NAS Integer Sort on Multi–Threaded Shared Memory Machines

Thomas Grün[1] and Mark A. Hillebrand[1]

(gruen@cs.uni-sb.de, mah@studcs.uni-sb.de)
Computer Science Department, University of the Saarland
Postfach 151150, Geb. 45, 66041 Saarbrücken, Germany

Abstract. We investigate the performance of multi–threaded shared memory machines, like the commercial Tera MTA or the experimental SB–PRAM, on the Integer Sort benchmark of the NAS Parallel Benchmark Suite. The benchmark models the ranking phase of a counting sort algorithm, i.e. it computes for each element x of a vector of small integers the position of the first occurrence of x in the sorted version of that vector. Both aforementioned machines require for the execution of the benchmark about three times less clock cycles than vector computers and an order of magnitude less cycles than general purpose DMMs (distributed memory machines) or SMMs (shared memory machines). The reasons for this behavior are investigated. It turns out that both processors can take advantage of a fetch–and–add operation and that due to multi–threading no time is lost waiting for memory accesses to complete.

Key Words: rank sort, multi–threading, fetch–and–add, shared memory computers, Tera MTA, SB–PRAM

1 Introduction

One of the first programs that ran on a node processor of the Tera MTA (multi–threaded architecture) [5, 4] was the Integer Sort (IS) from the Parallel Benchmark suite [8, 2] of the Numerical Aerospace Simulation Facility (NAS) at NASA Ames Research Center. According to a press release of Tera Corporation [16], a single node processor of the Tera machine was able to beat a one–processor Cray T90, which hitherto held the record for one processor machines, by more than 10 percent. A closer look on the Tera implementation reveals that the Tera program has implemented more functionality than the benchmark description requires. If only the benchmark requirements had been implemented, the Tera MTA were faster than the Cray by a factor of three. The SB–PRAM [3, 12, 9] has many features in common with the Tera MTA, although it is inspired by a totally different idea. Unlike with the Tera MTA, which is targeted at high performance numerical computing, the basic goal in the SB–PRAM design was to implement the PRAM model (parallel random access machine) from theoretical computer science [13]. The design is based on Ranade's Fluent machine [15] with some extentions like multi–prefix operations, which is the implementation of a "fetch–and–add" operation known from the NYU Ultracomputer [11]. The current 4–processor prototype [6] will be extended to 64 processors as a proof of concept, but due to budget limitations of a university project, the machine will not offer competitive performance. However, it is described in [10] how a High–Performance–PRAM (HPP) with a tenfold performance could be built using

* This work was partly supported by the German Science Foundation (DFG) under contract SFB 124, TP D4.

1995 standard CMOS technology and modest architectural changes. The above two machines differ from most of todays shared memory parallel computers in two aspects: First, they implement the UMA (uniform memory access) model instead of the NUMA (non uniform memory access) scheme found in todays scalable shared memory machines like SGI Origin, Sun Starfire or Sequent Numa-Q. In other words, they do not employ data caches with a cache coherence protocol, but hide memory latency by means of multi–threading. Second, they provide special hardware for fetch–and–add instructions or full/empty flags (only Tera MTA) in the memory system. Commercial microprocessors, which are employed in most of todays parallel computers do not offer this kind of hardware support for running efficient parallel programs.

The IS benchmark is part of the NAS (Numerical Aerospace Simulation Facility) Parallel Benchmark Suite (NPB) [8]. It models the ranking step of a counting sort (kind of bucket sort) application, which occurs for instance in particle simulations. The IS benchmark takes a list L of small integers as an input and computes for every element $x \in L$ the rank $r(x)$ as the number of $k \in L, k < x$. On this benchmark, the best program on a 4–processor SB–PRAM spends 3.3 cycles per element (CPE) on the average, and the single–node Tera machine needs 2.7 cycles. Both machines are expected to scale without significant loss of efficiency. Most of todays general purpose parallel computers require more than 30 CPE if at least 4 processors are working, and only vector computers like the Cray T90 or the Fujitsu VPP700, approach ten cycles per element. In this article, we describe how the IS benchmark was implemented on the SB–PRAM. We show, how easy and intuitive it is to implement the IS program using the PRAM programming model. A series of optimizations that lead to the code with the best performance will be described and evaluated. A lower bound on the theoretical peak performance of the SB–PRAM will be derived and compared to the performance of the actual code. Then, we investigate the performance of the HPP, the Tera MTA, and explore why todays commercial parallel computers are significantly slower than the aforementioned machines. In this process, we outline which modifications of the simple PRAM code are necessary for the different machines.

