Implementing a Parallel List on the SB-PRAM

Andreas Paul, Jochen Röhrig
University of Saarland — Computer Science Department
PO Box 151150, 66041 Saarbrücken, Germany
{ap,roehrig}@cs.uni-sb.de

Abstract

We give a description of a C++ implementation of a dynamic parallel list developed for the SB-PRAM, a massively parallel scalable shared memory computer. We show that access time on the elements stored in the parallel list is comparable with that of a sequential list. The implementation can easily be ported to other shared memory platforms supporting fast locking mechanisms and parallel prefix operations.

1. Introduction

Being one of the most often used data structures on sequential computers, the linear list does not have a definite counterpart on parallel computers. The reason for this situation lies on the one hand in the large variety of architectures and on the other hand in the different possibilities of the semantics of its access functions. Among the questions to answer are: *"How often shall a stored element be read during an iteration over the data structure?"*, *"How can an order be defined on the stored elements?"*, *"Is full control over the position of the elements needed?"*.

Since the demands on the data structure have a great influence on its memory consumption and on the time needed to access its elements we must state exactly, how universal the data structure has to be. During the parallelization of a library for solving partial differential equations [9], we had to create a "parallel list" for shared memory computers with the following features (let p be the number of processors accessing the list in parallel):

- The capacity of the parallel list should not be fixed.
- The fixed memory overhead should be at most linear in p, and the memory overhead for each stored element should be as small as possible.
- The access time (especially for read operations) should be independent of p. Furthermore, it should be comparable to the time needed for accessing elements in a sequential list.
- For reasons of predictability (which is very useful for debugging) there should be a sequential order of the list elements.
- Controlling the position of the elements in the list is not necessary. We only need an iteration function *read_next*() for visiting the elements of the list.
- Apart from visiting elements we also need to add new elements to the list during an iteration. Both actions should be performable independently and in parallel. Two functions should provide two different ways of adding an element e to the list: *insert*(e) and *append*(e) place e at the "head" and "tail" of the list respectively. We call appended elements and elements residing in the list at the beginning of an iteration *active* and inserted elements *sleeping*.
- During an iteration, each active element is visited *exactly once*, sleeping elements are *not* visited. After an iteration all sleeping elements become active.

Among the implementations of parallel data structures, we found no structure which met all of these demands. However, we found two promising approaches:

First, the Visit List of the NYU-Ultracomputer project [3]. Unfortunately, this data structure only guarantees, that during an iteration, each element is visited *at least once*.

Second, the parallel queue implementation for the SB-PRAM [10] which was derived from the one of the NYU-Ultracomputer project [6, 13]. Meeting a big part of our demands, this data structure would nevertheless need some extensions to support all of the requirements stated above.

Since we intended to use the parallel list on the SB-PRAM, we decided to take the second alternative as a base for the new data structure.

The rest of this paper is organized as follows: In section 2, we give a short overview over the relevant features of the SB-PRAM. In section 3 we derive the data structure which satisfies our requirements and describe the access functions for the parallel list. In section 4 we present an analysis of memory and time consumption. Section 5 gives a short discussion of the results of our work.

2. The SB-PRAM

The SB-PRAM is a shared memory parallel computer which emulates the PRAM model from theoretical computer science [4]. For implementation details consult [1, 2, 5, 7, 8, 11, 12]. [1]

The programmers point of view of the SB-PRAM is that of a priority CRCW PRAM [7] with n "*virtual*" processors (vP's). The priority CRCW model provides sequential semantics for the execution of parallel programs.

The vP's are simulated by multithreaded physical processors (pP's) in round robin order. Each pP can simulate up to 32 vP's, thus hiding the latency of accesses to the global memory[2]. Hashing successive memory addresses onto different memory modules avoids hot spots. Requests to the same memory cell are combined thus reducing the traffic between processors and memory and avoiding serialization of requests in the network switches and at the memory modules. Latency hiding by simulating vP's, hashing memory addresses and combining memory requests lead to a uniform fast memory access time.

The final version of the SB-PRAM prototype will have 64 pP's corresponding to 2048 vP's with an instruction issue frequency of 7 MHz per pP.

If in the following we are talking about "processors" we always refer to virtual processors.

2.1. Global memory access

In addition to the standard operations LOAD and STORE, the SB-PRAM hardware provides the parallel operations MP and SYNC. The semantics of MP (multiprefix) is defined as follows: Consider an associative binary operator $\circ$, a memory cell a holding the value v and t processors $p_1, \ldots, p_t$ (ordered in decreasing priority) holding values $v_1, \ldots, v_t$. Let the processors execute operations $r_i = \mathtt{mp}\circ(a, v_i)$, $1 \leq i \leq t$, in parallel. Then the result r_i for processor i after the termination of the t operations equals $r_i = v \circ v_1 \circ \ldots \circ v_{i-1}$ and the value of a is defined as $a = v \circ v_1 \circ \ldots \circ v_t$. The SB-PRAM supports the operations $+$, max (on signed integers) and binary and and or which are computed "on the fly" by ALUs in the network nodes. Thus an MP-operation does not take more time than a LOAD-operation. SYNC has the same semantics like MP but does not deliver a result to the processors. The SYNC- and MP-operations are accessible via C++ inline functions

word mp<`Op`>(**void*** a, **word** v)   and
void sync<`Op`>(**void*** a, **word** v)

with `<Op>`$\in\{$**add**, **max**, **or**, **and**$\}$.

