The Programming Environment of the SB-PRAM

T. Grün, T. Rauber, J. Röhrig
Computer Science Department, Universität des Saarlandes,
PF 151150, 66041 Saarbrücken, Germany
{gruen,rauber,bird}@cs.uni-sb.de

Abstract

The SB-PRAM is a shared-memory parallel computer that realizes the CRCW-PRAM model from theoretical computer science. In this paper, the SB-PRAM system is described from a programmers point of view. Special emphasis is put on the process creation scheme and on the efficient implementation of synchronization constructs of the P4 library.

Key Words  architecture, massively parallel systems, software, PRAM, shared memory

1 Introduction

The theoretical PRAM Model [6] is widely used in the theory community for specifying parallel algorithms in an elegant way [6]. A PRAM consists of an unbounded set of processors which compute synchronously in parallel. There is a single unbounded shared memory in which each processor can access any cell in unit time. This allows a synchronous execution of parallel programs on the instruction level leading to a fine grain parallelism without time consuming synchronization. There are different possibilities for dealing with concurrent accesses to a single memory cell. The EREW (exclusive read, exclusive write) model forbids concurrent read or write accesses, i.e. read or write accesses of different processors to the same memory cell at the same time. The CREW (concurrent read, exclusive write) model allows concurrent read accesses, but forbids concurrent write accesses. The CRCW model allows both concurrent read and write accesses.

The SB-PRAM [1] realizes a priority CRCW-PRAM, i.e. for concurrent write operations to the same memory cell the processor with the highest processor number is allowed to write into the memory cell, the values of the other processors are thrown away. The global shared memory can be accessed in unit time, i.e. in the same time in which an arithmetic operation can be executed. This is obtained by hiding the latency of the network between the processors and the memory modules by a pipelining of the processors: each physical processor simulates a number of virtual processors.

In Sections 2 and 3, we give a short description of the SB-PRAM hardware and system structure. The compiler and the programming environment will be outlined in Section 4. Section 5 shows how the P4 library for parallel shared memory computers can be implemented efficiently.

2 SB-PRAM Hardware

Figure 1 gives an overview of the SB-PRAM hardware. There are \( p = 128 \) physical processors connected to a global shared memory, consisting of \( p \) memory modules. For each memory request, a network packet is created and sent through the butterfly network to its destination module.

The addresses are hashed to yield an equal distribution of the accesses to the memory modules, as well as to randomize network traffic. For hiding the memory latency, two mechanisms are used: First, a delayed load is introduced, i.e. a processor must not use the result of a load instruction in the next instruction. Second, each physical processor simulates 32 virtual processors, which are scheduled in a round robin mode. From a virtual processor’s view, memory can be accessed in the same time as an arithmetic operation. This compares well to other shared memory machines like the KSR1/2 [10], the Stanford DASH [9], or the BBN Butterfly [4] which use caching and work with a physically distributed memory.

Simulations of network traffic and memory module load for different access patterns [5] have suggested to simulate \( \zeta(3 \log p) \) virtual processors per physical processor, where \( \zeta(x) \) denotes the smallest power of two, being greater than \( x \).

The physical processor [7] implements a Berkeley RISC instruction set, augmented by single precision floating point instructions. Memory is always addressed by 32 bit aligned words. The processor has direct access to the on-board program memory. For simulating the virtual processors, a fast SRAM holds 32 register sets, consisting of 32 registers each. The non-volatile, dual-ported local memory does not hold any application data, but serves mainly as a disk buffer. All application data are stored in the global memory.

Before a memory access request is transformed to a
network packet, some operations are performed on the non-hashed address. If the highest address bit is set, a special register — the hi_base register — is added to the current address. Otherwise the lo_base special register is added. This allows the virtual processors to map two different, logical address spaces onto physical memory. Then, the resulting address is checked against the memory protection registers which specify the upper (hi_protect) and lower (lo_protect) bounds of user addressable memory. If the address is out of range, an error is reported. Using these hardware facilities, it is possible to implement a simple memory management system for multiprogramming. Furthermore, memory segmentation in a multiprogramming environment can be prevented.

In the current implementation, the lo_base and lo_protect register are identical. All special registers mentioned above, can be specified independently for all virtual processors.

