mode is the main issue of this paper.

Switching to asynchronous mode for a `<statement>` is done by

```
farm <statement>
```

Within `<statement>` any synchronization is suspended; at the end of `<statement>` the processors synchronize explicitly within their current leaf group.

Functions are declared to be either synchronous (`sync`) or asynchronous (`async`). *Synchronous regions*, namely `sync` functions (except from `farm` bodies) and `start` bodies, are executed in synchronous mode, while *asynchronous regions*, i.e. `async` functions (except from `start` bodies) and `farm` bodies, are executed in asynchronous mode. In order to obtain this static classification of code, from asynchronous regions only `async` functions can be called. A call to an `async` function from a synchronous region is automatically casted to a `farm` body. Using `farm` within asynchronous mode is superfluous and will be ignored.

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.

This is only possible because there is no virtual processing in Fork95. Consider e.g. the following program fragment

```
sync void foo( sh int x, a )
{ ... if ($<10)  a = x;
      else       y = a;   ...  }
```

By the semantics of synchronous mode all processors of the `else` part must read the same value of `a`. In order to guarantee this in the presence of virtual processing, it would be required to keep a group lock[1] for each shared variable. Due to the presence of pointers and weak typing in C, a lock would be required for each shared memory cell!

Pipelining through an arbitrary graph can be implemented in synchronous mode in a rather straightforward manner, as we have shown in [8]. This both covers pipelining through multidimensional arrays as used by systolic algorithms and all sorts of trees as needed in certain combinatorial algorithms. We summarize that Fork95's synchronous mode saves the time and space overhead for locking and unlocking but incurs some overhead to maintain the SI during synchronous program execution.

The importance of being asynchronous. In asynchronous program regions there are no implicit synchronization points. Maintaining the SI requires a significant overhead also for the cases where each group consists of only one processor, or when the SI is not required for consistency because of the absence of data dependencies. Hence marking such regions as asynchronous can lead to substantial savings. In our experimental work, we found considerate usage of the `farm` statement and asynchronous functions to pay off in significant performance improvements (execution time was reduced by up to 50 percent).

2.3. Hierarchical group concept

In synchronous mode, the synchronicity invariant permits to relax synchronicity in two ways: either by using `farm` as described above (and thus leaving synchronous mode), or by splitting a group into subgroups and maintaining the invariant only within each of the subgroups. This has to be taken into consideration if control flow may diverge due to private branching conditions. Shared `if` or loop conditions do not affect the synchronicity, as the branch taken is the same for all processors executing it. At an `if` statement, a private condition causes the current processor group to be split into two subgroups: the processors for which the condition evaluates to nonzero form the first subgroup and execute the `then` part while the remaining processors execute the `else` part. The available shared address space of the parent group is subdivided among the new subgroups. When both subgroups have finished the execution of their branch, they are released, and the parent group is reactivated by exact synchronization of all its processors. — A similar subgroup construction is required also at loops with private exit condition. All processors that will execute the first iteration of the loop enter the subgroup and stay therein as long as they iterate. Processors that leave the loop body are just waiting at the end of the loop for the last processors of their (parent) group.

Subgroup construction can also be done explicitly. Executing

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

means the following: First, the shared expression e_1 is evaluated to the number of subgroups to be created. Then the current leaf group is split into that many subgroups. Evaluating e_2, every processor determines the number of the newly created leaf group it will be a member of. Finally, by evaluating e_3, each processor may renumber its group–local processor ID within the new leaf group. Note that empty subgroups (with no processors) are possible; an empty subgroup's work is immediately finished, though. Continuing, we partition the parent group's shared memory subspace into that many equally–sized slices and assign each of them to one subgroup, such that each subgroup has its own shared memory space. Now, each subgroup executes `<statement>`; the processors within each subgroup work synchronously, but different subgroups can choose different control flow paths. After `<statement>` has been completed, the processors of all subgroups are synchronized, the subgroups and their shared memory subspaces are released, and the parent group is reactivated as the current leaf group.

Clearly, if a leaf group consists of only one processor, the effect is the same as working in asynchronous mode. However, the latter avoids the expensive time penalty of continued subgroup formation and throttling of computation by continued shared memory space fragmentation.

2.4. Pointers and heaps

The usage of pointers in Fork95 is as flexible as in C, since all private address subspaces have been embedded into the global shared memory of the SB-PRAM. Thus, shared pointer variables may point to private objects, and vice versa. The programmer is responsible for such assignments making sense.

Up to now, Fork95 supplies two kinds of heaps: one automatic shared heap for each group, and one private heap for each processor. While space on the private

[1] This would partially sequentialize the `then` with the `else` group w.r.t. accessing variable `a`.

heaps can be allocated by the asynchronous `malloc` function known from C, space on the automatic shared heap is allocated using the synchronous `shalloc` function. The live range of objects allocated by `shalloc` is limited to the live range of the group by which that `shalloc` was executed. Thus, such objects are automatically removed if the group allocating them is released. Supplying a third variant for a global, "permanent" shared heap is addressed later in this paper.