[1]Further information is available at the URL `http://www-wjp.cs.uni-sb.de/projects/sbpram`.

[2]Additionally, LOAD operations impose one delay slot.

2.2. Synchronization primitives

For our parallel list we need three of the synchronization primitives described in [10]: the *fair lock*, the *barrier* and the *tdr(test-decrement-retest)*-operation

A *fair lock* l is used to protect a critical code section s in which no more than one process may be active at a given time. Entering, executing and leaving s is done by executing a "**fair_lock**(l); s; **fair_unlock**(l)"-sequence of statements. As soon as a process P has terminated its **fair_lock**(l)-operation it is guaranteed that no other process will execute s until P has terminated its **fair_unlock**(l)-operation, i.e. P is granted exclusive access to s. The term *fair* arises from the fact that the lock mechanism grants access to s in the same order in which the waiting processes have started their **fair_lock**(l)-operations and thus ensures that no process can starve within a **fair_lock**(l)-operation.

A *barrier* b is used to guarantee that all processes of a process group have terminated a part of a parallel program before they go on executing the next part of the program. If p processes are executing a "s_1; **barrier**(b, p); s_2"-sequence of statements none of the processes may begin executing s_2 before all p processes have terminated the execution of s_1 and have started their **barrier**(b, p)-operation.

An execution t of a **tdr**(c, δ, b)-operation returns **true**, if in t the shared counter c could be decremented by δ without violating the condition $c \geq b$ after the decrementation, otherwise t returns **false,** and c is not decremented. The **tdr**()-operation was introduced in [6].

A serialization free implementation of the barrier and the **tdr**()-operation on the SB-PRAM is described in [10].

3. The parallel list

We want to support the following operations on the parallel list:

***init*():** Initialization of the parallel list.

***read_next*():** Iteration over the elements of the list, where each active element is read exactly once.

***append*():** Dynamic addition of an element, which will be read in the current iteration.

***insert*():** Dynamic addition of an element, which will not be read in the current iteration.

***reinit*():** Separation of the iterations on the list and activation of sleeping elements.

We considered two alternatives for implementing the parallel list:

First, simulating the parallel list with two parallel queues [10], a working queue and an auxiliary queue. The *read_next*()-function, for example, could be realized by popping the first element from the working queue and appending it to the auxiliary queue. After an iteration the two queues would be swapped.

Second, extending the parallel queue of [10] to a parallel list. We decided to choose this strategy since it promises to provide faster access functions than the first one. As we will see, for example, in *read_next*() we can leave the read element in the list instead of popping it from a queue and appending it to another.

3.1. The general principle

We derive our parallel list from a sequential list that meets our requirements.

3.1.1. The sequential case. A sequential list (figure 1) consists of two sublists: a *sleeplist* on the left hand side

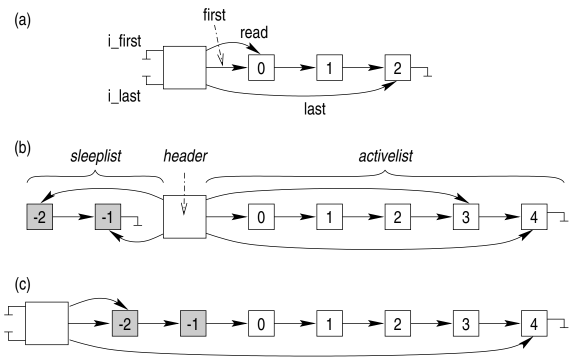

Figure 1. Sequential list

which holds all sleeping elements and an *activelist* on the right hand side which holds all active elements

Situation (a) shows a list with three elements at the beginning of an iteration I. The $read$-pointer points to the first element to be read and $first$ and $last$ point to the first and to the last element of the activelist respectively. The sleeplist is empty (i_first and i_last are NULL). Situation (b) shows the state of the list during I after two *append*()-, three *read_next*()- and two *insert*()-operations. Note that the inserted elements are not read during I. In situation (c) *reinit*() was invoked after the termination of I. The two inserted elements were shifted to the beginning of the activelist and all other active elements were shifted to the right by two positions. The newly inserted elements will be the first elements to be read in the next iteration.

The sequential list of figure 1 is not usable in a parallel context: accesses to the list involve changing the values of pointers, which may lead to inconsistencies if done by several processes in parallel. The sequential list can be "parallelized" by protecting the critical section in which pointer values are changed by means of a lock. However, serialization might lead to a considerable loss in performance, especially in a massively parallel context [6].

