# HPP: A High Performance PRAM⋆

Arno Formella, Jörg Keller⋆⋆ and Thomas Walle

Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, Germany
{formella,jkeller,twalle}@cs.uni-sb.de

**Abstract.** We present a fast shared memory multiprocessor with uniform memory access time. A first prototype (SB-PRAM) is running with 4 processors, a 128 processor version is under construction. A second implementation (**HPP**) using latest VLSI technology and high speed links shall run at a speed of 96 MHz. To achieve this speed, we first investigate a re-design of the hardware of the SB-PRAM. We then balance processor speed and memory bandwidth by investigating the relation between local computation and global memory access in several benchmark applications. On numerical codes such as Linpack 2 resp. 8 GFlop/s shall be possible with 128 resp. 512 processors, thus approaching processor performance of an Intel Paragon XPS. On non-numerical codes, i.e., circuit simulation and ray tracing, we achieve speedups over a one processor SGI challenge of 35 and 81 for 128 processors and 140 and 327 for 512 processors.

## 1 Introduction

Most of todays massively parallel machines are distributed memory multiprocessors (DMM). They are well suited for numerical problems which are mostly regular, but programming irregular problems is complicated, and performance is often poor. Programming of irregular problems is simplified on shared memory multiprocessors (SMM) such as KSR1 or DASH. However, their non-uniform memory access (NUMA) demands an extensive tuning to obtain expected performance.

SMM with uniform memory access (UMA) avoid these problems, but bus-based machines such as Sequent Symmetry are restricted to small numbers of processors, and PRAM emulations (parallel random access machine) have long been theoretical.

The SB-PRAM is an architecture that emulates a PRAM. Key concepts are avoiding hot spots by universal hashing, implementing concurrent access by combining, and hiding latency by synchronous multithreading with hardware support for multiple contexts. Furthermore, parallel prefix computations without serialization are supported. A 4-processor prototype is running, a 128-processor machine is under construction. We investigate a re-design of the SB-PRAM hardware with todays technology, including the use of high speed network links and the use of processors that allow multiple local instructions per global instruction. We explore to which extent applications can exploit this feature without expensive compiler optimizations. We use both numerical and irregular non-numerical benchmarks.

Our first result is that the improved PRAM called **HPP** runs at a speed of $93.6$ MHz and achieves a ten-fold performance improvement over the SB-PRAM for a 128 processor machine and about 40-fold for a 512 processor machine.

Our second result is that the **HPP** shows good performance on the numerical codes and superior performance on the non-numerical benchmarks. Programming is quite simple, extensive performance tuning was not necessary. For numerical codes similar to Linpack (with matrices of size 16000), we obtain a performance of 15.6 MFlop/s per processor approaching the record performance of 19.5 MFlop/s per processor as rated in [2, p. 379]. For circuit simulation, speedups of 140 over a SUN Sparc20 or a one processor SGI Challenge are possible with a 512-**HPP**. For ray tracing, speedups of 325 over a one processor SGI Challenge are possible with a 512-**HPP**. For 128 processors, these speedups are 35 and 81, respectively. We are not aware of similar performance data on any other parallel machine.

A project with similar goals is the Tera computer [3], now marketed as Tera MTA. Tera directly targeted leading edge technology, e.g., a GaAs processor with a 4 ns cycle time is to be used. To our knowledge, there is no prototype yet available. Tera also uses multithreading and interleaving of global and local instructions, and also provides hardware support for multiple contexts. However, the Tera processor simulates several instructions of each thread before switching and thus looses synchronous behavior. Also, the Tera machine does not support combining and their fetch&add primitive leads to serialization. Furthermore, Tera's processors do not have local memories, thus the fraction of global instructions will be much higher than in **HPP**.

The remainder of the article is organized as follows. In section 2, we briefly review the SB-PRAM architecture and the prototype's technology. In section 3, we investigate how processors, network nodes and network links can be improved. In section 4, we investigate our benchmark applications and show which performance gain is possible by careful instruction scheduling. In section 5, we conclude and present further directions of research.

## 2  SB-PRAM

The SB-PRAM [1] is a massively parallel multiprocessor architecture with $p$ processors providing users with a shared memory. The global memory is physically distributed among $p$ memory modules. Memory requests are transmitted between the processors and the memory modules via a butterfly network. While machines such as KSR1 [2] or Stanford DASH [8] use caches and coherence protocols to avoid slow remote access, the SB-PRAM uses universal hashing to distribute addresses among the memory modules. Every shared memory access is remote. The hashing avoids module congestion and leads to a large but uniform memory access time, whereas caching leads to large variations in access time. The latency to access global memory is hidden by using multithreaded processors which simulate $v$ *virtual processors* (vPs) in a pipeline. Each vP 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 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 sums is implemented by the same mechanism. The network nodes can perform simple integer arithmetic.