The paper is organized as follows. Section 2 discusses the IS benchmark specification and Section 3 describes the structure of the SB–PRAM computer in detail. In Section 4, a series of increasingly powerful benchmark implementations on the SB–PRAM is presented and analyzed. Section 5 evaluates which performance could be achieved with the High–Performance–PRAM. Section 6 describes the algorithmic changes for the Tera MTA and the resulting performance. Section 7 lists the performance of other parallel computers, contrasts their results with the SB–PRAM and Tera figures, and investigates why todays commercial parallel computers are significantly slower. Section 8 is the summary.

2 The NAS Integer Sort Benchmark

Despite its name, the NAS Integer Sort program does not really sort a list of small integer values, but it ranks them by assigning every list element a number indicating its position in the sorted list. The benchmark is parameterized and scalable, i.e., there are four classes which define the parameters that control the memory consumption and the run time of the benchmark. The benchmark specification is listed in Figure 1 and the set of constants defining the classes are listed in Table 1.

The random numbers for the keys are generated in two steps. First, a linear congruence method produces uniformly distributed pseudo random numbers r_i using the following recursive algorithm: $r_0 = 314159265, \quad r_{i+1} = 5^{13} \cdot r_i \bmod 2^{46}$. Then,

1) Generate N random keys $K_0, \ldots, K_{N-1}$ with $K_j \in \{0, \ldots, B_{max} - 1\}$
   using the specified random number generator
2) Begin timing
3) Do, for $i = 1$ to 10
   a) Modify the sequence of keys:
      $K_i \longleftarrow i, \quad K_{i+10} \longleftarrow (B_{max} - i)$
   b) Ranking
      Compute $rank : \{0, \ldots, N-1\} \longrightarrow \{0, \ldots, B_{max} - 1\}$
      so that $rank(j) = \#\{k \mid K_k < K_j\}$.
   c) Partial verification test
      Check the value of $rank(j)$ for a set of specified $j \in \{0, \ldots, N-1\}$.
4) End timing
5) Perform full verification test
   Sort the keys according to their rank and verify that $K_0 \leq K_1 \leq \ldots \leq K_{N-1}$.

Fig. 1. NAS Integer Sort Specification

	Sample	Class A	Class B	Class C
N	2^{16}	2^{23}	2^{25}	2^{27}
B_{max}	2^{11}	2^{19}	2^{21}	2^{23}

Table 1. Constants for the Respective Classes

the keys K_i are computed as $K_i = \frac{1}{4} \cdot \left(\sum_{k=0}^{3} r_{4i+k}\right) \cdot \frac{B_{max}}{2^{46}}$ resulting in a gaussian distribution of the keys.

3 The SB–PRAM Design

In this section, the SB–PRAM design is presented. First the key concepts, which center around the network design, are discussed from a technical point of view. Then, it is shown, how this machine implements the PRAM model from theoretical computer science. Finally, the programming environment and software tools are sketched.

Key Concepts. The SB–PRAM [3] is a shared memory computer with up to 128 processors and an equal number of memory modules, which are connected by a butterfly network. Processors send their requests to access a memory location via the butterfly network to the appropriate memory modules and receive an answer packet if the request was of type LOAD. There are no caches in the memory system. Three key concepts are employed to yield a uniform load distribution on the memory modules and to hide memory latency: (a) synchronous *multi–threading* with a delayed load; (b) *hashing* of subsequent logical addresses to physical addresses that are spread over all memory modules; (c) butterfly network with input buffers and *combining* of packets.

For hiding memory latency, a physical processor (PP), i.e., the processor chip, simulates 32 virtual processors (VP), which have their own register sets in hardware and are scheduled round robin after each instruction (*synchronous multi–threading*). Because a VP must not use the result of a load request in the successive instruction

(*delayed load*), a network and memory latency of 63 CPU clocks can be tolerated. A simple linear *hash function* has proven to be sufficient [7] for distributing memory requests uniformly on all memory modules. If two packets with the same hashed memory address meet in a network node, they are coalesced into one packet, and, in the case of a LOAD, the corresponding answer packet is duplicated on the way back from memory to the processors. This *combining* facility avoids hot spots on a memory module, if many processors address the same cell concurrently.