3.1.2. The parallel case. The basic idea of our parallel list implementation is to avoid serialization by splitting up the sleeplist and the activelist of the sequential list into k sleeplists and k activelists as shown in figure 2 (here: $k = 4$). The sublists[3] are accessed via an array of list headers. A header of a sublist S holds the same pointers as in the sequential case. If multiple processes try to access the same activelist or sleeplist in parallel, they still must be serialized. However, if k is large enough and if the processes are distributed smartly to the sublists, this situation is very unlikely to occur.

Process Distribution. Distributing the processes to the sublists is done by means of three global counters gr_cnt, ga_cnt and gi_cnt. At the beginning of a *read_next*()-operation counter gr_cnt is incremented by executing $x =$ **mpadd**$(gr\_cnt, 1)$; x mod k corresponds to the index of the activelist from which the next element is to be read. Counter ga_cnt is used analogously in *append*().

Counter gi_cnt is decremented at the beginning of an *insert*()-operation by executing $x =$ **mpadd**$(gi\_cnt, -1)$ where x mod k determines the sleeplist into which the element is to be inserted.

We require k to be a power of two which speeds up the calculation of the modulus values and frees us from taking care of possible overflows of the global counters.

Figure 2 shows the states of a parallel list with 4 sublists that correspond to the states of the sequential list of figure 1. Using the global counters as described above leads to a round robin distribution of the elements to the sublists. Situation (a) shows the parallel list at the beginning of an iteration I. The active elements are distributed on activelists 0, 1 and 2; activelist 3 is empty, and all sleeplists are empty since we require all elements to be active at the beginning of an iteration. The local $first$-, $read$- and $last$-pointers of activelists 0, 1 and 2 point to the only element e in the sublist signifying that for the considered sublist e is the first active element, that e will be the first element to be read and that the next active element to be appended will be placed behind e respectively. Each element e is numbered by the value x that was returned by the $x =$ **mpadd**$(ga\_cnt, 1)$-statement at the beginning of the *append*()-operation during which e was added to its sublist (in the following we say that e was appended "at the ga_cnt-value x"; analogously for gr_cnt and gi_cnt.). Since for *read_next*() we use the same method for distributing the processes to the sublists, this numbering also defines the order in which the elements will be read. This can be stated in an invariant:

[3]If we're talking about "sublist" in the following we always refer to one activelist and its corresponding sleeplist.

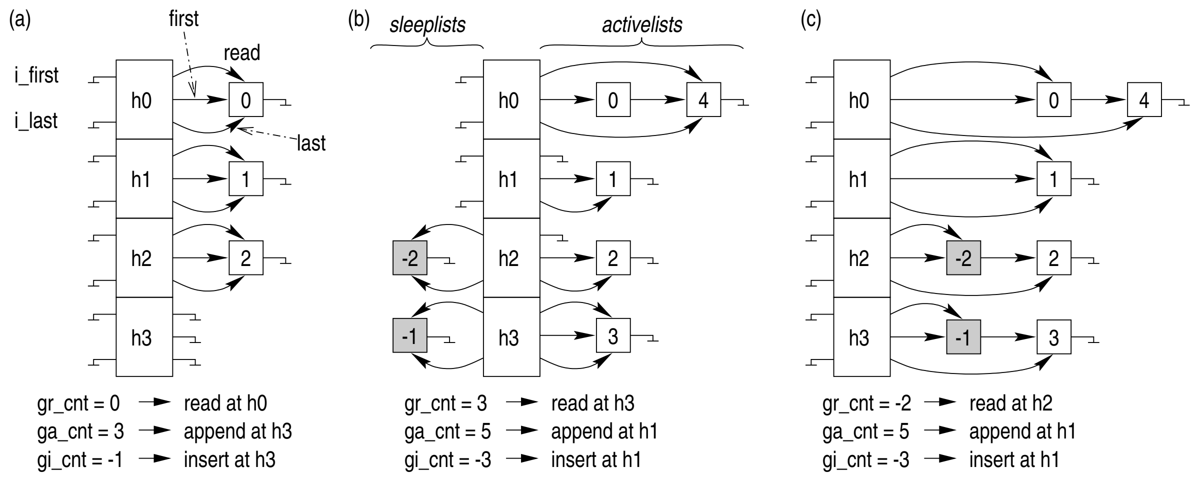
(a)
first
read
i_first
i_last
last
h0
h1
h2
h3
0
1
2
gr_cnt = 0 → read at h0
ga_cnt = 3 → append at h3
gi_cnt = -1 → insert at h3
(b)
sleeplists
activelists
-2
-1
4
3
gr_cnt = 3 → read at h3
ga_cnt = 5 → append at h1
gi_cnt = -3 → insert at h1
(c)
gr_cnt = -2 → read at h2
ga_cnt = 5 → append at h1
gi_cnt = -3 → insert at h1

Figure 2. Parallel list

(I1) If an element e is appended to the parallel list at a ga_cnt-value x then e must be read at the same value x of gr_cnt.