The SB-PRAM prototype consists of $p = 128$ physical processors and the same number of memory modules. Each physical processor implements $v = 32$ vPs which are scheduled round-robin for every instruction. Load instructions to global memory are delayed, i.e., the result is only available in the next but one instruction. The physical processor is realized as an ASIC. The register sets of the vPs are held off-chip in a fast static RAM. The processor runs at a speed of $8$ MHz, which is confined by the speed of the interconnection network, as we will see.

The sorting device is realized as a linear sorting array in a separate ASIC. It receives requests at processor speed, and sends requests with network speed. A request consists of a 32 bit address, a 32 bit data word, and 6 mode bits. An answer to a request is a 32 bit data word.

The network speed is 32 MHz. This frequency is determined by using the minimum of (a) the critical path in the network chip, which allows 36 MHz and (b) the speed of the chip I/O which allows 32 MHz. A network chip implements a routing switch with two inputs and two outputs. Due to pin restrictions, a request must be transmitted or received in two cycles. Selection starts after having received the first part of a request and takes two cycles as well. Each network link has four control signals in each direction and thus consists of $(32 + 32 + 6)/2 + 4 = 39$ bits in forward direction and $32/2 + 4 = 20$ bits in backward direction.

As the network needs two cycles to handle a request, a processor utilizing the network at its peak bandwidth can have a speed of at most $16$ MHz. We assume here that a processor is able to access the global memory via the network in every instruction. However, a utilization of 100% is not possible because conflicts can occur within the network. To keep the protocol between processors, sorting devices, and network nodes simple, we chose cycle times that are multiples of each other. Hence, $8$ MHz was the maximum frequency for the processor, utilizing half of the network's peak bandwidth.

The network consists of 7 stages, each with 64 network chips. We implement on a printed circuit board either a 3-stage butterfly network or two 2-stage butterfly networks. Thus, we obtain three levels, each consisting of 16 boards. The wiring between boards is done by flat cables. A link is realized by two cables consisting of 130 wires in total (incl. ground).

## 3 Technological Improvements

The speed of the SB-PRAM processor, $8$ MHz, is quite slow. This speed is determined by the speed of the SB-PRAM network. The network speed is limited by three factors: the processing speed of the network chip, the I/O capacity of the network chip, and the capacity of transmissions between network boards.

To make the SB-PRAM faster, we explore how these limitations change by the use of 1995 technology and how processors and memory modules can be adapted to such a faster network. We investigate how fast we can clock network chips, how fast we can transmit and receive requests with network chips, and how fast the network links can be.

### 3.1 Network Chips

Our current network chip is fabricated in Thesys' 2 metal layer $0.8 \mu m$ HCMOS technology (Master THA172 with about 70K gates and 200 signal pins). A worst case anal-

ysis determined the maximum clock frequency to be 36 MHz. To estimate today's maximum frequency we compare different technology levels from some manufacturers and extrapolate the factor by which the nearly available $0.35\,\mu m$ technology will run faster.

Motorola's M5C technology is $1.875$ times faster than our current. The extrapolation from $0.5\,\mu m$ to $0.35\,\mu m$ gives another factor of $1.4$. Thus, an overall speedup of about $2.6$ will be possible, which yields to an internal clock speed of $2.6 \cdot 36 = 93.6\,$MHz. At the cost of the chip price an even bigger factor is possible, if we switch to faster technologies such as ECL, GaAs or full custom design.

If we make chip I/O independent of the inner computation by placing a register before and after each I/O pad we get the following values for input and output transfer times, i.e. the propagation delay from the external register to the internal register and vice versa. Considering todays common technology the clock-to-output time of the external register is $4\,$ns, the typical delay of an input pad is $1\,$ns, and the internal register setup time is $2\,$ns. If we use a PLL for the internal clock, only the delay of the clock input pad has to be added due to clock skew. This totals to $8\,$ns. In consideration of some smaller delays due to board wires we can say that the inputs can be driven with a frequency of $100\,$MHz. For outputs, the clock-to-output time of the internal register is $3\,$ns. The output driver delay is $4\,$ns if we assume a capacitive load of $20\,$pF. This can be achieved if the external register is close to the network chip. The external register has a maximum setup time of $2\,$ns. This totals to $9\,$ns. Hence, the I/O transfer time is not critical.