The design ensures that the VPs are never stalled in practice, neither by network congestion on the way to the memory modules nor by answer packets that arrive too late. Although this issue has been intensively investigated in [7] and [17] using different and detailed simulators, the main criticism on the SB–PRAM is that these investigations used only simulations. Therefore, a 64 processor machine is being built as a proof of concept. At the time of this writing, the re–design of an earlier 4–processor prototype [6] has been completed. The 64 processor machine is expected to be running in summer 1998.

PRAM Programming Model and Multi–Prefix. The SB–PRAM implements a *priority CRCW PRAM*, which is the strongest PRAM model of theoretical computer science [13]. As requested by the PRAM model, the VPs execute a program synchronously, which is guaranteed by the network implementation. The combining scheme is extended not to only merge (respectively duplicate) the packets, but to perform simple operations on the data. The multi–prefix operations (MP) of a processor take advantage of this enhancement. Let a memory cell, addressed by A, contain the value $old(0)$ and let some virtual processors $i = 1, \ldots, n$, numbered according to their priority, execute synchronously the machine instruction `new(i)=MP⊙(A,old(i))` using an operation $\odot \in \{\mathrm{add}, \mathrm{and}, \mathrm{or}, \max\}$. In the network and the memory module, the values $new(j) = \bigodot_{k=0}^{j-1} old(k)$ $(\forall j = 1, \ldots, n+1)$ are calculated by a parallel prefix computation scheme. After the operation is completed, each processor receives $new(i)$ as a result, and $new(n+1)$ is stored in the participating memory cell. The operation is named "multi–prefix", because multiple parallel prefix operations can be performed in parallel on different memory cells.

Although the parallel prefix computation in the network nodes is rather hard to follow up, the programmer need not be concerned with it. He can think of an instruction "`Rx=mpadd(addr,Rx)`" that is executed by some VPs in parallel, as if there stood a sequence "`H=M(addr); M(addr)=M(addr)+Rx; Rx=H`" that was executed by the VPs one after another in ascending order of their priorities. This *sequential semantics* describes the behavior of the machine shortly and correct.

Compiler and Software. The GNU `gcc` compiler has been ported to the SB–PRAM. The C language has been extended by two storage class qualifiers `shared` and `private`, which determine whether a variable can be accessed by all VPs or every VP has its own copy of that variable. The functionality of special SB–PRAM instructions like multi–prefix is provided by inline assembly code, which is encapsulated in "`#define`"–statements so as to facilitate its use by C programmers. Due to space restrictions, we refer to [12, 9] for a more complete discussion of the programming model, the operating system and the simulation environment.

4 SB–PRAM Implementation

In this section a series of increasingly faster algorithms is presented. The first algorithm illustrates the ease of programming on the SB–PRAM. The subsequent opti-

```
<0.0>   shared unsigned keys[N];              /* key array */
<0.1>   shared unsigned hist[Bmax];           /* histogram array */
<0.2>   shared unsigned rank[Bmax];           /* rank array */
<0.3>   shared sbp_sync_barrier_t barrier;    /* barrier synchron. */
<0.4>   shared unsigned int mp_loop;         /* shared loop counter */
<0.5>   shared unsigned int ranksum;         /* accumulated rank */
<0.6>   void nas_is_rank()
<0.7>   { register unsigned int elem;
<1.0>     sbp_sync_barrier(&barrier,PROCS);
<1.1>     mp_loop = 0;
<1.2>     while( (elem=sbp_mpadd(&mp_loop,1)) < Bmax )
<1.3>       hist[elem] = 0;
<2.0>     sbp_sync_barrier(&barrier,PROCS);
<2.1>     mp_loop = 0;
<2.2>     while( (elem=sbp_mpadd(&mp_loop,1)) < N )
<2.3>       sbp_syncadd( &hist[keys[elem]],1);
<3.0>     sbp_sync_barrier(&barrier,PROCS);
<3.1>     ranksum = 0;
<3.2>     mp_loop = 0;
<3.3>     while( (elem=sbp_mpadd(&mp_loop,1)) < Bmax )
<3.4>       rank[elem] = sbp_mpadd(&ranksum,hist[elem]);
<4.0>     return;
<4.1>   }
```

Fig. 2. Simple IS Implementation on the SB–PRAM

mizations address some weaknesses of the current compiler and could be done automatically by an optimizing compiler. A calculation of the peak performance is given at the end of this section.

