# HPP
# A HIGH PERFORMANCE PRAM

**Arno Formella** **Jörg Keller** **Thomas Walle**

02/1996

Fachbereich 14 Informatik
Universität des Saarlandes
Postfach 151150
66041 Saarbrücken
Germany

SFB 124, D4

# HPP: A High Performance PRAM

Arno Formella Jörg Keller* Thomas Walle
Universität des Saarlandes, FB 14 Informatik
Postfach 151150, 66041 Saarbrücken, Germany

**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 optical links shall run at a speed of 96 MHz. To achieve this speed, we first investigate the re-design of ASICs and network links. 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 and 8 GFlop/s shall be possible with 128 and 512 processors, respectively, thus approaching processor performance of Intel Paragon XPS. As non-numerical codes we consider circuit simulation and raytracing. We achieve speedups over a one processor SGI challenge of 35 and 81 for 128 processors and 140 and 325 for 512 processors.

**Keywords:** shared memory multiprocessor, parallel random access machine, multithreading, latency hiding, irregular applications

## 1 Introduction

Parallel systems are used for the solution of computationally intensive problems. While numerical problems are mostly regular and thus suited for distributed memory machines (DMM's), other problems are often irregular. Examples are simulations or raytracing. For those problems, shared memory multiprocessors (SMM) are more suitable as they do not require partitioning in order to minimize replication and avoid expensive message passing.

Current massively parallel SMM's such as KSR1 or Stanford DASH are NUMA (non–uniform memory access) machines [3]. The large variation in memory access time requires careful tuning of applications to obtain expected performance. Tuning often requires partitioning to exploit locality and to avoid false sharing of cached pages, and thus leads to problems similar to programming DMM's (see [3, p. 610–611]).

These problems are avoided in a UMA (uniform memory access) machine, in theoretical computer science also known as a PRAM (parallel random access machine). Bus-based UMA machines have been built for small numbers of processors, e.g. Sequent symmetry. There are several approaches to simulate a PRAM with many processors on more realistic multiprocessors (see [19] for a survey). One of these simulations, Ranade's Fluent Machine

*Supported by a DFG Habilitation Fellowship.

[15], has been turned into a parallel architecture called SB–PRAM [1]. Key concepts are avoiding hot spots by universal hashing, implementing concurrent access by combining, hiding latency by synchronous multithreading with hardware support for multiple contexts. Furthermore, parallel prefix computations without serialization are supported.

We investigate how the SB–PRAM can be made faster in order to be competitive to the speed of DMM's. To do this, we first consider hardware improvements, i.e., the re-design of network chips and the use of optical links. We then investigate the influence of allowing multiple local instructions per global instruction. We present the necessary hardware requirements and explore to which extent applications can exploit this feature without expensive compiler optimizations. We use both numerical and irregular non-numerical benchmarks.

Our main 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. In inner loops, numerical codes achieve a performance of 8 to 16 GFlop/s on a 512 processor **HPP**, 2 to 4 GFlop/s on a 128 processor **HPP**. Thus for codes similar to Linpack (with matrices of size 16000) we obtain a performance of 15.6 MFlop/s per processor, approaching the performance of the Intel Paragon XPS with 19.5 MFlop/s per processor[1]. For circuit simulations, speedups of 140 over a SUN Sparc20 or a one processor SGI Challenge are possible with a 512 processor **HPP**, if several input vectors are simulated simultaneously. For raytracing, speedups of 325 over a one processor SGI Challenge are possible with a 512 processor **HPP**. For 128 processors, these speedups are 35 and 81, respectively.

A project with similar goals is the Tera computer [4], now marketed as Tera MTA. Tera directly targeted leading edge technology, e.g. a GaAs processor with a 4 ns cycle time should 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 does not have local memory at processors, 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 [2] is a massively parallel multiprocessor architecture with $p$ processors providing users with a virtual shared memory. It is based on Ranade's Fluent Machine [15].