For our current ASIC the number of signal pins was a limiting factor due to the cost of a larger package. Therefore, we had to multiplex inputs and outputs. Because the external multiplexer circuit works close to its limits, we can not apply this trick at higher clock rates. But the implementation of a routing switch needs only 4 links with 59 bit each, i.e., 236 signal pins in total. If we assume that we must must add one power pin for every three signal pins about 320 pins will be needed, which is possible with todays ASICs.

## 3.2   Links between boards

The links between two network boards, respectively between processor or memory boards and network boards, reach a maximum length of about $1.5\,$m. Even if we assume higher integration of the entire machine, the length will not be less than $1\,$m. Thus, the transmission with flat cables will be reasonable only up to a frequency of $35\,$MHz. To achieve the same bandwidth as the chips, three flits of one packet have to be transmitted staggered. This leads to a complicated external logic; three different clocks have to be generated for the registers. Furthermore, a multiplexing circuit has to be added. The ICs as well as the additional connectors need an enormous amount of area and cause an additional external propagation delay to the the I/O transfer time. Moreover, one has to employ three cables with 100 wires each, which leads to mechanical problems. Thus, flat cables do not seem suitable.

As an alternative approach we investigate high speed transmission via serial lines. Actually there are three possibilities commercially available:

1. Hewlett-Packard 1012/1014 Transceiver module: 21 bit at 66 MHz; parallel and serial ECL interface (level shifters to TTL needed); 17.2mm x 23.2mm; 6 chips per link;

2. G-Link-T/R from Laser2000: the HP module together with an laser/diode on a tiny printed circuit board forming an optical link: 21 bit at 50 MHz; level shifters can be placed under the modules; 44.45mm x 75.74mm; 6 modules per link;
3. Motorola MC100SX1451-FI100 Autobahn Spanceiver: 16 bit at 50 MHz; parallel TTL and serial ECL interface; 17.4mm x 17.4mm; 10 chips per link;

In all three cases we have to apply two multiplexed cables for one link. But in contrast to flat cables we can choose very fast and very high integrated multiplexer/register ICs which we use in our current machine, too. If we generate an inverted clock the external circuitry can be held simple (cf. Fig. 1).
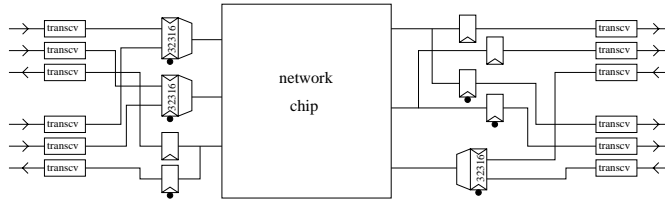


**Fig. 1.** Two multiplexed channels with 3 wires each

In case (2) the construction of a board which contains a 4x4 butterfly network, i.e. 8 links on each side, seems impossible. The soon available Motorola chip with 200MByte/s (-FI200; 16 bit at 100MHz) gives (3) an advantage over the others.

### 3.3 Processor

Simulations for the 128-SB-PRAM have shown that if we are using only 50% of the network bandwidth the waiting time of the processors is negligible if we do a random access in each instruction. Assuming two cycles to handle a request by the network the processor runs by a factor of 4 slower than the network. If we run the network with $93.6$ MHz this is not a limitation.

The processor's I/O needs are as follows: in every cycle it does an instruction fetch (32 bit address, 32 bit instruction). In addition, it might send a request (70 bit) and receive an answer to a request (32 bit). This is too not a limitation as it requires only 166 signal pins at $23.4$ MHz.

The processor can even run faster by using the observation that a load request needs only one network cycle to be handled. In an application, the total number of store instructions is usually not larger than the total number of load instructions. Thus, the average network load is less than $3/4 \cdot 50\% = 37.5\%$. If we increase the processor speed by a factor of $4/3$ to $31.2$ MHz the average load will be 50% again. "Bursts" of store instructions increase the load for a time to $4/3 \cdot 50\% = 67\%$. We observed such bursts in our benchmarks only at the entry of functions. These pushes however can be handled in a local memory and do not need network access at all.

