Scientific Applications on the SB-PRAM

Arno Formella, Thomas Grün, Wolfgang Paul, Gudula Rünger
Computer Science Department, Universität des Saarlandes
D–66041 Saarbrücken, Germany

Jörg Keller
Fernuniversität-GH Hagen, LG Techn. Informatik II
D–58084 Hagen, Germany

Thomas Rauber
Universität Halle-Wittenberg, Fachbereich Mathematik und Informatik
D–66120 Halle (Saale), Germany

We give a brief survey of the SB-PRAM architecture as a realization of the priority CRCW PRAM model. The sequential semantics of a parallel program allows the use of classical techniques to prove its correctness. The SB-PRAM shows good performance and speedup for the presented five scientific applications which are—due to their irregular behavior—difficult to parallelize efficiently on other parallel machines.

1 Introduction

The SB-PRAM is a realization of a synchronously working priority CRCW PRAM with single-instruction multiprefix operation. A four processor prototype is built.[13] Synchronous means that all processors execute one instruction within one time step (clock cycle). Concurrent read and concurrent write (CRCW) means that different processors can read or write the same memory cell simultaneously. Priority means that in case of concurrent write the participating processor with largest number wins. Multiprefix means that several groups of processors can perform parallel prefix operations at the same time. The prefix order is determined by the priority as well. The execution time of such an operation is equal to the execution time of a fixed point arithmetic operation.

The concepts underlying the SB-PRAM have not been developed independently from others. The concept of virtual processors in hardware was already used in the Denelcor HEP,[17] and is used again in the Tera MTA. The concepts of hashing and combining were already used in the NYU Ultracomputer and IBM RP3. NYU Ultracomputer, IBM RP3, Tera MTA, and Stanford DASH also have hardware support for parallel prefix operations,[12] although DASH is restricted to increment/decrement. The aim of the SB-PRAM which is based on the Fluent Machine[15] is to bring all these concepts together in a single machine.

Section 2 introduces the PRAM model. In Section 3, we show how to prove the correctness of a program on the SB-PRAM. Section 4 briefly describes the hardware of the SB-PRAM and the current status of the prototype. It follows a survey of the system software in Section 5. Section 6 presents the programming model from a users point of view and briefly discusses some applications already ported to the machine.

2 PRAM Model

The PRAM (Parallel Random Access Machine)[5] is a model that allows the specification and the analysis of parallel algorithms in a way that abstracts from particular architectures. Hence, programmers can concentrate on parallelism and need not to be concerned about implementation details. The model assumes a finite number n of processors, each working like a random access machine. All processors can access a shared memory with the same speed as they perform a fixed point arithmetic operation. The processors work synchronously and, hence, there are no race conditions because of different processor speeds as it might happen in distributed systems.

There are several variants of the PRAM model, depending on whether one or several processors are allowed to simultaneously read or write a shared memory location. The variants are EREW (exclusive read exclusive write), CREW (concurrent read exclusive write) and CRCW (concurrent read concurrent write); the ERCW variant is not used. If concurrent writes are allowed, there are several possibilities to determine which processor succeeds: An *arbitrary* processor might succeed, one can request that all participating processors write a *common* value, or one can specify a *priority* among the processors, such that the one with the highest priority succeeds. The priority CRCW PRAM is the strongest model. The PRAM model has been successfully employed to develop a wealth of parallel algorithms.[2,7,10]

3 Sequential Semantics

How can we prove that a parallel program for the SB-PRAM is correct? Generally, we cannot expect that we can prove the correctness of a parallel program more easily than a sequential program, but the synchronous priority-CRCW model provides a simple technique that enables us to prove the correctness in the same way as for a sequential program.

Consider the following example. Let $p_{i_1}, \ldots, p_{i_k}$ for $i_1 < \ldots < i_k$ be a group of processors. Each processor p_{i_j} owns a local variable x_j (for instance a value stored in a register). Now, consider a concurrent write operation is executed, i. e., each processor p_{i_j} writes its value x_j into the global memory M at address a. Let us write this (similar to C syntax) as $p_{i_j} : M[a] = x_j;$. For the priority CRCW PRAM, the k concurrent writes can be seen as k consecutive write operations according to the order of the processors. Hence, in a proof we can assume that the k assignments are performed one after another:

$$M[a] = x_1; \; \ldots \; M[a] = x_k;$$

The value finally stored is x_k. The more interesting example of a parallel prefix operation illustrates more clearly the advantage of the sequential semantics. Let us assume that each of the k processors participates in a parallel prefix operation *mpadd*. With the same notation as above, let us write this as $p_{i_j} : mpadd(x_j, \&a);$. For the priority CRCW PRAM, the k concurrent prefix operations can be seen as $3k$ consecutive assignments:

$$y_1 = x_1; \; x_1 = M[a]; \; M[a] += y_1; \; \ldots \; y_k = x_k; \; x_k = M[a]; \; M[a] += y_k;$$