Pointers to functions are also supported. For efficiency reasons, 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.

2.5. Multiprefix instructions

The SB-PRAM offers built-in multiprefix instructions which allow the computation of multiprefix integer addition, maximization, `and` and `or` for up to 4096 processors within 2 CPU cycles. Fork95 makes these powerful instructions available as Fork95 operators (atomic expression operators, not functions). For instance, consider

```
k = mpadd( &shvar, expression );
```

All m processors executing this statement at the same clock cycle and with the same shared address `&shvar` as first parameter, let them be indexed $j = 0, ..., m-1$ in the order of increasing `__PROC_NR__`, evaluate first their private `expression` locally into a private integer value e_j. Then, processor j assigns its private integer variable `k` the value $s+e_0+e_1+\ldots+e_{j-1}$ where s denotes the previous value of the shared integer variable `shvar`. Immediately after the execution of the `mpadd` instruction, `shvar` contains, as a side effect, the global sum $s + \sum_j e_j$ of all participating expressions.

3. Semantics of `join`: The bus analogy

A useful analogy to understand the behaviour of the new `join` operator is a *bus stop*. Imagine a city with several excursion *bus lines*. 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 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. If a passenger in spe happens to be the first at the bus stop (which means that he 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 for the time of the *bus tour*. During the tour, they can allocate shared objects that are 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.

What does this mean in the context of parallel programming? The behaviour described in the bus analogy is supplied in Fork95 by a language construct

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

The passengers are the processors. Each `join` instruction installs a unique bus line with a bus stop. `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.[2] `SMalloc` is a statement executed by the bus driver to install a new shared stack and heap for the bus. Its specification is optional. The bus tour corresponds to the proper body of the join instruction and must be a synchronous `statement`.

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 the processor go back to the bus stop and try again to get the bus, similar to `continue` in loops.

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

Internally, there is for each bus a lock `gone`, i.e., a shared variable that guards access to the ticket au-

[2] 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.

tomaton, which, in turn, is a shared variable 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 that processor waits for a bus whose return he is blocking by his 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.

4. Implementation

4.1. Shared memory allocation

One possibility to allocate a new shared stack and heap for the group of processors in the bus is, of course, a call to the permanent shared malloc routine, a sequential asynchronous Fork95 function. This, however, is too simple because we just want to use the `join` construct to implement such parallel storage allocation frameworks. Another possibility would be to let the compiler allocate a statically fixed quantum of shared memory for every bus. This has the drawback that it excludes reuse of this memory for other purposes while no bus is running. Therefore, we offer a different solution: The bus driver sacrifices a memory block from his private heap for the new shared stack of the new bus. The size of this memory block could be chosen dynamically such that, for instance, half of the bus driver's currently free portion of its private heap might be nationalized[3].

4.2. Data structures

For each `join` instruction encountered in the code, the following global variables are allocated to implement the bus concept (given as C pseudocode to facilitate the description. The implementation has been coded in SB-PRAM assembler):

```
sh char *SM;
sh int gone;
sh int ticket;
```

`gone` and `ticket` are statically initialized by zero at the beginning of each Fork95 program.

[3]To compute this quantity, we provide an asynchronous library function `int pravail()` which returns the number of free private storage cells.

4.3. Translation of the join instruction

The join instruction is only admissible in asynchronous mode. `delaystmt` is optional; it should be chosen appropriately to delay the departure of the bus as desired. Variable `ticket` can be used as a parameter in `delaystmt`. `springoffcond` should evaluate to a nonzero value if a processor that entered the bus should leave immediately when starting. `SMalloc` must return a pointer to a block of memory which will install the bus's shared memory.
The instruction