### 3.4 Commercial Processors

If we switch to the technology of commercial processors, it should be possible to implement the SB-PRAM processor including the register sets in a single chip. As we have $v = 32$ register sets, each with 32 registers of 32 bits, this requires an 8 KByte dual-ported RAM. On-chip memories of this size are possible, e.g., the DEC Alpha has first level data and instruction caches on-chip, each with 8 KByte. The sorting device can be implemented on the processor chip, too. This would not even increase the pin count. For the interface to the network the same limitations as for the network chips hold.

We can run the processor faster by a factor $r$, where $r$ is an integer. We must therefore ensure that only one of $r$ instructions accesses the global memory. Then, the average network load is not altered. Furthermore, a delayed load by $2r - 1$ instructions must be tolerated. Note that with increasing $r$ the danger of an overstressed network due to bursts and thus waiting time of the processors grows.

The benchmarks of the next section suggest that a value of $r = 3$ is possible on a range of applications. This pushes processor speed to $3 \cdot 31.2 = 93.6$ MHz which is still below the actual clock frequency of commercial microprocessors. The floating point unit has enough pipeline stages due to the 32 vPs.

Chip I/O can be brought down to a speed of $93.6/2 = 46.8$ MHz by using alternate busses for vPs with odd and even numbers, respectively. For instructions, which now must be fetched every 10 nanoseconds, we will use an on-chip instruction cache and a second level cache off-chip. If the vPs run synchronously, the on-chip cache will be large enough to deliver almost all instructions at the requested speed. If the vPs run asynchronously, e.g., each one operates on a different application, the size of the instruction cache is too small to serve 32 vPs. They can be served by the second level cache, but this may slow down the machine by a factor of 3 to 4. This slowdown can be avoided by carefully assigning vPs to applications. As long as the vPs of one physical processor belong to only four groups, the on-chip cache should still suffice.

### 3.5 Memory

The memory boards must handle requests at a rate of $23.4$ MHz, provided that all memory modules are utilized evenly. The universal hashing only supports "almost" even utilization. Furthermore, there may be bursts by requests arriving every other network cycle, i.e., at $93.6/2 = 46.8$ MHz.

To handle this, each memory module consists of four banks of EDRAM which is a fast dynamic RAM with on-chip cache. The EDRAM is available in a $1M \times 36$ SIMM package with a cycle time of $85\ ns$. If we assume that at most each third request accesses a certain bank, then the module can handle requests at a rate of $35$ MHz. In case of bursts, packets are queued at each bank. This avoids blocking of one bank while another bank is crowded.

### 3.6 Machine size

The design of the **HPP** so far assumes that it has 128 processors as employed in the SB-PRAM. A machine with 512 processors is possible as well. As mentioned earlier, a 3-stage butterfly network can be implemented on one network board. Thus, a 9-stage butterfly network can be implemented without increasing the three stages of network

boards. As the additional network links are all on-board, the memory access latency increases only slightly. 32 vPs per physical processor are still sufficient to hide this latency. The speed of the machine is not affected.

# 4   Applications

Let us first characterize the instruction stream of a physical processor. The behavior of the machine as a synchronous PRAM is determined only by the correct sequence of global instructions for all instruction streams of the vP. Machine instructions can be divided into global and local instructions. Global instructions are those loading from or storing to shared memory. Local instructions are all other instructions. If the compiler achieves that in each request slot (time for one request to shared memory) at most one global instruction is scheduled, the run time can be reduced by a factor of $r$, where $r$ is the number of instructions executed in one request slot. The code of irregular applications typically exhibits only 10% global instructions. Local variables and stacks, for instance, can be held local. However, the delay slot for load instructions can introduce additional dependencies.

The instruction stream of a vP is a trace through the basic block graph during run time. The ratio of the total number of instructions to the number of global instructions gives an upper bound for the improvement that can be achieved by speeding up the execution time of a local instruction. Because we want to make a worst case analysis, we consider the worst case ratio which can be found in any possible trace in the basic block graph. The calculation of this worst trace is straight forward through a two pass analysis of the machine program. Because the machine model is a synchronous PRAM no additional simulations are necessary.

The network can be accessed with a request rate of $f = 31.2\,\text{MHz}$ (see section 3.3).

The peek floating point performance $P$ in inner loops with $i$ local instructions and $c$ global instructions is given by $P = p \cdot f \cdot n/x$, where $p$ is the number of physical processors, $n$ is the number of floating point operations, and $x = \max c, (i + c)/r$. In the sequel we analyze the performance of four applications, two simple numerical loops and two inner loops of irregular applications.