Figure 2(b) corresponds to figure 1(b) and shows the parallel list during I after two *append*()'s, three *read_next*()'s and two *insert*()'s. The counters were changed accordingly: ga_cnt was incremented by 2, gr_cnt was incremented by 3, and gi_cnt was decremented by 2. The new elements are numbered according to the above stated principle. The inserted (sleeping) elements have numbers smaller than zero corresponding to the value of gi_cnt at the time of their insertion and are distributed on the sleeplists round robin but in the opposite direction of the appended elements. This reflects our requirement that elements added to the list by *append*()'s are put at the end of the parallel list whereas elements added by *insert*()'s are put at its beginning.

Figure 2(c) shows the list of figure 2(b) at the end of the *reinit*()-operation which was executed after the termination of I. As in the sequential case the newly inserted elements are moved to the beginning of the corresponding activelists, thus becoming the active elements which will be read first during the next iteration. Counters ga_cnt and gi_cnt remain unchanged whereas gr_cnt is set to $gi\_cnt + 1$ which corresponds to the number of the new first element of the list, i.e. the element which was inserted most recently. This ensures that the following invariant concerning the formerly sleeping elements is maintained during the next iteration:

(I2) If an element e is inserted into the parallel list at a gi_cnt-value x then e must be read at the same value x of gr_cnt.

Local Access Protocol. Using the global counters as described above ensures maintenance of **(I1)** and **(I2)** as long as there is no contention on the sublists. If there is contention (which is very probable to happen if k is smaller than the maximum number of processes accessing the parallel list concurrently) it is straightforward to construct a scenario which leads to a violation of **(I1)** and **(I2)** [10]. To avoid such situations we introduce a local counter la_cnt for each activelist S_a which holds the number of the next element to be appended to S_a. An appender executes the following sequence of statements:

```
⟨1⟩  $x = $ **mpadd**$(ga\_cnt,\ 1)$;
⟨2⟩  **while**$(la\_cnt \neq x)$; *do nothing*
⟨3⟩    *append element*
⟨4⟩  **syncadd**$(la\_cnt,\ k)$;
```

Lines ⟨2⟩ and ⟨4⟩ guarantee the correct order amongst appenders that access S_a in parallel. Moreover, as a positive side effect, ⟨2⟩ and ⟨4⟩ ensure that the critical section ⟨3⟩ is always executed by at most one appender, thus acting as a lock protecting ⟨3⟩.

Additionally we need to introduce counters li_cnt and lr_cnt for each sublist which act as counterparts of la_cnt in *insert*() and *read_next*().

The three counters ensure correct ordering and serialization amongst processes of the same kind but not amongst different kinds of processes. For sleeplists this does not impose a problem since sleeplists are only accessed by one kind of processes (namely inserters). Activelists, however, may be accessed by appenders and readers in parallel. Moreover, appenders and readers accessing the same activelist manipulate pointer values which may lead to inconsistencies if done in parallel. For each activelist we therefore introduce a lock which is used to serialize concurrent accesses of appenders and readers to the activelist.

3.2. The data structure

For presenting the class definitions, we employ a pseudo-C++ notation in which, for reasons of simplification, block structures are marked by the indentation level.

Table 1 shows the structure of the objects which are used to build up a parallel list. Class `ParEle` is intended to be

```
class ParEle
 friend class ParHeader ;
 friend class ParList ;
 ⟨pe1⟩ ParEle *next;

class ParHeader
 friend class ParEle ;
 friend class ParList ;
 ⟨ph1⟩ fair_lock_t lock;
 ⟨ph2⟩ int lr_cnt, la_cnt;
 ⟨ph3⟩ ParEle *read;
 ⟨ph4⟩ ParEle *first, *last;
 ⟨ph5⟩ int li_cnt;
 ⟨ph6⟩ ParEle *i_first, *i_last;

class ParList
 friend class ParHeader ;
 friend class ParEle ;
  ⟨pl1⟩ barrier_t ri_barrier;
  ⟨pl2⟩ int size;
  ⟨pl3⟩ ParHeader *hdarray;
  ⟨pl4⟩ int r_num;
  ⟨pl5⟩ int gr_cnt, ga_cnt;
  ⟨pl6⟩ int gi_cnt;
  ⟨pl7⟩ void init(int );
  ⟨pl8⟩ void append(ParEle *);
  ⟨pl9⟩ ParEle *read_next();
 ⟨pl10⟩ void insert(ParEle *);
 ⟨pl11⟩ void reinit(int , int );
```

Table 1. Classes ParEle, ParHeader and ParList

used as a base class for classes containing “useful” data. It therefore merely consists of a *next*-pointer which is used to build up a linked list of `ParEle`-objects.

A `ParHeader`-object represents the header of a sublist. All members of class `ParHeader` were already presented in section 3.1.2.