First Implementation. A straightforward implementation of the ranking step, which is the core of the benchmark, is listed in Figure 2. In addition to the key, histogram, and rank arrays (`keys`, `hist` and `rank`), two other data items `mp_loop` and `barrier` are `shared`, i.e., they may be accessed concurrently by all processes. The latter two variables are used to implement three successive parallel loops: First, the histogram array is cleared; then, a histogram of the key distribution is built, and, finally, the ranks are computed from the histogram.

Contrarily to the shared variables in lines (0.0-0.5), the variable `elem` (0.7) is a register variable, which is private to every process. When a process enters the `nas_is_rank()` function, it first encounters the barrier synchronization in line (1.0). This construct waits until all `PROCS` processes have entered the function and assures that the processes run synchronously at machine instruction level when they leave the barrier.[1] In (1.1) all processes initialize the `mp_loop` variable. This is a good example for the typical CRCW PRAM programming style. Although *all* processors write a zero to the `mp_loop` variable, their packets are combined in the network, and only one of the packets — the one of the processor with the highest priority — reaches the memory module; thus, performance does not suffer from this programming style. The

[1] The barrier synchronization lasts 16 machine cycles if the processes are already synchronous when entering the function.

while–loop in line (1.2) implements a parallel loop on the shared counter `mp_loop`. Every process gets the next loop index by executing `elem=sbp_mpadd(&mp_loop,1)`. If two or more processes issue a multi-prefix add operation in the same round, the network combines these requests also without performance penalty. Lines (2.1-2.2) and (3.2-3.3) implement the same parallel loop mechanism, and the barriers in (2.0) and (3.0) separate the three phases of the ranking algorithm. In (1.3) the histogram array is filled with zeroes. In line (2.3) the histogram is built: the `elem`th element of the `keys` array is read and the appropriate field in the `hist` array is incremented. The operation `sbp_syncadd` is a variant of the multi–prefix add operation in which the return value is discarded, i.e., only the correct update of the involved memory cell is of interest. After this step, `hist[k]` contains the number of occurrences of the key `k` in the `keys` array. In the third phase of the algorithm the rank is computed as

$$\texttt{rank[0]} = 0; \qquad \texttt{rank}[i] = \sum_{k=0}^{i-1} \texttt{hist}[k] \;, \quad \forall i = 1, \ldots, B_{max}.$$

If there were B_{max} processors available, the ranks of all elements could be calculated in one parallel step using the multi–prefix add instruction. Since there are not enough processors, this huge prefix sum computation must be broken down into smaller pieces. The loop (3.3-3.4) accomplishes this task, relying on synchronous execution, which guarantees that the position of a VP with regard to the sequential semantics is the same in both `mpadd` instructions involved. If the VPs were not synchronous — e.g., the VPs were not switched round robin — the position of the VPs in the second `mpadd` would not match the index `elem` that was received in the first `mpadd` instruction. Then, the order of `hist` entries added to `ranksum` would be garbled. The program, as listed above, spends 15.88 CPE on the average. In the next paragraphs, we present optimizations that lead to a more than 4–fold speedup.

Compiler Optimizations. The ad–hoc port of the GNU `gcc` compiler does not support automatic parallelization, nor does it incorporate a detailed machine model of the SB–PRAM. Therefore, the generated code is not optimal. In the next two paragraphs we address some of the problems and discuss programming tricks for getting faster code.

The shared loop counter `mp_loop`, although contributing to code readability, should be replaced by a register variable that is incremented by `PROCS` in each iteration. This change does not affect the program semantics (because of synchronous execution), but it enables the compiler to do some optimizations without the need for analyzing the parallel program with respect to the shared variable. As an example, the addition of the array base address (e.g. `&hist[0]` in loop 1) can be moved from the loop body to the loop initialization. Furthermore, we coalesce the separate `hist` and `rank` arrays into one array `hr`, which saves address computations in loop 3. Altogether, these program modifications reduce the average cycles per element from 15.88 down to 11.25.