## 4.1   Numerical Applications

The inner loop of the machine code for the *dot product* of a row vector of a matrix with a column vector of the same or another matrix consists of six instructions. The length of the vectors must be known at compile time. Two of the instructions count as floating point operations. This results in a performance of approximately 341 MFlop/s on the SB-PRAM prototype. Two of the instructions are global load instructions. Without software pipelining the block can be executed in three request slots, i.e., two load instructions and one delay slot. So the peak performance on matrix multiply on the 128-**HPP** will be close to 2.66 GFlop/s for $4096 \times 4096$ matrices or for a set of smaller matrices allowing enough parallelism. A 512-**HPP** would yield 10.64 GFlop/s for $16384 \times 16384$ matrices, or an appropriate set of smaller ones.

As a more complex example of numerical code, we chose an *indexed dot product*. A row of an index matrix is used to address a row of a data matrix and a column of another index matrix is used to address a column of the same or another data matrix. The translation of a C routine to machine code for the body of the loop is straight

forward as well. There occur additional load instructions as well as more complex index calculations in the basic block. The inner loop of the indexed dot product has eleven machine instructions. Two of them count as floating point operations. This results in a performance of approximately 186 MFlop/s on the SB-PRAM. Four of the instructions are global load instructions. Without software pipelining the block can be executed in five request slots. So the peak performance on such a routine, which covers a large set of often used inner loops in scientific programming, will be close to 1.6 GFlop/s on the 128-**HPP**, provided there is sufficient parallelism to explore. A 512-**HPP** will yield 6.38 GFlop/s.

If one includes software pipelining, a more sophisticated compiler can reduce the number of request slots to two for the dot product and to four for the indexed dot product, thus almost 4 respectively 2 GFlop/s seem to be achievable on a 128-**HPP**. This increases to 16 respectively 8 GFlop/s on a 512-**HPP**. In [2, p. 379] the performance of 19 machines on the Linpack benchmark is listed. The fastest machine in the list is an Intel Paragon XPS with 1872 processors. It achieves a performance of 36.45 GFlop/s at a matrix size of 17500. Hence, the performance per processor is 19.5 MFlop/s. On indexed dot product, the 128-**HPP** achieves a performance of $1/128 \cdot 2$ GFlop/s $= 15.6$ MFlop/s per node. There is no performance data available for this loop on other parallel machines.

## 4.2 Irregular Applications

The third example is taken from the SPLASH benchmark suite [11]. We analyzed the inner loop of the parallel *discrete event simulator* as implemented in [7]. In this loop more than 60 percent of the total run time is spent. The basic block graph contains 59 nodes. The worst case ratio of total to global instructions on any trace is about 2.6, i.e., we expect—including the faster clock speed—an improvement of about 10 on the **HPP** compared to the SB-PRAM.

Note, that the achievable speedup for discrete event simulation is strongly limited by the critical path of the circuit being simulated as long as the conservative approach is implemented. Due to the possibility to use very efficient parallel data structures with concurrent access to shared data, more aggressive simulation methods become an interesting and promising research area. If the **HPP** is implemented with more processors than the SB-PRAM, they can be used only effectively if there is enough parallelism to exploit. Concurrent simulation of more than one test pattern seems to be the method of choice, where the representation of the simulated circuit is stored only once in memory. An example are production tests for ASICs that usually consist of 10 to 50 independent groups of patterns.

In [10], the SB-PRAM implementation is compared to a sequential implementation to obtain an absolute speedup. There, for the benchmark circuit Multiplier, both a SUN Sparc 20 and a SGI Challenge need about 12 seconds to simulate the circuit on the input vectors delivered with SPLASH. The SB-PRAM with 16 processors needs about 27 seconds. Then, the **HPP** with 16 processors obtains an *absolute* speedup of $12/27 \cdot 10 = 4.4$. On a 128-**HPP**, $128/16 = 8$ test patterns can be simulated simultaneously. On a 512-**HPP**, this increases to 32. Thus, the speedups rise to 35.5 and 142.2, respectively.

The last example consists of the inner loop of a *ray tracer* [6]. The basic block graph has 83 nodes, the subroutine calls to the function calculating intersection points are not counted. The program spends approximately 80 percent of the run time in this

loop. Any trace through the complex loop reveals a ratio of total to global instructions in the range of 4.5 and 8. No consecutive global instructions appear. So it seems that the proposed improvement of performing three times as much local instructions as global instructions is easy to achieve.