A `ParList`-object corresponds to the global control structure of a parallel list. The barrier in ⟨pl1⟩ is used to synchronize the processes during *reinit*(). Member *size* ⟨pl2⟩ holds the number of sublists which are accessed via *hdarray* ⟨pl3⟩. During an iteration, member *r_num* ⟨pl4⟩ holds the number of elements not yet read. The global counters in ⟨pl5⟩/⟨pl6⟩ as well as the member functions ⟨pl8⟩–⟨pl11⟩ were already presented in section 3.1.2.

3.3. The access functions

3.3.1. Function init(). Table 2 shows the implementation of *init*(). In ⟨i1⟩–⟨i4⟩ the global member variables are initialized. In the loop ⟨i5⟩–⟨i11⟩ the local member variables of the sublist headers are initialized. Counters *lr_cnt* and *la_cnt* of sublist S_i are set to i since this will be the value of *gr_cnt* and *ga_cnt* for the first *read_next*() and the first *append*() on S_i respectively; *li_cnt* of S_i is set to $-size + i$ since elements are inserted in the opposite direction as appended or read. Moreover, this choice of the initial counter values simplifies the reinitialization of the list (cf. 3.3.5). Since the initializations of the different sublist headers are independent of each other, they can be processed by several processes in parallel. We therefore also provide a function *par_init*() in which ⟨i5⟩–⟨i11⟩ are replaced by a parallel loop, leading to an execution time of $O(\lceil \frac{k}{p} \rceil)$ for p processes.

```
void init(int k)
  ⟨i1⟩  barrier_init(ri_barrier);
  ⟨i2⟩  $size = k$;
  ⟨i3⟩  $r\_num = gr\_cnt = ga\_cnt = 0$;
  ⟨i4⟩  $gi\_cnt = -1$;
  ⟨i5⟩  for(int $i = 0$; $i < size$; $i$++)
  ⟨i6⟩    ParHeader &$h = hdarray[i]$;
  ⟨i7⟩    fair_lock_init($h.lock$);
  ⟨i8⟩    $h.lr\_cnt = h.la\_cnt = i$;
  ⟨i9⟩    $h.li\_cnt = -size + i$;
 ⟨i10⟩    $h.read = h.first = h.last =$ NULL;
 ⟨i11⟩    $h.i\_first = h.i\_last =$ NULL;
```

Table 2. Implementation of init()

3.3.2. Function append(). Table 3 shows the implementation of *append*(). Let A be a process executing an *append*()-operation. As already mentioned in 3.2, *r_num* holds the number of elements not yet read during the current iteration. Therefore A indicates in ⟨a1⟩ that there will be a new element to be read. The value x computed in ⟨a2⟩ is used in ⟨a3⟩ to determine the header h of the activelist S_a to which *ele* will be appended. In ⟨a5⟩ A waits until all preceding appenders that were directed to S_a have finished their access. After having excluded readers from accessing S_a in ⟨a6⟩ A appends *ele* to S_a (⟨a7⟩ for empty S_a, ⟨a8⟩ for non-empty S_a). In any case *ele* becomes the last element of S_a ⟨a9⟩. If all elements of S_a were already read when A began its *append*()-operation, the *read*-pointer of S_a must be set to *ele* ⟨a10⟩ to indicate that *ele* is the next element to be read from S_a. After having granted access to S_a for readers ⟨a11⟩, A increments *la_cnt* by the number of sublists ⟨a12⟩ to free the next appender from its waiting loop ⟨a5⟩.

```
void append(ParEle *ele)
  ⟨a1⟩  syncadd(&$r\_num$, 1);
  ⟨a2⟩  int $x$ = mpadd(&$ga\_cnt$, 1);
  ⟨a3⟩  ParHeader &$h = hdarray[x \% size]$;
  ⟨a4⟩  $ele \rightarrow next =$ NULL;
  ⟨a5⟩  while($h.la\_cnt \neq x$);
  ⟨a6⟩  fair_lock($h.lock$);
  ⟨a7⟩    if($h.first ==$ NULL) $h.first = ele$;
  ⟨a8⟩    else $h.last \rightarrow next = ele$;
  ⟨a9⟩    $h.last = ele$;
 ⟨a10⟩    if($h.read ==$ NULL) $h.read = ele$;
 ⟨a11⟩  fair_unlock($h.lock$);
 ⟨a12⟩  syncadd(&($h.la\_cnt$), $size$);
```

Table 3. Implementation of append()

3.3.3. Function read_next(). Table 4 shows the implementation of *read_next*(). The execution of *read_next*() by a pro-

```
ParEle *read_next(void)
 ⟨r1⟩  ParEle *ele = NULL;
 ⟨r2⟩  if(tdr(&r_num, 1, 0))
 ⟨r3⟩    int x = mpadd(&gr_cnt, 1);
 ⟨r4⟩    ParHeader &h = hdarray[x%size];
 ⟨r5⟩    while(h.lr_cnt ≠ x);
 ⟨r6⟩    while(h.la_cnt == x);
 ⟨r7⟩    fair_lock(h.lock);
 ⟨r8⟩      ele = h.read;
 ⟨r9⟩      h.read = ele→next;
 ⟨r10⟩   fair_unlock(h.lock);
 ⟨r11⟩   syncadd(&(h.lr_cnt), size);
 ⟨r12⟩ return ele;
```

Table 4. Implementation of read_next()

cess R can only succeed if there are elements to be read. This is checked by using the **tdr**()-operation in ⟨r2⟩. If the **tdr**() does not succeed, R returns NULL (⟨r1⟩/⟨r12⟩), otherwise R proceeds.