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

is translated as follows (again described as Fork95 pseudocode)

```
save the old value of $;
if (!gone) { /* I am allowed to enter the bus. */
  $ = mpadd( &ticket, 1 );  /* get a ticket */
  if ($==0) {
    /* I am the bus driver and set up its      */
    SM = SMalloc;              /* shared memory */
    save the old shared group pointer, build the
    private group frame; compute the new shared
    group, stack, and heap pointer from SM;
    allocate a new shared group frame in SM block;
    delaystmt;
    gone = 1;
    wait two cycles. Now the value of the ticket
    variable remains unchanged; it is equal to
    the exact number of passengers.
    Write it into the new synchronization cell.
  }
  else {
    /* I am not the bus driver. */
    save old shared group pointer and
    build my private group frame.
    /* Then I have to wait until the driver
       has set up shared memory SM for me: */
    while (!gone) /*wait*/;
    /* now the value in SM is valid,
       I can adjust my shared pointers. */
    compute the new shared group, stack, and
    heap pointer from SM.
    Now the new synchronization cell contains
    the group size.
  }
  /* departure of the bus */
  if (springoffcond) {
    /* I decide to spring off. My membership
       in the bus group must be cancelled: */
    mpadd ( &synccell, -1 );
    remove the private group frame and
    restore the stack pointers;
    goto NO_BUS_TOUR;
  }
  else {
    /* I ride: synchronize and take off */
    call the synchronization routine
```

```
      using the new synchronization cell;
   beginsync { /* enter synchronous region */
       statement;
   };           /* leave synchronous region */
   /* Bus has returned. The processors
      inside are still synchronous. */
   ticket = 0;
   gone = 0;  /* re-open ticket automaton,
                 leave the bus */
   remove the group frames and
   restore the stack pointers;
   restore the old value of $;
} }
else {  /* The bus is not waiting. */
NO_BUS_TOUR:
   restore the old value of $;
   useful_work  /* if specified;
           a retry therein redoes the join */
} /* end of join. */
```

5. Examples

5.1. Parallel memory allocation

As an example let us consider a simple storage allocation scheme where all memory is divided into N equally sized blocks. Pointers to free blocks are maintained by a queue `avail`. 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;
```

To implement operations `void free(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 free(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.

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

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. Timings for the parallel shared heap memory allocator, using locks (second column) and `join` (third column). The measurements are taken on the SBPRAM simulator.

```
  result = NULL;

  join(for(t=0;t<2;t++); 0; malloc(100))  {
    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` by means of locks. The run time figures for a varying number of processors are given in Fig. 1. The break–even point is approximately at 128 processors. For $p > 128$ the run time for the asynchronous run time begins to grow linearly in the number of processors since the sequentialization due to the sequential critical section dominates the execution time.

5.2. Parallel critical sections

A (sequential) critical section, as known from many concurrent programming platforms and operating systems, is a block of statements that can be executed by only one processor at a time. (In our analogy, this corresponds to a bus with only one passenger.)

Our construct `join` allows the implementation of a generalization of this concept, namely *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.

We conjecture that this concept of a parallel critical section is an important step towards the usability of Fork95 for programming not only for nice "stand–alone" parallel algorithms but also of realistic and critical applications such as parallel operating system kernels.

5.3. Encapsulating existing synchronous Fork95 code

A second advantage is that now existing synchronous Fork95 code can be re-used much easier than before — it just has to be encapsulated into an appropriate `join` instruction. Since now several `join`s can be nested, it is possible to construct more complex Fork95 software packages [9].

6. Sequential versus Parallel Critical Sections

To compare the runtime behavior of sequential critical sections versus parallel critical sections, we have implemented a test suite with the following parameters: The asynchronous variant using usual sequential critical sections of length D_{crit} clock cycles (excluding overhead for locking/unlocking and the time to wait for the lock to become free), and further executing $D_{noncrit}$ cycles in a noncritical section. The synchronous variant needs no locks, executes $D_{noncrit}$ cycles in a `join` statement, excluding the overhead for the current `join` implementation. Each processor generates N queries to access the shared resource; the time T before issuing the next query is modeled using an exponentially distributed probability function.

The overall simulation is done according to the following pseudocode:

```
sh simple_lock lock;
initlock( lock );
desynchronize the processors randomly;
for (i=0; i<N; i++) {
  generate(y);
#if JOIN
  join ( do 20 cycles, 0, malloc(100) ) {
      do D_noncrit cycles
  }
#else
  lockup( &lock );
  do D_crit cycles;
  unlock( &lock );
  do D_noncrit cycles;
#endif
}
barrier;
```

where *generate*(y) is a delay routine that takes exactly $T = 6t$ steps[4] with probability $(1-2^{-y})^{(t-1)}2^{-y}$. Thus y controls the density of queries.

With $D_{noncrit} = 384$ and the overhead of the current implementation of `join` being around 150 cycles (which is due to programming in high level language and will be drastically reduced once coded in assembler), we obtain the measurements on the SB-PRAM given in Fig. 2.

Three variations of issuing queries to the parallel resp. sequential critical section have been examined: The first one ($y = 0$) assumes that a processor raises a new query immediately after the previous one has been answered. For long critical sections ($D_{crit} = 50$ cycles) the break–even point is between 8 and 16 processors; for short critical sections ($D_{crit} = 15$ cycles) it is at 32 processors. $y = 4$ models exponentially distributed query generation with an expected delay of around 200 cycles from the completion of the previous query. The break–even point for long critical sections is near 32 processors and for short critical sections at 128 processors. $y = 8$ models exponentially distributed query generation with an expected delay of around 3000 cycles from the completion of the previous query. The break–even points are here at 512 and 2048 processors, respectively.