Besides reducing loop overhead, the biggest challenge to the compiler is to fill the delay slots. The SB–PRAM has no branch delay slots, which are common in todays microprocessors, but there are delay slots after load instructions, which must be filled with instructions that do not consume the load result. In the IS program three delay slots cannot be filled due to data dependencies in lines (2.2-2.3) and (3.3-3.4). The answer to this problem is *loop unrolling*, which, as a side effect, further reduces loop overhead. Since our `gcc` port does not support automatic loop unrolling, we have manually unrolled the loops in the C source code, and obtained four versions C2, C4, C16, and C64 with up to 64-fold unrolled loops. In the assembly code, the instructions of two loop bodies can be interleaved, because there are no data dependencies between

them. The interleaving technique guarantees that all write delay slots are filled with instructions of the alternate loop body at the cost of having only half the number of registers available in each loop body. Table 2 lists the CPE values for these programs.

loop unroll factor	none	2	4	16	64
corresponding CPE	11.25	7.91	5.28	4.65	4.49

Table 2. Results of Loop Unrolling

Theoretical Peak Performance. We now analyze how many instructions are necessary at least to implement the IS benchmark on the SB–PRAM. Hence, we hand–code the loop bodies in assembly language and neglect loop overhead.

If every VP is assigned a different but contiguous portion of the coalesced histogram/rank array `hr`, a single store instruction with auto–increment is sufficient to clear the histogram array. There exists an assembly instruction equivalent to the C code "`*hr_ptr++=0`", whereby `hr_ptr` is a properly initialized private pointer to the portion of the `hr` array belonging to the specific VP. No data dependencies are present in the histogram computation of loop 2. Hence, all VPs can work independently. Figure 3(a) lists the assembly code for handling two elements; it makes use of the interleaving technique sketched earlier. The first assembly language instruction combines the loading of a key and the incrementing of the key pointer `k_ptr` by 2·`PROCS`. In the second instruction, the pointer `hr_ptr` is computed, and, finally the `syncadd` operation increments the histogram entry. Figure 3(b) lists the assem-

(a):

```
elem1 = M( k_ptr1+=2*PROCS)
elem2 = M( k_ptr2+=2*PROCS)
hr_ptr1 = elem1 + hr_base
hr_ptr2 = elem2 + hr_base
syncadd( hr_ptr1, 1)
syncadd( hr_ptr2, 1)
```

(b):

```
elem1 = M( hr_ptr1+=2*PROCS)
elem2 = M( hr_ptr2+=2*PROCS)
elem1 = mpadd( ranksum, elem1)
elem2 = mpadd( ranksum, elem2)
M( hr_ptr1) = elem1
M( hr_ptr2) = elem2
```

Fig. 3. Interleaved Loop Bodies: (a) Loop 2, and (b) Loop 3.

bly code for the third loop, which also makes use of the pipelining technique. In the first instruction an entry of the unified histogram/rank array is read, and the corresponding pointer `hr_ptr` is adjusted. Then, the accumulated rank is computed in the multi–prefix add on `ranksum`, and the result is written to the `hr` array. If the histogram and rank arrays were not coalesced, the incrementing of the rank pointer (last line) would require an additional instruction.

Loop 2 contributes most instructions, because it is executed `N` times whereas loops 1 and 3 are executed only B_{max} times. The ratio $r = \frac{B_{max}}{\mathtt{N}}$ is the constant $\frac{1}{16}$ for the benchmark classes A, B and C. Thus

$$\mathrm{CPE(ranking)} = \underbrace{3}_{\text{loop 2}} + \; r \cdot (\underbrace{1}_{\text{loop 1}} + \underbrace{3}_{\text{loop 3}}) = 3.25 \qquad .$$

In fact, the benchmark implemented with the above assembly language macros comes close to the optimal CPE value of 3.25 (cf. Table 3). The previous C version is slower by roughly one cycle, because the compiler does not use "load with auto-add of `2*PROCS`" but generates two separate instructions instead.

loop unroll factor	4	16	64
corresponding CPE	4.10	3.46	3.30

Table 3. Program with Assembler Macros and Loop Unrolling

4.1 Absolute Performance and Scalability

Hitherto, we have talked about cycles and neglected technology which determines how long a cycle lasts. Furthermore, we have only stated experimental results of programs running on a 4 processor SB–PRAM. Table 4 summarizes the CPE values of all algorithm variants discussed so far and lists the elapsed wall clock time of the class A benchmark on a SB–PRAM computer with up to 64 physical processors.

algorithm variant	CPE	wall clock time [s]			
	1PP	1PP	4PP	16PP	64PP
First PRAM program	15.88	166.51	41.65	10.49	2.61
Loop and constants	11.25	117.96	29.49	7.38	1.85
C2 (interleaving)	7.91	82.94	20.74	5.19	1.30
C4 (unrolling)	5.28	55.36	13.84	3.46	0.87
C64	4.49	47.08	11.77	2.95	0.74
ASM64	3.30	34.60	8.65	2.18	0.55

Table 4. Summary of Execution Times

Because no caches are employed in the SB–PRAM, the run times of class B and C can be simply and accurately predicted; they are higher by a factor of 4, respectively 16. The execution times scale well: a 64 processor machine encounters less than one percent of performance degradation.