[1] In [3, p. 379], an Intel Paragon XPS with 1872 processors is reported to obtain a performance of 36.45 GFlop/s on Linpack with a matrix size of 17500. This machine was the fastest among all machines listed.

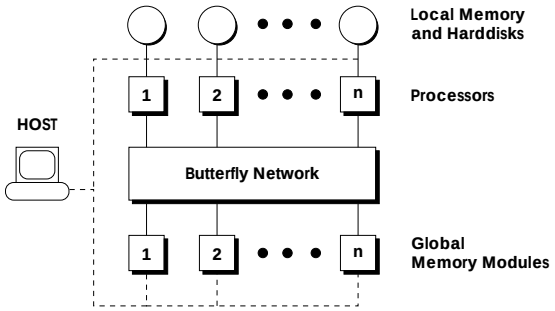


Figure 1: SB–PRAM Architecture

## 2.1 Architecture

As in all massively parallel shared memory machines, the global memory of the SB–PRAM is physically distributed among $p$ memory modules. Memory requests are transmitted between processors and memory modules via an interconnection network, see Fig. 1.

Machines such as KSR1 [3] or Stanford DASH [12] realize the shared memory by caching remote data and keeping the caches coherent (cf. e.g. KSR1's Allcache Protocol). This leads to large variations in memory access time which makes performance prediction and tuning difficult [11].

In contrast, the SB–PRAM uses universal hashing to distribute addresses among the memory modules in a random fashion, every memory access is remote. The 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.

The interconnection network is a butterfly network. Network latency is $c \log p$ cycles, 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.

## 2.2 Prototype

The SB–PRAM prototype [1] consists of $p = 128$ physical processors and the same number of memory modules[2]. Each physical processor implements $v = 32$ virtual processors 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 virtual processors are held off–chip

[2]Currently, a 4-processor version is running, the complete machine is still under construction.

in a fast static RAM. The processor runs at a speed of 8 MHz, which is determined by the speed of the interconnection network, as we will see.

The sorting device needed to inject requests into the network 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 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.

Network links split requests 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 each 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 one cable consisting of 100 wires, 59 for signals and 41 for 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 [18]. It uses about 70k gates and has 192 signal pins. The critical path is caused by the 32-bit integer ALU. A worst case analysis determined the maximum clock frequency to be 32 MHz. To estimate today's maximum frequency we compare different technology levels

| manufacturer | technology ($\mu m$) | typ. delay (ps) |
|---|---|---|
| Motorola HDC | 1.0 | 530 |
| Motorola H4C | 0.7 | 365 |
| Motorola M5C | 0.5 | 240 |
| Thesys THA1008 | 0.8 | 450 |

Table 1: Typical Delays of 2-Nand Gates with Fanout 2 for various Technologies.

from some manufacturers. Referring to Table 1 we estimate 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. From the three values of Motorola technologies we extrapolate the $0.35\mu m$ technology to be about 1.4 faster than the M5C technology. Thus, an overall speedup of about 2.6 will be possible, i.e., a today's routing chip would run with an internal clock speed of $2.6 \cdot 36 = 93.6\,\mathrm{MHz}$. The cost of the chips would not increase dramatically while adhering to standard CMOS technology (in contrast to even faster technologies such as ECL, GaAs or full custom design).

The second point that is critical for the chip speed are input and output times. If we make chip I/O independent of the inner computation as shown in Fig. 2 we get the following values for input and output times.