The value x computed in ⟨r3⟩ is used in ⟨r4⟩ to determine the header h of the activelist S_a from which R will read its element e_x. In ⟨r5⟩ R waits until all preceding readers that were directed to S_a have finished their access. However, having finished ⟨r5⟩, R can not be sure that e_x is already stored in S_a: Since the appender A of e_x increments r_num in ⟨a1⟩ before actually having put e_x into the list (⟨a7⟩–⟨a10⟩), it might be that R succeeds in decrementing r_num ⟨r2⟩ and terminating ⟨r5⟩ before A has appended e_x to S_a. Thus R must check whether e_x has already been put into S_a by comparing the local read- and append-counters of S_a ⟨r6⟩. Note that if $h.la\_cnt \neq x$ it is guaranteed that e_x was appended to S_a: Since $h.lr\_cnt == x$ for R after ⟨r5⟩, all elements of S_a that precede e_x have already been read and thus, of course, must have been appended to S_a. But this means that $h.la\_cnt$ must have at least the value x when R executes ⟨r6⟩, and therefore $h.la\_cnt == x$ $(== h.lr\_cnt)$ if and only if e_x was not yet appended to S_a .

After having excluded appenders from accessing S_a in ⟨r7⟩ R reads e_x in ⟨r8⟩/⟨r9⟩. Then R regrants access to S_a for appenders ⟨r10⟩ and for the next reader ⟨r11⟩. Finally R returns e_x ⟨r12⟩.

3.3.4. Function insert(). Insertion of an element is performed in a similar way to *append*(). Here *gi_cnt* plays the role of *ga_cnt*, as well as for each sublist header, *li_cnt* is the counterpart of la_cnt.

The code for *insert*() is shown in table 5. In contrast to ⟨a1⟩, we must not increase r_num since r_num only denotes the number of *active* elements, which are not yet read. Lines ⟨s1⟩–⟨s3⟩ correspond to ⟨a2⟩, ⟨a3⟩ and ⟨a5⟩.

```
void insert(ParEle *ele)
 ⟨s1⟩ int x = mpadd(&gi_cnt, −1);
 ⟨s2⟩ ParHeader &h = hdarray[x%size];
 ⟨s3⟩ while(h.li_cnt ≠ x);
 ⟨s4⟩ ele→next = h.i_first;
 ⟨s5⟩ h.i_first = ele;
 ⟨s6⟩ if(h.i_last == NULL) h.i_last = ele;
 ⟨s7⟩ syncadd(&(h.li_cnt), −size);
```

Table 5. Implementation of insert()

We cannot set the $next$ pointer of our element earlier than in ⟨s4⟩, because $h.i\_first$ could be changed by another inserter. In ⟨s5⟩/⟨s6⟩ we put the element at the head of the local sleeplist. Decrementing $h.li\_cnt$ ⟨s7⟩ finishes the critical section and signals to a potential process waiting in ⟨s3⟩ to start its insertion. We now state two important properties of the insert protocol:

1. If we had added the sleeping elements to the sleeplists by *append*()'s instead of *insert*()'s in opposite direction (i.e. the most recently inserted element was appended first) and if the initial value of *ga_cnt* (*la_cnt*) would have been the final value of $gi\_cnt + 1$ $(li\_cnt + size)$, we would have obtained sleeplists identical to the actual ones. To put it into other words: the protocol of *insert*() is *compatible* to that of *append*().
2. As a consequence of ⟨i3⟩, ⟨i4⟩, ⟨i8⟩, ⟨i9⟩ and the insert and append protocols, for each $x \in \{gi\_cnt + 1, \ldots, ga\_cnt\}$ there exists an element in the list which was added to a sublist i, $i = x \bmod size$, x resulting from an **mpadd** operation of ⟨s1⟩ or ⟨a2⟩ (cf. figure 3). As we will see, this fact simplifies the implementation of *reinit*().