5 High–Performance–PRAM

Due to budget limitations, the current SB–PRAM uses affordable, slow 0.8 μ 2-layer metal CMOS technology for its 3 network and processor semi–custom ASICs. Thus, the network chips are limited to 32 MHz, and, as a result, the processor is clocked at 8 MHz. The HPP, which has been proposed in [10], uses 1995 top CMOS technology, which speeds up the network by a factor 3 and lets the processors issue global memory requests at 31.2 MHz. Although not explicitely stated in [10], probably a further load delay slot must be introduced.

In addition to this technological improvement, the processor is changed so that is can execute two local instructions (computation, branch, or access to a new local

memory) per global memory access. Although the local instructions are executed one after another, the change has the same effect as superscalar processor design. In an average compiler–generated program, there are enough local instructions to keep the processor busy. The HPP has about ten times the performance of the SB–PRAM on typical applications.

Benchmark Performance. The CPE value does not depend on the clock rate of the machine. Thus, an improvement has to come from the local instructions that can be processed in parallel with a global memory access. In the highly optimized assembly code, only the addition in loop 2 can be hidden. The CPE drops from 3.25 to 2.25, which is about a 30% improvement. On the other hand, the C64 version now also achieves that speed. Hence, the IS benchmark would be 5.6 times faster on the HPP than it is on the SB–PRAM.

6 Tera MTA

The design goal of the Tera MTA [5, 4] was to build a powerful multi–purpose parallel computer with special emphasis on numerical programs. In order to meet this goal, it was determined early in the design phase to employ GaAs technology and liquid cooling. A processor chip runs at 300 MHz and has a power dissipation of 6 KW per processor.

Similar to the SB–PRAM, the Tera MTA hides memory latency by means of multi–threading. Unlike the SB–PRAM, it does not schedule a fixed number of VPs round robin, but it can support up to 128 threads, all of which can have up to 8 memory requests outstanding. Thus, up to 1024 cycles of memory latency can be hidden, while the average memory latency is about 70 cycles. New threads can be created using a low overhead mechanism, the penalty for an instruction that attempts to use a register that is waiting for a memory request to arrive is only one cycle. The memory system of the Tera MTA is completely different from that of the SB–PRAM: the network topology is a kind of 3–dimensional torus, where alternate z–planes provide only connections in x–direction, respectively y–direction. If l is the side length of this cube, $p = l^2$ processor modules, $2p$ or $4p$ memory modules and some I/O modules sparsely populate the l^3 nodes of the routing mesh [4]. The messages are processed by a randomized routing scheme that may detour packets if the output link in the right direction is either overloaded or out of order. This scheme provides fault tolerance in an elegant way, because faulty nodes or links may be flagged down. If every processor issues a LOAD request in every cycle, the network utilization is at least 75%. To our knowledge, there is no published information on the performance of the network protocol under heavy load. The machine supports fetch–and–add instructions, but, unlike with the SB–PRAM, the requests are not combined in the network, but serviced at the memory modules. Because the Tera threads may become asynchronous due to race conditions in the network, the machine does not offer the sequential semantics of the SB–PRAM.

The Tera compilers for C, C++, and Fortran 77 parallelize a sequential program automatically; hints from the user can be provided. The compiler detects data dependencies like linear recurrences and generates appropriate parallel code, e.g., using techniques like cyclic reduction.

Benchmark Performance. In the third loop, the Tera MTA computes the prefix–sums by a recursive approach. First, all threads receive a contiguous portion of the `hr` array and compute the local prefix sums of their part. Then, the i-th thread writes the sum

of its data in the i-th position of a second array `reduce[]`, of which the prefix sums are calculated recursively. After the recursion, the i-th thread adds `recursion[i]` to its precomputed elements with a fetch–and–add instruction. This implementation requires per `hr[]` element 3 Tera instructions plus a small logarithmic term. This is not significantly more than the 3 SB–PRAM instructions.