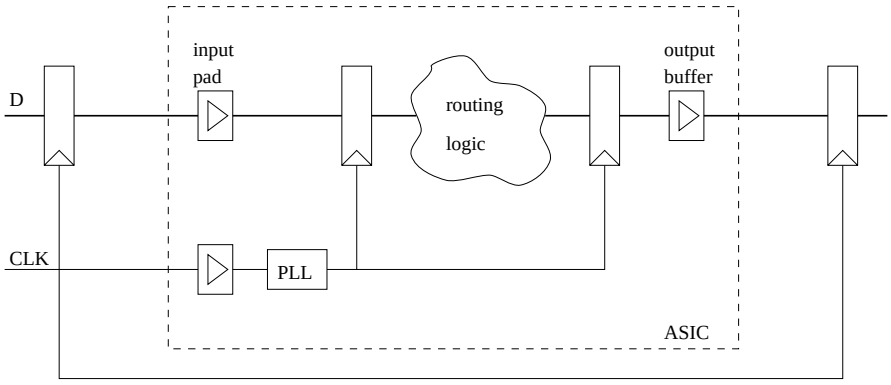


Figure 2: Input and output circuit

The access time to the external register is 4 ns, the typical delay of the 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. 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 internal register access time is 3 ns, the output driver delay is 4 ns if we assume a capacitive load of 20pF. 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, also allowing for a frequency of 100 MHz. As these times should not increase for better technologies, the timing of the chip I/O is not critical.

For our current ASIC the number of signal pins was a limiting factor due to the cost of a

bigger package. Thus, in forward direction we multiplexed 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 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. First, this leads to a complicated external logic. Three different clocks have to be generated for the registers. Furthermore, a multiplexer circuit has to be added. Both implementations with multiplexers as well as with drivers have decisive disadvantages. The IC's as well as the additional connectors need an enormous amount of area. Second, the additional logic increases the external propagation delay (see Fig. 2). Third, 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 the transmission via optical links. Actually a transceiver circuit with 1 GBit/s bandwidth is available [8, 9]. Up to a frequency of 66 MHz a word of 20 bits can be transmitted in parallel. The area needed on each side is about[3] 1836 $mm^2$. Here too, we have to apply two multiplexed cables for one link. But differently 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. 3).

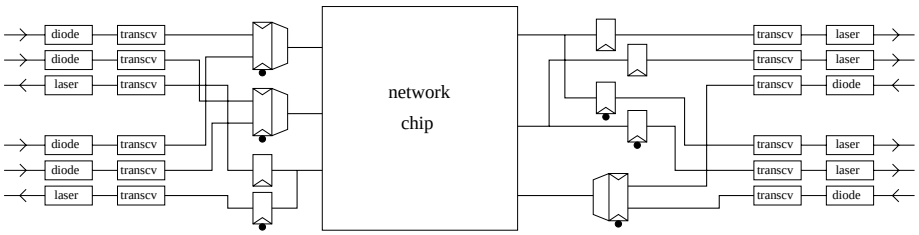


Figure 3: Two multiplexed optical channels with 3 wires each

If we take into consideration that optical cables with multiple optical wires are available, it seems possible to achieve the desired bandwidth with optical links with the same logical behavior of the network as with flat cables.

### 3.3 Processor

If we run the network with 93.6 MHz and need two cycles to handle a request, the processor must be able to run at $93.6/4 = 23.4$ MHz. This is not a limitation for internal speed.

[3] package dimensions: HDMP 1012/1014 (17,2mm x 23,2mm), LSC4110 (30,15mm x 12,75mm).

The processor's I/O needs are as follows: in every cycle it does an instruction fetch, i.e., it outputs a 32 bit address and reads a 32 bit instruction. In addition, it might send a request (70 bit) and receive an answer to a request (32 bit). This is not a limitation as it requires 166 signal pins at 23.4 MHz, a requirement that is met by the network chip. Commercial processors allow even higher rates, e.g. the DEC Alpha has a 128 bit data bus and 32 bit address bus at a speed of 50 MHz [5].

The processor can even run faster by using the following observation. In an application, the total number of store instructions is not larger than the total number of load instructions (provided that the input is read from and the output is written to a disk.) The network provides enough bandwidth to send two store requests in two instructions, each request consisting of two parts. On average, however, a processor will only send a load request and a store request, consisting of only three parts together. Hence, the bandwidth will suffice on the average if the processor runs faster by a factor of 4/3, achieving a speed of 31.2 MHz. We call this frequency the *request rate*. The time between two requests is called *request slot*. As load instructions are delayed, answers to loads are available after two request slots.