The SB-PRAM offers multiprefix instructions [11] which enable several processors to perform simple operations on a memory cell in parallel. Let the processors $i_1, \ldots, i_n$ ($i_j < i_{j+1}$) execute synchronously the instruction

$$\text{MP} \odot <\text{address}>, <\text{operand}>,$$

with $\odot \in \{\text{ADD}, \text{OR}, \text{AND}, \text{MAX}\}$
on a memory cell $M$ with the specified address. Processor $i_j$ contributes operand $o_j$. After the multiprefix instruction

- $M$ holds the value $m \odot (C_{k=1}^{n} o_k)$, where $m$ denotes the old content of $M$, and
- processor $i_j$ receives $m \odot (C_{k=j-1}^{n} o_k)$ as result.

The operation $\odot$ is computed on the fly by ALUs in the network nodes using the parallel prefix computation scheme. A multiprefix operation takes the same execution time as a read operation. The name “multiprefix” arises from the possibility that several groups of processors perform distinct parallel prefix operations in parallel.

3 SB-PRAM System

Figure 2 shows the overall structure of the SB-PRAM system. A development workstation which can be located everywhere in the Internet is connected to the actual SB-PRAM system.

Figure 2: Overview of the SB-PRAM System

The development workstation contains the cross development software. The tool set consists of a compiler, assembler, linker, loader, and a simulator for the resulting code\(^1\). The application programs and the operating system have been developed using the simulator before the real hardware was available.

For interfacing with the real SB-PRAM system, a control program for remote process execution is used. It simulates a terminal for standard I/O, giving the programmer a feel of telneting to the SB-PRAM host. The host computer is connected to the SB-PRAM machine via a host adaptor, which allows access to global memory and program memories. Apart from memory access, the host adaptor generates control signals (like interrupts or reset) for the SB-PRAM and receives interrupt requests from the physical processors.

The SB-PRAM operating system PRAMOS is distributed among the host computer and the physical processors of the SB-PRAM. The host computer handles the user login procedure and schedules the program execution requests. When a program has been started, the host only acts as an I/O processor for the SB-PRAM, handling terminal I/O to the control program running on the development computer and disk I/O to the host file system.

4 System Software

The programming environment of the SB-PRAM offers two programming models and the corresponding compilers: The FORK model and the P4 model. In the following, we describe the P4 model and the accompanying libraries in more detail. A description of the FORK

\(^1\)The software is located at ftp-wjp.cs.uni-sb.de in the directory /pub/pram
model can be found in [8]. In particular, we consider the interaction of the compiler with the operating system for program execution and system calls.

### 4.1 PGCC Compiler

The SB-PRAM compiler for the P4 model is the PGCC compiler which is a port of the GNU C compiler with several extensions and restrictions. The most severe restriction is that all primitive data types are 32 bit wide. When using char variables with 32 bit, internal computations and terminal I/O are not affected, but file I/O becomes complicated. We provide the UNIX programs `sbrcat` and `sbruncat` to convert ASCII files between the 8 bit and 32 bit character format.

Another restriction is, that the data type double is internally mapped to the float type because the current SB-PRAM processor provides floating-point operations only for 32-bit values. This may create problems for numerical applications if these require a high precision. The difficulties for these applications could be prevented by emulating all double operations in assembler functions, but this would significantly slow down the performance of the machine.

As extension to the ANSI C language, the new storage class qualifiers private and shared are provided. They determine whether each process gets its own copy of a variable or whether all processes share the same variable. Similarly, the function `shmalloc()` allocates dynamically space for shared variables, whereas the standard `malloc()` function reserves private space.

The PGCC compiler creates object files in COFF format, using the segments: text, 1ddata, gpdata, gsdata, gpbss, gsbs and args. The usual data and bss segments are split up into a private (gpdata, gpbss) and shared (gsdata, gsbs) portion of the global memory. The loader relocates the segments, relying on the base register addressing scheme for private data. The 1ddata provides information necessary for the startup code and args contains the command line arguments. The memory structure during program execution is shown in Figure 3.