Because the authors have no access to a real Tera MTA, the figures in the rest of this section are derived from [4], where a run time of 1.53 seconds is stated, which corresponds to a CPE of 6.08. This is 14% higher than the theoretical minimum CPE, which is calculated as $5\frac{5}{16}$. However, Tera has implemented more functionality than the benchmark requires, or, at least, more than the serial and parallel sample NPB implementations of IS provide. In addition to the three loops that we have discussed, Tera executes a fourth loop: `for(i=0; i<N; i++) pos[i]=rank[keys[i]]++;` It computes the position of the i–th element of `keys` in the sorted version of the `keys` array. This loop alone accounts for 3 CPE. If this computation was omitted, the minimum CPE drops to $2\frac{5}{16}$. Considering the 14% overhead, we end up with 2.64 cycles per element or 0.66 seconds execution time. This is 3 times faster than a single node of a Cray T916.

7 Comparison with Other Parallel Computers

Table 5 lists some benchmark results [8, 2] of vector computers (Cray, Fujitsu), shared memory (SGI Origin), and distributed memory machines for IS class A.

Machine	clock	minimum			maximum		
Name	[MHz]	#procs	time	CPE	#procs	time	CPE
Fujitsu VPP300	140	1	2.40	4.00	16	1.02	27.24
Cray T916	450	1	2.02	10.83	8	0.38	16.30
Cray J916	100	1	13.75	16.39	8	2.21	21.08
Cray T3E	300	2	16.90	120.88	256	0.20	183.11
SGI Origin 2000	195	1	13.66	31.75	32	1.21	90.00
SGI Power Challenge	90	1	20.00	21.45	8	5.00	42.92
IBM SP2 Wide Nodes	66	8	4.93	31.50	128	0.59	60.32

Table 5. Class A Benchmark Results

The Fujitsu vector computer is fast, both in absolute time and CPE, but does not scale; the 4 processor time is 1.25 seconds (8.33 CPE). The Cray T916 in ECL technology and its CMOS counterpart J916 scale, but impose a limit at 16 processors. The Cray T3E, a 3–D torus network using DEC Alpha processors in the processing nodes, implements a non–cache-coherent shared memory. It scales well and holds the record in absolute run time for this benchmark. The CC–NUMA (cache–coherent non–uniform–memory–access) SMM (shared memory machine) SGI Origin 2000, as well as other DMMs (distributed memory machines), have CPE values of more than 30 if at least 4 processors are used. The reasons for the significantly higher CPE value compared to SB–PRAM and Tera MTA are: (a) the above machines do not have a fetch–and–add instruction, which leads to the elegant realization of loop 2; (b) memory accesses last longer than one cycle if the data item is not found in the cache.

Sample code implementation. The parallel sample code of the IS benchmark makes use of MPI (message passing interface) [1], a popular function library designed to make DMM–style applications portable. In order to avoid synchronization when building the histogram in loop 2, the program partitions and redistributes the `keys` array so that the individual processors handle different intervals of the `keys` domain, and, hence, there are no more data dependencies in loop 2. Before the counting starts, each processor scans a part of the input data and sorts them according to their value in a few buckets. Then, a mapping of buckets onto processors is determined (remember, the keys have a gaussian distribution), and in a all–to–all communication the buckets are sent to their destinations. The communication network of the DMMs limits the speed of this step. Since the buckets are allocated in main memory, and the data is typically moved two times between main memory and the network interface card on DMMs, the redistribution adds at least 6 CPE. On SMMs, which can realize message passing via shared memory, or on vector computers, which can issue even more than one memory access in a cycle and have traditionally a fast memory system, the redistribution may be done faster.

CC–NUMA. Cache–coherent non–uniform–memory–access SMMs, like the SGI Origin, have overcome the argument "SMMs do not scale" by other means than the SB–PRAM or the Tera MTA. While multi–threading relies on having enough parallelism available to keep the processors busy, CC–NUMA machines try to minimize the *average* memory latency by caching, at the risk of running idle on cache misses [14]. A simple consideration shows, that CC–NUMA SMMs should implement rather the DMM–oriented sample code instead of taking the direct approach with locks on the `hr[]` elements in loop 2. We neglect the overhead for updating the histogram, assume that all memory accesses except for `hr[]` are cached, and concentrate only on the average memory latency introduced by accessing `hr[keys[i]]` in loop 2. We assume that on a p–processor machine $\frac{1}{p}$ of the `hr[]` elements are cached on every processor and the miss penalty is 20 cycles (i.e. 100 ns on a 200 MHz computer). Thus, the average memory latency is

$$t = \underbrace{\frac{1}{p} \cdot 1}_{\text{cache hit}} + \underbrace{\frac{p-1}{p} \cdot 20}_{\text{cache miss}} \quad .$$

On machines with more than $p = 16$ processors, t is greater than $18\frac{13}{16}$. Thus, the exchanging of `hr[]` entries between the coherent caches is expensive. If the remaining cycles for the program are taken into account, the CPE is an order of magnitude higher than on multi–threaded SMMs with fetch–and–add.