Hence, in the end we have the following *correct* semantics of a parallel prefix add operation:

$$M[a] = M[a] + \sum_{1 \le l \le k} x_l \quad x_j = M[a] + \sum_{l<j} x_l;$$

Thus, we can split the execution of one step of the synchronously working parallel machine into a sequence of simple assignments. The method works well only if the operations which are performed concurrently on the same global memory address are identical. For mixing read, write, and multiprefix operations see the hardware description in Section 4.

4 The SB-PRAM

The SB-PRAM[1] is a massively parallel multiprocessor architecture with p processors that implements a priority CRCW PRAM with $n = v \cdot p$ processors. It is based on Ranade's Fluent Machine.[14] The shared memory is physically distributed among p memory modules. Memory requests are transmitted between the processors and the memory modules via a butterfly network. The SB-PRAM uses universal hashing to distribute addresses among the memory modules. Every shared memory access is remote. Hashing avoids module congestion and leads to a large but uniform memory access time. The latency to access global memory is hidden by using multithreaded processors which simulate v *virtual processors* in a pipeline. Each virtual processor has its own register set, thus context switching does not cause any overhead. Network latency is found to be $3 \log p$ cycles (by simulation) even in the presence of contention, hence v can be set to that or any larger value.

Concurrent access of multiple processors to some memory cell is handled by combining. The requests of each physical processor are sorted according to their hashed addresses. The sorted order of requests is maintained in each network node by merging the incoming streams of requests. Requests to one cell must inevitably meet and can be combined. Answers are duplicated on the way back. Computation of parallel prefix operation is implemented by the same mechanism. The network nodes can perform simple integer arithmetic.

The sequential semantics of the priority PRAM permits us to split one round of virtual processors into several subrounds, which makes the sorting array cheaper. Only $v/4$ requests are sorted at a time. This leads to four subrounds in the emulation of one PRAM step. Combining and concurrent writing is performed in the memory for the different subrounds. To guarantee that different operations on one cell are not issued in the same round, they can be scheduled at run time in odd or even rounds, respectively.

The SB-PRAM prototype consists of $p = 128$ physical processors and the same number of memory modules. A version with 4 processors is running; the assembly of the complete prototype is not finished yet. Each physical processor implements $v = 32$ virtual processors which are scheduled round-robin for every instruction. Load instructions to global memory are delayed by one cycle. The speed of routing chips is 32 MHz. Due to pin restrictions, requests are transmitted between chips in two cycles. Selection starts after having received the first part of a request and takes two cycles as well, hence network speed is 16 MHz. A network utilization of 100% is not possible because conflicts can occur. To keep the protocol between processors and network chips simple, we chose cycle times that are multiples of each other. Hence, 8 MHz was the maximum frequency for the processor.

The physical processor, the sorting chip and the network chip are realized as ASICs. The register sets of the virtual processors are held off-chip in a fast static RAM. With high end technology, e. g., modern process technology, optical links in the interconnection network, and a more sophisticated compiler, e. g., separating local and global memory operations, a clock frequency of 96 MHz for the processor seems to be possible.[4]

5 System Software

In order to guarantee synchronous operation of the SB-PRAM and, thus, sequential semantics of programs, we avoid using interrupts. For handling external I/O, e. g., accessing disks, the operating system reserves one virtual processor per physical processor. It polls on interrupt requests and then handles them. So, 3% of the processing power is lost for the user.

In a multiprogramming environment, an interrupt is the only practical means for aborting programs. Here, we interrupt all virtual processors at the same time, and take care that the virtual processors which do not belong to the aborted program leave the interrupt service routine simultaneously.

We ported and extended the GNU C-compiler for the SB-PRAM. The differences to standard C lie in two newly introduced keywords, namely `shared` and `private` as variable classes, and different sizes for the elementary data types (SB-PRAM is not byte but word oriented). Besides simple C, the parallel programming language FORK[9] is implemented for the machine. A library of PRAM algorithms using FORK is under construction.[11]

To express parallelism in C programs, the parallel library p4 has been ported to the SB-PRAM.[8] Among other routines, the library provides several types of locks, barriers and ask-for constructs. To achieve portability, all of them are based on a simple lock function, which must be adapted to a new architecture. Besides the normal p4 macros, we implemented a very efficient parallel library which makes use of the multiprefix operation.