Our assumption only holds if there are no "bursts" of store instructions. The only case where we observed those in our benchmarks (see next section) were sequences of push instructions at the entry of functions. These push's however are handled in processors' local memories and do not need network access. Even if short bursts occur, these would utilize $(4/3) \cdot 50\,\% = 67\,\%$ of the network bandwidth. This can be tolerated if it does not last too long.

### 3.4 Commercial processors

If we switch to the technology of commercial processors, it should be possible to implement the SB–PRAM processor in a single chip. First, it should be possible to put the register sets on–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.

Second, the sorting device can be implemented on the processor chip. This would not even increase the pin count. The speed of the sorting algorithm is determined by a critical path through an ALU. Thus, the speed can be similar to the speed of the network chip.

Third, we can run the processor faster by a factor $r$, where $r$ is an integer. We must ensure that in a request slot, which now contains $r$ instructions, only one instruction accesses the global memory. Thus, the request rate is not altered. Furthermore, a delayed load by $2r - 1$ instructions must be tolerated, because the answer to a load request is available after two request slots. Finally, "bursts" of load instructions must not happen. Note, that this improvement will only work if the fraction of global instructions does not exceed $1/r$.

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, the speed of the network. The internal processor speed is still below the actual numbers for commercial microprocessors. As we have 32 pipeline stages because of the virtual processors, even complex floating point operations are as fast as integer operations. However, we have to take care of chip I/O and instruction memory.

Chip I/O can be brought down to a speed of $93.6/2 = 46.8$ MHz by doubling all busses and the use of alternate busses for virtual processors with odd and even numbers, respec-

tively. 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 virtual processors run synchronously, the on-chip cache will be large enough to deliver almost all instructions at the requested speed. If the virtual processors run asynchronously, e.g. if each operates on a different application, the size of the instruction cache is too small to serve 32 virtual processors. They can be served by the second level cache which is large enough, but this will slow down the machine by a factor of 3 to 4. This slowdown can be avoided by carefully assigning virtual processors to applications. If the virtual processors 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 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 [14], which is fast dynamic RAM with on chip cache. The EDRAM is available in a 1M×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 because another bank is crowded.

### 3.6 Machine size

The design of the **HPP** so far assumes that it has 128 processors as in the SB–PRAM. A machine with 512 processors is possible as well. As mentioned in Section 2.2, 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. Thus 32 virtual processors per physical processor are still sufficient to hide this latency. The speed of the machine is not affected.

## 4 Applications

Before we discuss some application programs, we analyze the instruction stream of a physical processor. The machine instructions of the SB–PRAM can be divided into global and local instructions. Global instructions are those loading from or storing to global memory. Local instructions are all other instructions. 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 virtual processors. If the compiler achieves that in each request slot just 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. In assembler programs of irregular applications, typically only 10% of all instructions are global. Note, that local variables and stacks are held in local memory. Because a delay slot is implemented for global load instructions, one has to take care of true-dependencies in a schedule both between two global instructions and between one global and following local instructions.

The instruction stream of a virtual processor is given by a trace through the basic block graph during run time. The ratio of the 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.

The network can be accessed with a request rate of $f = 31.2\,\mathrm{MHz}$ (see section 3.3). As we count only the number $c$ of global memory accesses (load and store operations), the peek floating point performance $P$ in inner loops is given by

$$P = p * n/c * f$$

where $p$ is the number of physical processors and $n$ is the number of floating point operations.

We present four examples. Two of them are simple numerical loops, namely the dot product as base for matrix multiplication and an indexed dot product as base for many complex address patterns in numerical applications. The other two examples are inner loops of irregular applications, namely discrete event simulation and raytracing, which usually exhibit poor performance on distributed memory machines.