Generally the `join` construct is profitable for a high density of queries to the shared resource which is influenced by the number of processors as well as the average density of queries on each processor. It is interesting to observe that at high numbers of processors, with the density of queries being very high, the query generation time becomes meaningless because the sequentialization by the critical section (and thus, D_{crit}) dominates the execution time.

[7] analyzes the average time behaviour of sequential and parallel critical sections using stochastic models based on the theory of queuing systems in discrete time. This work confirmed our empirical results and provided an exact prediction of the break–even point for given model parameters.

In our experiments we assumed that query generation follows some regular pattern; the modeling produced expected values for query generation time that

[4] The numbers that occur here are multiples of 6 because 6 is the length of an empty `for` loop iteration.

y	$y=0$			$y=4$			$y=8$		
D_{crit}		50	15		50	15		50	15
procs	JOIN	asynchronous		JOIN	asynchronous		JOIN	asynchronous	
1	9507	7865	7613	14070	12410	12158	79965	78323	78071
8	9533	10553	8537	17741	14795	14371	81365	78323	78071
16	11057	13635	9593	19051	15855	15603	110929	108123	107871
32	11057	19769	11705	23623	21043	20791	154789	152111	151775
64	11057	32067	15951	23499	32331	20791	245539	243903	243651
128	11057	56643	24403	26725	56851	26871	262023	260511	260191
256	11057	105795	41301	28263	105999	41499	271863	270349	270029
512	11057	204089	75065	28273	204301	75289	319501	316609	316289
1024	11057	400697	142681	32873	400909	142873	358467	401037	356577
2048	11057	793913	277817	32851	794125	278041	402481	794243	400591
4096	12605	1580531	548191	37601	1580557	548377	402481	1580673	548547

Figure 2. Measurements for the test scenario on the SBPRAM (the simulator provides exact timings), given in clock cycles.

were equally resp. exponentially distributed. In real parallel programs, this may not generally be the case. Instead, it is more likely that, considering a longer trace of program execution, there will occur bursts of more intensive query generation, separated by time intervals with rather sparse query generation. We believe that this scenario will even better profit from the `join` statement because it allows to collect also the large amount of queries in a burst interval and process them immediately as a whole, while the asynchronous variant would result in a long queue of processors waiting to enter the critical section.

7. Conclusion

We presented a new language construct for synchronous parallel programming languages, the `join`. Complementary to farming, `join` supports the implementation of synchronous parallel critical sections, thus avoiding the bottleneck of sequential critical sections in a massively parallel programming environment.

An interesting direction of further research is the extension of Fork95 with object–oriented features. Object–oriented programming allows for a different and surprisingly elegant view on busses: An individual bus line, along with its properties common for all processors, could be interpreted as a special object equipped with certain data and methods. Especially, arrays of bus lines could be constructed each of which may run independently of the others. The idea of encapsulating access to shared data structures into some kind of object has already been proposed for sequential critical sections in OPUS where they are called *shared data abstractions* [5].

The Fork95 compiler is available by FTP via `ftp.informatik.uni-trier.de/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. The Fork95 compiler and documentation is also available on the WWW, `http://www.informatik.uni-trier.de/ ~kessler/fork95.html`.

References

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

[2] F. Abolhassan, J. Keller, and W. Paul. On the cost–effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. on Parallel and Distr. Processing*, pages 2–9. IEEE, Dec. 1991.

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

[4] H. Bal. *Programming Distributed Systems.* Prentice Hall, 1990.

[5] B. Chapman, P. Mehrotra, J. V. Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. In *Proc. CONPAR 94, Linz, Austria*, Sept. 1994.

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

[7] J. Hofmann, N. Müller, and K. Natarajan. Parallel versus Sequential Task-Processing: a New Performance Model in Discrete Time. to appear as a technical report of Dept. of Mathematics and Computer Science, University of Trier, Germany. Submitted, 1996.

[8] C. Keßler and H. Seidl. Integrating Synchronous and Asynchronous Paradigms: The Fork95 Parallel Programming Language. In W. Giloi, S. Jähnichen, and B. Shriver, eds., *Proc. 2nd Int. Conf. on Massively Parallel Programming Models*, pages 134–141. IEEE CS Press, Oct. 1995. See also: Techn. report 95-05, FB IV Informatik, Univ. Trier.

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

[10] C. W. Keßler and H. Seidl. Fork95 Language and Compiler for the SB-PRAM. In *Proc. 5th Workshop on Compilers for Parallel Computers*, pages 409–421. Dept. of Computer Architecture, University of Malaga, Spain. Report UMA-DAC-95/09, June 28–30 1995.