The sequential semantics is the means to prove the correctness of the implementation. The basic operations for a lock, for instance, are implemented in the following way:

```
lock_init(l)   l=0;
lock(l)        while(mpmax(l,1)!=0);
unlock(l)      l=0;
```

The lock is initialized by setting the lock variable to 0. One or more processors can do this step concurrently at the beginning of a program. If a processor p wants to access the lock, p performs the `lock` operation until the lock is granted. The multiprefix operation *mpmax* guarantees that only one processor achieves a return value of 1 and l is set to 1. An unsuccessful processor spins in the lock until its request is granted. The unlocking operation simply assigns 0 to l. A disadvantage of the simple lock described above is that the processor with lowest priority may never get the lock. However, a fair lock can be implemented by using a ticket mechanism:

```
fair_lock_init(l)  l.t=0; l.c=0;
fair_lock(l)       my_ticket=mpadd(l.t,1);
                   while(l.c!=my_ticket);
fair_unlock(l)     l.c++;
```

The fair lock is initialized by setting both the ticket counter and the current ticket number to 0. If a processor p wants to access the fair lock, p takes a unique ticket through an *mpadd* operation on the ticket counter. Then, p waits until the current ticket counter has reached its ticket value. The fair lock is released by incrementing the current ticket number, giving the next processor 'in line' the chance to acquire the lock.

With almost the same simplicity, we can implement different types of barriers and parallel queues. The overhead in these data structures is constant (from a virtual processor's point of view). Our implementation of a barrier has to execute 36 instructions on every virtual processor, if the processors arrive synchronously at the barrier. This time does not depend on the number of participating processors, while the p4 variant of a barrier has a run time of $68p - 4$, which is linear in the number p of participating processors, since the underlying locks serialize the barrier.

6 Applications

As example problems we consider five irregular applications, most of which are taken from the SPLASH application suite:[16,18] the routing algorithm LocusRoute from SPLASH-1 for VLSI design, a ray tracing algorithm with a hierarchical space subdivision[3], the adaptive hierarchical radiosity algorithm from SPLASH-2, the conservative discrete event simulator PTHOR from SPLASH-1, and the particle simulator MP3D from SPLASH-1. We modified the implementations to adapt them to the specific demands of an efficient implementation on an architecture providing a large number of processors.

We consider a *task oriented shared-memory model* as the parallel programming model in which we formulate a parallel program. The execution of an algorithm is split up into tasks representing well-defined portions of computation[6] which are executed concurrently on different processors. Different tasks cooperate by accessing the same data structures in the shared memory. To maintain determinism, a careful coordination of memory accesses is needed. This can be achieved by the *locking* mechanism as described in Section 5. A centralized *task pool* is used to store all executable tasks. New tasks are included after their creation but not before all tasks that they depend on have been executed. A task is removed from the pool when it is assigned to a processor starting its execution. The centralized task pool is implemented with multiprefix operations. Therefore, it can be accessed in parallel by several processors without serialization.

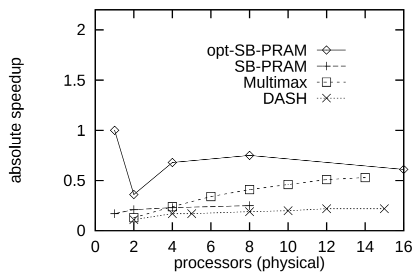
A potential user of a parallel machine is basically interested in the *run time* and the *memory requirement* of the application suite to be executed on the machine. Another important factor for the user is the effort needed to port or program the parallel application onto the machine. The effort should not outweigh the benefit. To convince a user to take advantage of a parallel computer, its performance must be *predictable*.

Besides run time, speedup and efficiency are the means often used for a quantitative performance analysis of parallel programs and the evaluation of different computer architectures. T^* denotes the run time of the fastest known sequential algorithm on one (physical) processor of the parallel machine and $T(p)$ denotes the run time of a parallel algorithm for p processors. We compare different algorithms and machines with two speedup definitions: *relative speedup* $s_r = T(1)/T(p)$ and *absolute speedup* $s_a = T^*/T(p)$.