### 4.1 Dot Product

The following C routine computes the dot product of a row vector of a matrix with a column vector of the same or another matrix. The length of the vectors has been set to a constant value.

```
Real DotProduct(Real *row, Real *col)
{
  Real sum=0.0;
  int  i;

  for(i=MSIZE;i;i--) {
    sum += *row++ * *col;
    col += MSIZE;
  }
  return(sum);
}
```

In assembly language for the SB–PRAM, the body of the loop may look like the following sequence. The syntax of the mnemonics is straight forward. The operands are given in the order: source, source, destination. We have chosen the same symbolic names as in the C routine instead of introducing register names. An instruction `popgn R1, I, R2` loads the value at address `R1+I` from global memory into register `R2` and increments `R1` with the immediate value `I`. Clearly, the registers must be preloaded with the appropriate values before entering the body of the loop.

```
        popgn   col, MSIZE, tmp1
L5:     popgn   row,     1, tmp2
        sub       i,     1,    i
        fmul   tmp2,  tmp1, tmp2
        popgn   col, MSIZE, tmp1
        fadd    sum,  tmp2,  sum
        bnz      L5
```

The inner loop of the dot product has six instructions. Only two count as floating point operations. This results in a performance of approximately 341 MFlop/s on the SB–PRAM prototype as described in section 2.2. Two of the instructions are global load instructions. There exists a dependency (`tmp2`) between line 1 and line 3 of the loop. Hence, 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 **HPP** will be close to 2.66 GFlop/s for $4096 \times 4096$-matrices or a set of smaller matrices allowing enough parallelism. A 512-**HPP** would yield 10.64 GFlop/s for $16384 \times 16384$-matrices.

### 4.2 Indexed Dot Product

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 following C routine shows the product with pointer arithmetic.

```
Real IndexDotProduct(Real *row, Real *col, int *i, int *j)
{
  Real sum=0.0;
  int  k;

  for(k=MSIZE;k;k--) {
    sum += *(row+*i++) * *(col+(*j * MSIZE));
    j   += MSIZE;
  }
  return(sum);
}
```

The translation of the C routine to machine language 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 following assembly sequence uses single assignment strategy for temporary values in registers, clearly, this register need can be reduced.

```
        popgn     i,     1, tmp1
L5:     popgn     j, MSIZE, tmp2
        add     row,  tmp1, tmp3
        ldgn   tmp3,     0, tmp4
        mul    tmp2, MSIZE, tmp5
        add     col,  tmp5, tmp6
        ldgn   tmp6,     0, tmp7
        sub       k,     1,    k
        fmul   tmp4,  tmp7, tmp8
        popgn     i,     1, tmp1
        fadd    sum,  tmp8,  sum
        bnz      L5
```

The inner loop of the indexed dot product has eleven instructions. Only two 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 **HPP**, provided there is sufficient parallelism to explore. A 512-processor version will yield 6.38 GFlop/s.

With 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 processor **HPP**. This increases to 16 and 8 GFlop/s on a 512 processor **HPP**. In [3, p. 379] the performance of 19 machines on the Linpack benchmark is listed. This performance is comparable to the performance on indexed dot product. The fastest machine 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 **HPP** achieves a performance of $1/128 \cdot 2\,\text{GFlop/s} = 15.6\,\text{MFlop/s}$.

### 4.3 Circuit Simulation

The third example is taken from the SPLASH benchmark suite [17]. We analyzed the inner loop of the parallel discrete event simulator as implemented in [10]. 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 consist of 10 to 50 independent groups of patterns, depending on the size of the ASIC.

In [16], 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 processor **HPP**, $128/16 = 8$ test patterns can be simulated simultaneously. On a 512 processor **HPP**, this increases to 32. Then the speedups rise to 35.5 and 142.2, respectively.