A contiguous memory block that is assigned by the operating system to the program, is partitioned by the compiler. The shared segments 1ddata, g pdata and gsbs are arranged in the lower half of the logical address space of each virtual processor and mapped onto the physical memory starting at the low protect address. For all virtual processors a private memory frame (PMF) is set up in the middle of the higher logical memory half and mapped onto an individual address range of the physical memory. The PMF comprises the constant sized args, g pdata, and gpbss segments, as well as the private stack and heap which can grow up to the sizes ps_size (respectively phsize). Both values are specified in the 1ddata segment.

The C Standard library has been adjusted to the 32 bit data types. The Mathematical Library implements all operations using 32 bit floating point format. A Multiprefix Library gives the C programmer access to the powerful multiprefix operations of the SB-PRAM. Finally, the Parallel Library contains a set of efficient parallel data structures and algorithms. These include different versions of locks, barriers and parallel queues.

### 4.2 PRAMOS Operating System

The operating system provides the basic facilities for program execution and access to files. It manages the system resources like memory space, I/O devices, and process allocation. The processors are allocated statically in multiples of the number of physical processors. This guarantees that each user program has at least one virtual processor on each physical processor, i.e. each processor has always direct access to all disks. One “service processor” per physical processor is used by the operating system to poll all external interrupt requests. This enables the applications to run parallel program parts synchronously.

The interface between the operating system and the compiler is specified by system calls. A large part of the system calls deals with the file system, using a UNIX like programming interface. Each virtual processor has access to the disk attached to the corresponding physical processor. Moreover, each processor can access the file system on the host processor and can make terminal I/O on the development workstation. A parallel file system that allows a processor to access all SB-PRAM disks, will be implemented as a library. If a processor p wants to output data to the disk of a physical processor to which it does not belong, the output is performed by another virtual processor that runs on the requested physical processor. This is possible, because the process allocation strategy guarantees that every program gets at least one virtual processor on each physical processor.

### 4.3 Program execution

A master/slave approach is used for process creation: At program start, only one process is active. An active process may create further processes, until no more idle processes are left. In the following, we describe the support of the operating system for the start of a user program, the creation and management of threads, and for program termination.

**Program Start** Having successfully logged in, a user requests to start an application program. Then, the host sends a load request to the SB-PRAM, indicating the required number of processors n and memory size m. If the PRAM part of the operating system can offer the resources needed, it requests the host to load the text and data segments. When the host has accomplished

2 Apart from the command line arguments, some parameters like ps_size, phsize and shsize may be specified.
this tasks, it sends an acknowledgment packet causing the PRAM to execute the program.

The initialization routines in the C–startup code are executed by all \( n \) processors, that have been allocated by a program, in parallel. At the end of the startup code, all but one processor enter a wait loop \( W \); let \( w \) be the number of processors waiting in \( W \). Then, the remaining processor starts executing the \texttt{main()} function of \( A \).

**Multi-Fork** At any point of the execution of \( A \) a running process \( P \) can create \( k \leq w \) new processes by executing a \texttt{multifork}(\( k \)) function call. Except for the possibility of simultaneously creating several processes in parallel, the semantics of \texttt{multifork}() is like the semantics of \texttt{fork}() in UNIX. The child processes inherit the context of \( P \) (i.e. the contents of \( P \)'s private data segments, private heap, and private stack and the information held for \( P \) on the PRAMOS–level) and start execution at the statement following the \texttt{multifork}() call.

At the beginning of a \texttt{multifork}(), \( P \) allocates a PMF for each of the \( k \) children. Then, the PMF of \( P \) is copied to the children’s PMFs. Copying is done by all \( w + 1 \) available processors; afterwards \( w - k \) processors reenter \( W \). The remaining \( k \) processors execute the rest of the initializations and start program execution.

**Program termination** As in UNIX, a process \( P \) terminates by calling \texttt{exit}(). The PMF of \( P \) is freed and the processor on which \( P \) was running reenters \( W \). If all \( n \) processors are in \( W \), the program will terminate. When one process of \( A \) issues the system call \texttt{abort}() all processes of \( A \) are forced to terminate immediately, resulting in a complete (abnormal) program termination. In this case, a C stack back trace will be generated in order to simplify debugging.

5 Implementation of P4

P4 is a library of C macros and subroutines for the implementation of portable parallel programs\(^3\). It provides communication concepts both for distributed memory computers and for shared memory computers (monitors). In the following, we will not consider the message passing concepts, although they have been ported successfully to the SB-PRAM.