We measure run time always as wall clock time on physical processors. For the SB-PRAM as a parallel machine with multi-threading processors, we present the run times and speedups in two forms: for a machine where each physical processor is simulating $v = 32$ virtual processors and for an optimally scaled machine for the given problem. To guarantee, in the second case, unit access time to memory for a virtual processor of an SB-PRAM with p physical processors, each physical processor must simulate at least $3 \log p$ virtual ones. The network, the sorting array, and the number of banks in the memory modules must be scaled as well. This leads to a special purpose machine *opt*-SB-PRAM for the problem which might be unfair in a comparison with other general purpose machines; however, we can present an upper bound for the speedup which is achievable with an SB-PRAM-like architecture. Because we are not interested in relative speedups respective to a slow virtual processor, we measured the run times $T(1)$ and T^* using one physical processor where only one virtual processor was busy, but we scaled the run time assuming that the virtual processor runs with the clock frequency of the physical one.

For circuit simulation as a special case of discrete event simulation the degree of parallelism is limited by the number of events that can be simulated in parallel. We implemented the conservative asynchronous PTHOR simulator using multiprefix operations whenever possible (for instance to resolve the deadlocks). The comparisons depicted in Figure 1 show that only the SB-PRAM is able to exhibit a speedup larger than 1 provided the inherent parallelism is not too small.

For complex scenes, ray tracing requires a highly unstructured read-only access to the entire data base. We used a pixel oriented task pool to paral-



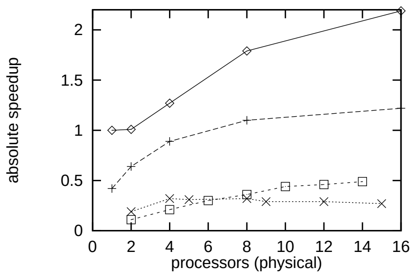
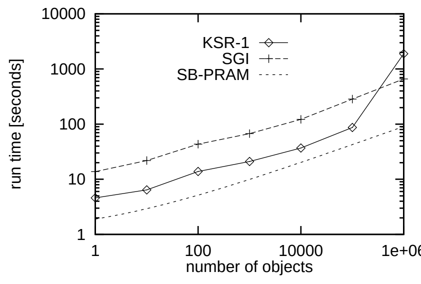


Figure 1: Absolute speedup on PTHOR for Dash-circuit (very low inherent parallelism), left, and for Multiplier-circuit (high inherent parallelism), right.

lelize the kernel. Due to the nature of the problem and the properties of the SB-PRAM almost linear absolute speedup can be achieved. Figure 2(left) compares the expected run time on an SB-PRAM with an 18 processor KSR-1 and a 1 processor SGI challenge for a benchmark scenes with increasing number of objects.[3]

We implemented LocusRoute with two parallel task queues: one for wire tasks and one for route tasks. Figure 2(right) shows the impact of different implementations for the queues. The SB-PRAM is clocked four times slower than DASH, an SB-PRAM achieves a better run time although the speedup is smaller (Figure 3). Instead of locking the data structure, we use multiprefix commands for updating the cost array. Thus, many processors can be employed without performance degradation.



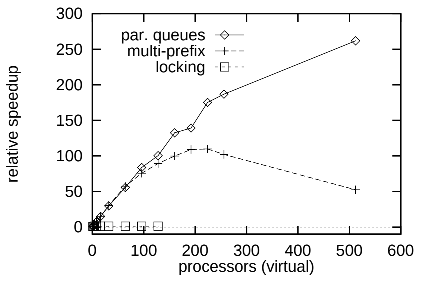


Figure 2: Left: Comparison of the run times for ray tracing. Right: Relative speedup from a virtual processor's of view (SB-PRAM) for LocusRoute with different implementations.

MP3D is a particle simulator for fluid dynamics problems, which uses a

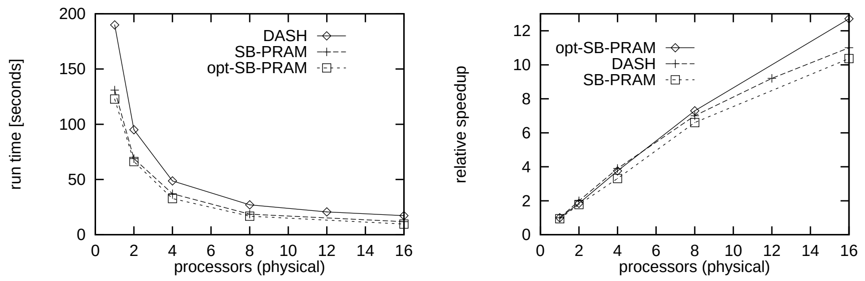


Figure 3: Run time and relative speedup of LocusRoute for the largest benchmark circuit.

uniform subdivision in cubic cells. We implemented a particle based task pool to achieve dynamic load balancing. The change of the static mapping of the original SPLASH implementation to the dynamic mapping leads to an increase of the relative speedup from 20% to 70%. The SB-PRAM can exploit efficiently the high degree of parallelism of this application and achieves a three times faster run time compared to DASH (Figure 4).