Clearly, exact performance data for the irregular application cannot be provided, but due to the fact that one of the fastest ray tracing methods has been parallelized with almost linear speedup even for a large number of processors, the performance of the **HPP** will be considerably larger than the one of any other parallel machine. In [5] it is shown that the SB-PRAM is seven times faster than an SGI challenge with one $150\,$MHz MIPS R4400 processor. The analysis of the **HPP** promises an $11.7$ times faster version of the ray tracer. Moreover, a 512-**HPP** will be about 327 times faster than a one processor SGI challenge on sufficiently large data bases.

## 5   Conclusions

We presented how to re-engineer the SB-PRAM multiprocessor. Fast ASIC technology and high speed network links allow to increase the request rate from $8\,$MHz to $31.2\,$MHz, i.e., by a factor of $3.9$. A further speed gain was obtained by separating local and global operations: during one global operation several local instructions can be executed. Hence, we get higher instruction throughput while the request rate remains unaltered.

We analyzed the performance both for numerical and irregular non-numerical benchmarks. We observed that the fraction of global to total instructions is low and that there is enough independence so that a compiler can statically schedule instructions without fancy optimizations if we overlap one global with two local instructions. Hence, the processor speed can be increased by a factor of 3 to $93.6\,$MHz. Further speed up might be possible by more aggressive dynamic scheduling, i.e., by using a superscalar, out-of-order issuing processor.

The peak performance of the resulting machine is $93.6 \cdot 128 \approx 12\,$GFlop/s. As the same architecture can be used with 512 processors, this increases to $48\,$GFlop/s. We guessed the performance of the benchmarks by inspecting the compiler generated assembler code of their kernels. For indexed vector operations, we obtain a performance of $15.6\,$MFlop/s per processor, thus approaching the $19.5.\,$MFlop/s per processor of an Intel Paragon on Linpack type applications. Indexed vector operation show on any parallel machine normally very poor performance.

For circuit simulations, we achieve an absolute speedup over a SUN Sparc 20 or a one processor SGI Challenge of 35 and 140 with 128 and 512 processors, respectively. For ray tracing, we achieve absolute speedups of 81 and 327 over a one processor SGI Challenge with 128 and 512 processors, respectively. Thus, for large data bases which cannot be duplicated to each processor **HPP** shows superior performance compared to any other parallel architecture.

Further improvements which have not been discussed in the paper might be thinkable. The network bandwidth can be doubled if we allow complete requests to be transmitted at once. In the current solution the amount of transceiver circuits and hence the total area per link would be doubled. This seems to be impossible to implement on one board. The problem of the enormous amount of external circuits can be overcome if we connect the serial line directly to the chip. Referring to the S3 project at Sun Microsystems [9], this is possible at a transmission frequency of $1\,$GHz. The amount of chip

area needed for each channel is $1\,mm^2$. Another project that deals with optical busses connected directly to a chip is IBM OETC/SCI Link [4] and the announced follow-up Jitney. Because the pin limitations are dropped in this case, we can choose a more suitable master which exploits the network chip better. For every link 12 serial lines are needed. As a chip has four links this totals to 48 serial lines respectively interface pins. With higher transmission rates on optical links, this number can be reduced. Then, more network nodes can be integrated in one chip both decreasing network latency and shrinking the machine.

## References

1. F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.
2. G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 2nd edition, 1994.
3. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM, 1990.
4. D. Engebretsen, D. M. Kuchta, R. C. Booth, J. D. Crow, W. G. Nation. *Parallel Fiber-Optic SCI Links. IEEE Micro*, pages 20–26, February 1996.
5. A. Formella. Ray Tracing Complex Scenes: Parallel or Sequential? In *Proceedings of $7^{th}$ IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 89–92, October 1995.
6. A. Formella and C. Gill. Ray Tracing: A Quantitative Analysis and a New Practical Algorithm. *The Visual Computer*, 11(9):465–476, December 1995.
7. J. Keller, Th. Rauber, and B. Rederlechner. Conservative Circuit Simulation on Shared–Memory Multiprocessors . In *Proc. 10th Workshop on Parallel and Distributed Simulation*, Philadelphia, USA, May 1996.
8. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
9. A.G. Nowatazyk, M.C. Browne, E.J. Kelly, and M. Parkin. S–connect: from networks of workstations to supercomputer performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 71–82, 1995.
10. B. Rederlechner. Parallele Diskrete Ereignissimulation auf der SB-PRAM. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1996.
11. J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.
12. L.G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 943–971. Elsevier, 1990.

This article was processed using the LATEX macro package with LLNCS style