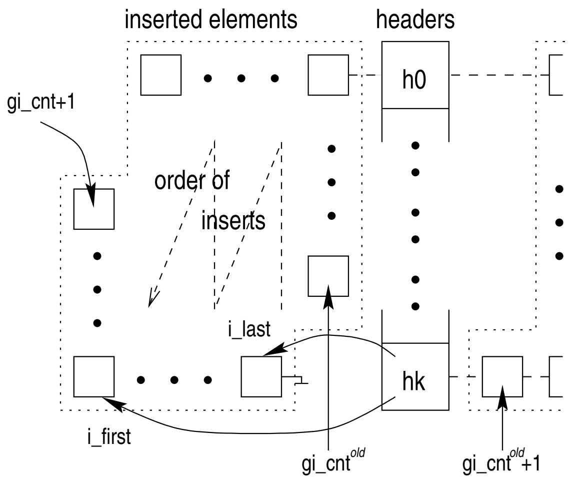

Figure 3. Insertion of elements

3.3.5. Function reinit(). The code for *reinit*() is shown in table 6. This function activates the sleeping elements and

```
void $reinit$(int $p$)
  ⟨ri1⟩  ParEle &$h$;
  ⟨ri2⟩  barrier($ri\_barrier$, $p$);
         executed by one process:
  ⟨ri3⟩    $gr\_cnt = gi\_cnt + 1$;
  ⟨ri4⟩    $r\_num = ga\_cnt - gr\_cnt$;
         foreach sublist header h pardo:
  ⟨ri5⟩    if($h.i\_first \neq$ NULL)
  ⟨ri6⟩      $h.i\_last \rightarrow next = h.first$;
  ⟨ri7⟩      $h.read = h.first = h.i\_first$;
  ⟨ri8⟩      $h.i\_first = h.i\_last =$ NULL;
  ⟨ri9⟩    else $h.read = h.first$;
 ⟨ri10⟩    $h.lr\_cnt = h.li\_cnt + size$;
 ⟨ri11⟩  barrier($ri\_barrier$, $p$);
```

Table 6. Implementation of reinit()

sets the list back to a state, from which a new iteration over the list elements can be started. Reinitialization is done in parallel by all p processes where each process treats some of the sublists. During the execution of *reinit*() we cannot allow *read_next*()-, *append*()- or *insert*()-accesses. This is guaranteed by gathering all p processes in a barrier at the begin and at the end of a *reinit*()-execution (⟨ri2⟩, ⟨ri11⟩).

We need to consider three aspects for the complete reinitialization:

Reinitialize the global control structure. Here we need to set gr_cnt to the smallest number x of an element, i.e. the smallest value a **mpadd** on a write counter (ga_cnt or gi_cnt) delivered, when an element was added to the list (⟨a2⟩, ⟨s1⟩). This need is a direct consequence of **(I1)** and **(I2)**. Due to *init*() and the insert protocol, x always equals $gi\_cnt + 1$; ⟨ri3⟩ thus ensures, that at the beginning of the next iteration, we will visit the head of the parallel list. Additionally, we must set r_num to the new total number of active elements ⟨ri4⟩. Here we can profit from property 2 of the insert protocol (cf. 3.3.4).

Activate sleeping elements. This is performed in ⟨ri6⟩. We can do this, because our insert protocol and ⟨i8⟩/⟨i9⟩ ensure, that the numbers of successive elements (no matter whether inserted or appended) in our sublists differ exactly by $size$.

Reinitialize the headers of the sequential sublists. Consider a sublist header h. If there were no inserts, ⟨ri9⟩ sets the local read counter to its original value. Otherwise we have to merge corresponding active- and sleeplists (⟨ri6⟩–⟨ri8⟩). Analogous to the setting of gr_cnt, we have to set the local read counter to the smallest number of an element stored in the sublist. If there were inserts during the last iteration, this is $h.li\_cnt + size$. The incrementation by $size$ is necessary to compensate for the last decrementation of $h.li\_cnt$ in ⟨s7⟩. If there were no inserts, $h.li\_cnt$ was not modified during the previous iteration. We now must distinguish two cases: In the first case the first element of the sublist was added by *append*() (i.e. there were no inserts on this sublist so far) or the list is still empty. Here ⟨i8⟩/⟨i9⟩ guarantee the correctness of the assignment ⟨ri10⟩: $h.lr\_cnt = i = -size + i + size = h.li\_cnt + size$. In the second case, the first element of the sublist was added by *insert*() some iterations before the considered one. Then we have the same situation as if the element was inserted in the previous iteration and know by induction over the number of reinitializations that the value $h.li\_cnt + size$ is correct.

4. Analysis

In order to determine the quality of our parallel list we analyze the amount of memory used by the data structure and the number of instructions consumed by the access functions.

4.1. Memory usage

On the SB-PRAM objects of elementary data types consume one and objects of type `barrier_t` or `fair_lock_t` consume two 32-bit memory words. From table 1 we derive the memory consumption of a parallel list containing n elements (cf. table 7).

Structure	Size (word)
`ParEle`	n
`ParHeader` Array	$20 \cdot 2^{\lceil \log_2 p \rceil}$
`ParList`	8

Table 7. Memory usage of the parallel list

The memory consumption for the `ParHeader` Array is computed as follows: One object of type `ParHeader` consumes 10 memory words. To avoid delay situations due to the wait-loops ⟨a5⟩, ⟨r5⟩ and ⟨s3⟩ the length of the array of `ParHeader`s should be at least the maximum number p of processes accessing the parallel list simultaneously. In the case of uniform access time this size completely avoids delay situations. However, we have observed that *read*()- and *append*()-operations can lead to serializations due to the locks, if the operations are executed on the same sublists. Depending on the access scheme, it can therefore be useful to enlarge the array by a small factor. In our case a factor of two proved to be sufficient. As already mentioned in 3.1.2, the total length of the array must be a power of two to guarantee correct behavior of the access protocols. Altogether, the recommended length of the `ParHeader` Array is the value specified in table 7.