The P4-library is intended to be portable. Therefore it makes little assumptions about the underlying system. Besides the \texttt{multifork}() mechanism for process creation only a lock mechanism had to be to provided for obtaining a first runnable version of P4.

The performance of P4 can be considerably improved by using special features of the SB-PRAM, especially the \texttt{multiprefix} instructions. Improvements were made to the P4–startup code, to the standard P4–monitors, to the shared memory allocation system, and to the shared memory message passing routines.

In the following, we describe the realization of locks on the SB-PRAM, give an overview over the standard P4–monitors, and describe the improvements.

5.1 Implementing a lock mechanism

P4 requires a lock mechanism that provides a data type \texttt{lock_t} and functions \texttt{lock_init()}, \texttt{lock()} and \texttt{unlock}(), to initialize a lock, to become owner of a lock, and to free a lock, respectively. A process can free a lock \( l \), even if the process is not the owner of \( l \).

On the SB-PRAM a simple lock mechanism can be implemented by using the \texttt{mpmar}–instruction (see Table 1). A lock consists of a single memory cell \( l \) which must be initialized with 0 before its first use. If several processes execute the \texttt{lock}(\( l \)) function concurrently, the \texttt{mpmar}–instruction returns 0 to only one process. All

\(^3\)P4 was developed at Argonne National Laboratory. Further information about P4 can be found in [3] and [2]. The original P4 distribution can be obtained by anonymous FTP from info.mcs.anl.gov in the directory /pub/p4.
other processes get 1 and continue executing the loop. Freeing the lock is done by executing `unlock(l).

<table>
<thead>
<tr>
<th>lock_init(1) :=</th>
<th>lock(1) :=</th>
<th>unlock(1) :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ l := 0; ]</td>
<td>[ while (mpmax(l, 1) \neq 0); l := 0; ]</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Simple lock on the SB-PRAM

5.2 Standard P4 Monitors

A detailed description of the P4 monitor concept can be found in [2]. There, a monitor is defined as "an abstract data type, consisting of data to be shared among concurrently executing processes, initialization code for the data, and a set of operations that can be performed on the data." At any time, only one process has access to the data protected by a monitor. Monitors are implemented by using a set of functions, called the monitor building primitives, which are based on the lock mechanism. P4 provides three standard monitors: the barrier-monitor for barrier synchronization, the getsub-monitor to implement the parallel loop, and the askfor-monitor to manage task pools. On the SB-PRAM, the performance of these standard monitors can be improved by permitting parallel access to the data protected by the monitor. This becomes possible due to the multiprocessor-instructions of the SB-PRAM. In the next section, we show exemplarily, how the performance of the barrier-monitor can be improved.

5.3 Improving the barrier-monitor

The barrier-monitor is used to synchronize processes. It provides a data type `barrier_t` and two functions `barrier_init()` and `barrier()`. Let \( b \) be a shared variable of type `barrier_t` which has been initialized by `barrier_init(b)`. To synchronize \( n \) processes, each process executes a `barrier(b, n)`-call which causes it to wait until all processes have issued their `barrier(b, n)`-call.

**Original P4 version** The data structure \( b \) consists of a counter \( b.c \) and two locks \( b.l_m \) and \( b.l_q \). The variable \( b.c \) counts the number of processes waiting in the barrier, \( b.l_m \) is used to prevent parallel access to the data protected by the monitor, and \( b.l_q \) is used as a "wait queue". Prior to its first use, a barrier must be initialized by exactly one process calling `p4_barrier_init(b)` (see Table 2). In line 4, the initializing process locks \( l_q \) causing the next process executing a `lock(b.l_q)` to block.

Process synchronization with the P4 barrier is split into two phases: In phase one, the first \( n - 1 \) processes, entering the `p4 Barrier()`-function, increment

\[ \text{p4 barrier init}(b) := \]
\[(1) \quad b.c := 0; \]
\[(2) \quad \text{lock}(b, l_m); \]
\[(3) \quad \text{lock}(b, l_q); \]
\[(4) \quad \text{lock}(b, l_q); \]
\[(5) \quad \text{if}(b.c < n - 1) \]
\[\{ \quad b.c + \text{unlock}(b, l_m); \]
\[\{ \quad \text{lock}(b, l_q); \}
\[\} \quad \text{if}(b.c > 0) \]
\[\{ \quad b.c := \text{unlock}(b, l_q); \}
\[\} \quad \text{unlock}(b, l_m); \]

Table 2: Implementation of `p4_barrier()`

\( b.c \) (which is protected by \( b.l_m \) in this phase), and afterwards block on \( b.l_q \) (line 2).