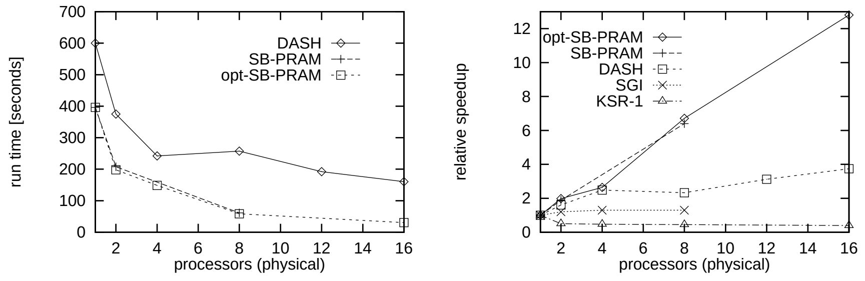


Figure 4: Run time of MP3D on DASH and SB-PRAM and relative speedup on DASH, SB-PRAM, KSR-1, and multiprocessor SGI. The number of particles was 40000 on DASH and SB-PRAM, 50000 on KSR-1, and 8000 on SGI.

The hierarchical radiosity method is used for a fast simulation of global illumination in environments with diffusely reflecting surfaces. For the SB-PRAM we use a lock management with modified tasks of finer granularity to generate enough work for the virtual processors. The resulting speedup for 32 processors is 24.5 compared to 27.5 for the DASH, but the resulting program is much simpler.

Acknowledgments: This work was supported by the German Science Foundation (DFG) under SFB 124 TP D4. For helpful discussion we thank Roberto Bisiani, University of Milan, Italy.

References

1. F. Abolhassan et al. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, Dec. 1993.
2. S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice–Hall, Englewood Cliffs, NJ, 1989.
3. A. Formella and C. Gill. Ray Tracing: A Quantitative Analysis and a New Practical Algorithm. *The Visual Computer*, 11(9):465–476, Dec. 1995.
4. A. Formella and J. Keller and T. Walle. HPP: A High Performance PRAM. In *Proc. Euro–Par'96*, vol. 2, pp. 425–434. LNCS 1124, Springer, Aug. 1996.
5. S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pp. 114–118, 1978.
6. I. Foster. *Designing and Building Parallel Programs.* Addison Wesley, 1995.
7. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.
8. Th. Grün, Th. Rauber, and J. Röhrig. The Programming Environment of the SB-PRAM. In *Proc. 7th IASTED/ISMM Int.l Conf. on Parallel and Distributed Computing and Systems, Washington DC*, Oct. 1995.
9. T. Hagerup, A. Schmitt, and H. Seidl. FORK: A high–level–language for PRAMs. *Future Generation Computer Systems*, 8(4):379–393, Sep. 1992.
10. J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.
11. C.W. Kessler and J.L. Träff. A library of basic pram algorithms and its implementation in fork. In *8th Symp. on Parallel Algorithms and Architechtures (SPAA)*, pp. 193–195, 1996.
12. D. Lenoski and W.D. Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann, 1996.
13. P. Bach et al.} Building the 4 Processor SB-PRAM Prototype. To appear in *Proc. 30th Hawaii Int.l Symp. on System Science HICSS-30*, Jan. 1997.
14. A.G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.
15. A.G. Ranade, S.N. Bhatt, and S.L. Johnson. The Fluent Abstract Machine. In *Proc. 5th MIT Conf. on Advanced Research in VLSI*, pp. 71–93, Cambridge, MA, 1988. MIT Press.
16. J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.
17. B.J. Smith. A pipelined shared resource MIMD computer. In *Proc. 1978 Int.l Conf. on Parallel Processing*, pp. 6–8. IEEE, 1978.
18. S.C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Int.l Symp. on Computer Architecture*, pp. 24–36, 1995.