4.2. Runtime

In order to ascertain the runtime of the routines of the parallel list, we count the number of instructions needed. This provides the desired informations, since on the SB-PRAM every instruction takes one machine cycle. The numbers given do not include the function prologue and epilogue, in which the CPU registers are saved or restored respectively.

The instruction counts provided in table 8 refer to an extended version of the parallel list which not only supports the access functions presented above but also supports the *removal* of elements. If we do not need to support this function, the values will be slightly smaller.

Functions	Instruction count
append()	$\{62\ldots67\}+6w_a+4l_a$
insert()	$\{38\ldots43\}+6w_i$
read_next()	$\{72\ldots83\}+6w_{r1}+6w_{r2}+4l_r$
reinit()	$\{95\ldots101\}+4b+\left\lceil\frac{k}{p}\right\rceil\cdot\{13\ldots30\}$

Table 8. Instruction counts

An instruction count i specified as $\{a\ldots b\}$ signifies that the possible values for i are in the range $\{a,\ldots,b\}$. The variables in this table have the following meaning (for $\frac{k}{p}\geq 1$ we have $w_a=w_i=w_{r1}=0$ in most cases):

w_a	no. of *additional* iterations spent in loop ⟨a5⟩
l_a	no. of *additional* iterations spent in lock ⟨a6⟩
w_i	no. of *additional* iterations spent in loop ⟨s3⟩
w_{r1}	no. of *additional* iterations spent in loop ⟨r5⟩
w_{r2}	no. of *additional* iterations spent in loop ⟨r6⟩
l_r	no. of *additional* iterations spent in lock ⟨r7⟩
p	no. of processes accessing the list simultaneously
k	no. of sublists of the parallel list
b	no. of loops the processes have to wait in barrier ⟨ri2⟩

5. Summary

We have developed a parallel list for the SB-PRAM which is scalable in the number of accessing processes. Being very fast, the expected number of instructions needed for r/w-operations is in the range $\{40,\ldots,90\}$. Since the list is implemented as a C++ class, other data structures can easily be derived from it, permitting an easy integration into user programs.

However, the data structure is not only altered by write- but also by read-operations. Therefore, it does not support nested iterations over its elements. In the case of read-only iterations, i.e. if several iterations shall be performed simultaneously and it is guaranteed that no *insert*()'s or *append*()'s are executed during the iterations, this disadvantage can be overcome: By duplicating gr_cnt, the lr_cnt's, the local $read$-pointers and slightly modifying the access functions, an arbitrary number of parallel read-only iterations can be supported.

The simpler case where processes shall also execute "private" iterations in addition to one parallel iteration can be handled by additionally linking the elements in a sequential list, which does not need to be altered during read-accesses.

Apart from the functions presented we also implemented a method to (incompletely) remove elements during an iteration and a method to "compress" the list erasing traces of formerly removed elements. Support for this kind of deletion will result in slightly higher memory consumption.

References

[1] F. Abolhassan, J. Keller, and W. J. Paul. On the Cost–Effectiveness and Realization of the Theoretical PRAM Model. FB 14 Informatik, SFB-Report 09/1991, Universität des Saarlandes, May 1991.

[2] P. Bach. Entwurf und Realisierung der Prozessorplatine der SB–PRAM. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1996.

[3] J. Edler. *Practical Structures for Parallel Operating System*. Ph.D. thesis, New York University, May 1995.

[4] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th ACM Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[5] T. Goeler. Der Sortierknoten der SB–PRAM. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1996.

[6] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Prog. Lang. and Sys.*, 5(2):164–189, Apr. 1983.

[7] J. Keller. *Zur Realisierbarkeit des PRAM Modells*. Dissertation, Universität des Saarlandes, FB Informatik, 1992.

[8] C. Lichtenau. Entwurf und Realisierung des Speicherboards der SB–PRAM. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1996.

[9] G. K. R. Becker and F.-T. Suttmeier. DEAL - Differential Equations Analysis Library. Available via http://gaia.iwr.uni-heidelberg.de, IWR Heidelberg, Germany, 1995.

[10] J. Röhrig. Implementierung der P4-Laufzeitbibliothek auf der SB–PRAM. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1996.

[11] D. Scheerer. *Der Prozessor der SB–PRAM*. Dissertation, Universität des Saarlandes, FB Informatik, 1995.

[12] T. Walle. *Das Netzwerk der SB–PRAM*. Dissertation, Universität des Saarlandes, FB Informatik, 1996.

[13] J. Wilson. *Operating System Data Structures for Shared-Memory MIMD Machines with Fetch-and-Add*. Ph.D. thesis, New York University, June 1988.