8 Conclusion

We have investigated the performance of the IS benchmark on the SB–PRAM and the Tera MTA, two scalable SMMs which employ multi–threading to hide memory latency instead of caching like CC–NUMA SMMs. Besides multi–threading, both machines take advantage of a fetch–and–add instruction to access shared data structures conflict–free. These two properties lead to a performance that is an order of magnitude higher compared with other scalable parallel computers, both DMMs and SMMs. The performance gain over vector computers, which do not scale, is more than a factor of 3.

In the course of implementing the IS benchmark we started with a PRAM–style program and showed which modifications should be made so that the SB–PRAM compiler can generate optimal code. The HPP section sketched which improvements are possible due to architectural and technological changes. The Tera section addresses the consequences of lacking synchronous execution and the last section illustrated the fundamental program changes necessary for non–multi–threaded parallel computers.

References

1. MPI Home Page. http:// www.mpi-forum.org/.
2. The NAS Parallel Benchmarks Home Page. http:// science.nas.nasa.gov/ Software/ NPB/.
3. 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.
4. R. Alverson, P. Briggs, S. Coatney, S. Kahan, and R. Korry. Tera hardware–software cooperation. In *Proc. of Supercomputing '97*. ACM/IEEE, San Jose, CA, nov 1997. also available under http:// www.tera.com/ web/ library-hscoop.html.
5. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *1990 International Conference on Supercomputing*, June 1990.
6. P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grün, and C. Lichtenau. Building the 4 Processor SB–PRAM Prototype. In *30thHawaii International Conference on System Sciences*, pages 14–23, January 1997.
7. C. Engelmann and J. Keller. Simulation–based comparison of hash functions for emulated shared memory. In *Proc. PARLE (Parallel Architectures and Languages Europe)*, pages 1–11, 1993.
8. D. Bailey et al. The NAS Parallel Benchmarks. RNR Technical Report RNR-94-007, NASA Ames Research Center, mar 1994. also available under http:// science.nas.nasa.gov/ Pubs/ TechReports/ RNRreports/ dbailey/ RNR-94-007/ html/ npbspec.html.
9. A. Formella, T. Grün, and C.W. Kessler. The sb–pram: Concept, design and construction. In *Draft Proceedings of 3rd International Working Conference on Massively Parallel Programming Models (MPPM-97)*, nov 1997. also available under http:// www-wjp.cs.uni-sb.de/ ~formella/ mppm.ps.gz.
10. A. Formella, J. Keller, and T. Walle. HPP: A high–performance–PRAM. In *Proceedings of the 2nd Europar*, volume II of *LNCS 1124*, pages 425–434. Springer, August 1996.
11. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, C–32(2):175–192, feb 1983.
12. T. Grün, T. Rauber, and J. Röhrig. The Programming Environment of the SB–PRAM. In *Proceedings of 7^{th} IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 504–509, Washington, D.C., October 1995. Acta Press.
13. R.M. Karp and V.L. Ramachandran. *Handbook of theoretical computer science*, volume A, chapter A survey of parallel algorithms for shared–memory machines, pages 869–941. Elsevier, 1990.
14. D.E. Lenoski and W.-D. Weber. *Scalable Shared–Memory Multiprocessing.* Morgan Kaufmann Publishers, 1995.
15. A.G. Ranade, S.N. Bhatt, and S.L. Johnson. The Fluent Abstract Machine. In *Proc. of the 5th MIT Conference on Advanced Research in VLSI*, pages 71–93, Cambridge, MA, 1988. MIT press.
16. Tera Corporation. Tera MTA computer sets new record for integer sorting. Press Release, 26. Mar. 97, mar 1997. available under www.tera.com.
17. T. Walle. *Das Netzwerk der SB–PRAM.* PhD thesis, University of the Saarland, 1997. in German.