### 4.4 Raytracing

The last example consists of the inner loop of a raytracer [7]. Here, the basic block graph has 83 nodes, where the subroutine calls to the function calculating intersection points are not counted. The program spends most of the run time (more than 80 percent for large scenes) 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. In the assembly code 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 raytracing methods has been parallelized with almost linear speedup even for a large amount of processors, the performance of the **HPP** will be considerably larger than the one of any other parallel machine. In [6] it is shown that the SB–PRAM prototype is seven times faster than an SGI challenge with one 150 MHz MIPS R4400 processor. The analysis of the **HPP** promises a 11.7 times faster version of the raytracer. Moreover, a 512 processor machine will be about 325 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. The request rate[4] was increased from 8 MHz to 31.2 MHz, i.e., by a factor of 3.9. The main changes were the use of fast ASIC technology for routing nodes and of optical network links. 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 verified this approach with several benchmarks, both numerical and irregular non-numerical ones. Our first observation is that we find enough local instructions to fill in: the fraction of global to total instructions is low because stack operations are local. Second, there is enough independence that a compiler can statically schedule instructions without fancy optimizations if we overlap one global with two local instructions. This increases processor speed by a factor three to 93.6 MHz. To be more flexible and possibly obtain further speed gains, we could switch to dynamic scheduling, i.e., by using a superscalar, out-of-order issuing processor.

[4] Number of global memory accesses per processor per time unit.

The peak performance of the resulting machine called **HPP** thus 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 linpack type applications, we obtain a performance of 15.6 MFlop/s per processor, thus approaching the 19.5. MFlop/s per processor of an Intel Paragon which was rated best in [3]. For circuit simulations, we achieve an absolute speedup of 35 and 140 over SUN Sparc 20 or one processor SGI Challenge with 128 and 512 processors, respectively. For raytracing, we achieve absolute speedups of 81 and 325 over a one processor SGI Challenge with 128 and 512 processors, respectively.

A further increase in speed is possible if we enhance the request rate of the machine. For instance, a factor two of network bandwidth is possible 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 needed 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 [13], this is possible at a transmission frequency of 1 GHz. The amount of chip area needed for each channel is $1\,mm^2$. Because the pin limitations are dropped too now, 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 of optical links, this number can be reduced. Then, more network nodes can be integrated in one chip. This decreases network latency. Furthermore the machine becomes smaller.

## References


[1] Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.

[2] Ferri Abolhassan, Jörg Keller, and Wolfgang J. Paul. On the cost–effectiveness of PRAMs. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9. IEEE, December 1991.

[3] George Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, 2nd edition, 1994.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM, 1990.

[5] Digital Equipment Corporation, Maynard, Mass. *DECchip 21064 and DECchip 21064A AlphaAXP Microprocessors: Hardware Reference Manual*, 1994.

[6] Arno Formella. Ray Tracing Complex Scenes: Parallel or Sequential? In M.H̃. Hamza, editor, *Proceedings of $7^{th}$ IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 89–92, October 1995.

[7] Arno Formella and Christian Gill. Ray Tracing: A Quantitative Analysis and a New Practical Algorithm. *The Visual Computer*, 11(9):465–476, December 1995.

[8] Hewlett Packard, Geneva, Switzerland. *HDMP-1012/1014 Low Cost Gigabit Rate Transmit/Receive Chipset*, 1994.

[9] Hewlett Packard, Geneva, Switzerland. *LSC4110-1 mW 14 Pin DIL Cooled Laser Module*, 1995.

[10] 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.

[11] Alexander C. Klaiber and Henry M. Levy. A comparison of message passing and shared-memory architectures for data-parallel programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[12] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[13] Andreas G. Nowatazyk, Michael C. Browne, Edmund J. Kelly, and Michael 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.

[14] Ramtron International Corporation, Colorado Springs, CO. *Specialty Memory Products Databook*, October 1994.

[15] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[16] Bernd Rederlechner. Parallele Diskrete Ereignissimulation auf der SB-PRAM. Diplomarbeit, Universität des Saarlandes, FB Informatik, 1996.

[17] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[18] Thesys GmbH, Erfurt, Germany. *THA 1008 Macro Cell Databook, Rev. 2.0*, April 1995.

[19] Leslie G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 943–971. Elsevier, 1990.