The second phase starts, when the \( n^{th} \) process enters line 3, while all other processes are trying to lock \( b.l_q \) in line 2. It decrements \( b.c \) and frees one of the blocking processes which, in turn, executes line 3. The last process freed unlocks \( b.l_m \) (line 4).

The access to the counter is serialized by the locks. So the minimal execution time for the barrier is \( O(n) \). In particular, `p4_barrier()` serializes program execution even if the processes are synchronous when entering the barrier. This may get expensive if many processes are to be synchronized and if the average time spent between two barrier synchronizations is small.

**Improved SB-PRAM version** On the SB-PRAM a barrier can be implemented using the `mpadd`-instruction and two counters \( b.c_u \) and \( b.c_p \) which must be initialized with 0 before the first use of the barrier (see Table 3).

\[ \text{sbp barrier init}(b) := \]
\[(1) \quad b.c_u := 0; \]
\[(2) \quad b.c_p := 0; \]
\[(3) \quad \text{while}(b.c_u \neq 0); \]
\[(4) \quad \text{if}(mpadd(b.c_p, 1) < n - 1) \]
\[\{ \quad \text{while}(b.c_u \neq 0); \]
\[\{ \quad \text{else} \}
\[\{ \quad b.c_p := 0; b.c_u := n; \}
\[\{ \quad mpadd(b.c_p, -1); \}

Table 3: Implementation of `sbp_barrier()`

The variable \( b.c_u \) is used to indicate whether the barrier is in the incrementing phase (\( b.c_u = 0 \)) or in the decrementing phase (\( b.c_u > 0 \)). The number of processes waiting in the barrier during the incrementing phase is counted by \( b.c_p \). In line 3 of `sbp_barrier()`, the \( n^{th} \) process switches the barrier from the incrementing to the decrementing phase. The variable \( b.c_p \) is reset to 0 and \( b.c_u \) is set to \( n \), allowing the other processes to leave the while loop in line 2. Lines 4 and 1 assure that no new processes enter the barrier until the current process group has left it.

Program execution is not serialized, when the pro-
cesses already run synchronously before entering the barrier. In this case, the time that expires between the moment when the first process enters the barrier and the moment when the last process exits the barrier is $O(1)$, with a very small constant. Figure 4 shows a comparison between $p4\_barrier()$ and $sbp\_barrier()$. The $n$ processes run synchronously when entering the barrier. The time needed for crossing the standard P4 barrier grows linearly with the number of processes $n$ whereas the time needed for crossing the SB-PRAM barrier is constant. (Note: On both axes a logarithmic scale is used.) Similar improvements are reached for other monitors.

![Figure 4: $p4\_barrier()$ versus $sbp\_barrier()$](image)

6 Conclusion and future work

The SB-PRAM is a shared memory computer with 4096 processors. It has been designed to efficiently support the theoretical PRAM model, but it is also well suited for the P4 programming model which has already been implemented on many hardware platforms. In this model, the use of the SB-PRAM multipref operations leads to an improvement of the standard P4 monitors by several orders of magnitude.

Although not explicitly required by the P4 programming model, we try to keep processors synchronous, and inhibit external interrupts (e.g. disk I/O) from delaying processors in an unpredictable way. In the current operating system design, this is achieved by reserving one virtual service processor per physical processor. In the future, we will study operating system variants with no service processors, as well as the consequences of neglecting synchrony.

So far, the complete software system has been developed and tested using the SB-PRAM simulator only. At the moment of this writing, the ASIC processor chip runs the first, simple programs\(^5\). We expect a small prototype with two physical processors to be available by the end of 1995. If this small prototype runs stable, a big machine with 128 physical processors will be built.

References


\(^5\)More information about the current status of the SB-PRAM project can be obtained via World Wide Web from the URL [http://www-wjp.cs.uni-sb.de/SPARM/](http://www-wjp.cs.uni-sb.de/SPARM/).