NAS Integer Sort on Multi-threaded Shared Memory Machines*

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

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

Abstract. Multi–threaded shared memory machines, like the commercial Tera MTA or the experimental SB–PRAM, have an extremely good performance on the Integer Sort benchmark of the NAS Parallel Benchmark Suite and are expected to scale. The number of CPU cycles is an order of magnitude lower than the numbers reported of general purpose distributed memory or shared memory machines; even vector computers are slower. The reasons for this behavior are investigated. It turns out that both machines 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. Except for non–scalable vector computers, the Cray T3E, which supports fetch–and–add but not multi–threading, is the only parallel computer that could challenge these machines.

1 Introduction

One of the first programs that ran on a node processor of the Tera MTA (multi–threaded architecture) [2,3] was the Integer Sort (IS) from the Parallel Benchmark suite [8,14] of the Numerical Aerospace Simulation Facility (NAS) at NASA Ames Research Center. According to press releases of Tera Corporation [17], 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 30 percent. The SB–PRAM [1,9,12,15] has many features in common with the Tera MTA, although it is inspired by a totally different idea, viz realizing the PRAM model from theoretical computer science [18]. Both 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 in the memory system. Commercial microprocessors, which are employed in most of todays parallel computers do not offer this kind

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

of hardware support for running efficient parallel programs. An exception is the Cray T3E, which is a UMA machine with fetch–and–add support but without multi–threading.

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 its position in the sorted list. 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 2.4 cycles. Both machines are expected to scale without significant loss of efficiency. A Cray T90 node approaches 5 CPE and the numbers reported for the MPI–based IS sample code on various distributed and shared memory computers (including the Cray T3E) are greater than 25 CPE. Besides, scalability is a problem on these machines. In this article we explain, why this numbers differ in such a wide range and investigate whether there are better implementations than the sample code for CC–NUMA machines and the Cray T3E.

The paper is organized as follows. Section 2 discusses the IS benchmark specification. Section 3 addresses multi–threaded shared memory machines and Section 4 discusses a fast algorithm for vector computers. Section 5 presents esults of the MPI–based message–passing sample code and Section 6 investigates CC–NUMA machines or the Cray T3E UMA machine could improve on these ults. Section 7 concludes.

2 The NAS Integer Sort Benchmark

The Integer Sort (IS) program is part of the NAS Numerical Parallel Benchmark suite [8]. Despite its name it does not sort but rank an array of small integers. The first revision (NPB 1) was a "paper and pencil" benchmark specification, so as not to preclude any programming tricks for a particular machine. The benchmark specifies that an array `keys[]` of N keys is filled using a fully specified random number generator that produces Gaussian distributed random integers in the range $[0, B_{max}-1]$. This key generation and a final verification step, which checks the results, are excluded from the timing. There are several "classes" of the benchmark, which define the parameter values N and B_{max}. We concentrate on class A ($N = 2^{23}, B_{max} = 2^{19}$); the parameters for class B and C are higher by a factor 4, and 16 respectively. The timed part of the benchmark consists of 10 iterations of (a) a modification of two keys, (b) the actual ranking step, and (c) a partial verification that tests if the well known rank of 5 keys has been computed correctly. A typical implementation of the inner loops that perform the actual ranking is listed in Table 1.

The NPB 2 revision of the benchmark intends to supplement the original benchmark description with sample implementations that can be used as a starting point for machine–specific optimizations. There is a serial implementation (NPB 2.3–serial) and an implementation based on the message passing standard

Table 1. Inner loops of the NAS IS benchmark

(1) `for i = 1 to B: count[i] = 0`	// clear count array
(2) `for i = 1 to N: count[key[i]]++`	// count keys
(3) `for i = 1 to B: pos[i] =` $\sum_{k=0}^{i-1}$ `count[i]`	// calc start position
(4) `for i = 1 to N: rank[i] = pos[key[i]]++`	// ONLY NPB 1

MPI (NPB 2.3b2). Unfortunately, the fourth loop of the benchmark, which was present in earlier sample implementations of NPB 1, has been omitted in NPB 2. However, the fourth loop is identical to the second loop except for storing the result, and does not require a new programming technique. Therefore, we use the NPB 2 variant of IS, class A throughout this paper.

Performance measures. The original performance measure of NAS IS is the elapsed time for 10 iterations of the inner loops. The main performance measure in our investigation is the number of CPU cycles per key element (CPE), because it emphasizes the architectural aspect rather than technology. We will point out situations in which CPE is an unfair measure.

3 Multi–threaded SMMs

First, we sketch the SB–PRAM design and highlight the Tera MTA differences. Then we describe the result of the best SB–PRAM implementation and indicate the algorithmic changes in the Tera MTA implementation.

3.1 Hardware Design

The SB–PRAM [1, 9, 12] 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 of 32 virtual processors (VP) 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. The combining facility is extended to do parallel prefix computations, which, due to synchronous execution, look to the programmer as if the VPs executed the operations one after another in a predefined order (sequential semantics).

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 [19] using different and detailed simulators, the main criticism on the SB–PRAM is that these investigations were based on simulations only. 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 [4] has been completed. The 64 processor machine is expected to be running in summer 1998.

In [10] a variant of the SB–PRAM, called High Performance PRAM (HPP), has been sketched. Due to modest architectural changes and using top 1995 technology, the HPP is expected to achieve the tenfold performance of the SB–PRAM on average compiler generated programs.

Tera MTA. The design goal of the Tera MTA [2,3,17] 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 $\geq$ 294 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 active memory requests. 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. 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. To our knowledge there is no detailed, technical description or a correctness proof of the network available to the public. The machine supports fetch–and–add instructions; the requests are not combined in the network, but serviced sequentially 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.

3.2 Benchmark Implementation

Due to some limitations of our `gcc` compiler port for the SB–PRAM, we have hand–coded the loop bodies in assembly language and manually unrolled the inner loops of the benchmark. For details we refer to [11]. We now present 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.

As on every other machine, the count array can be cleared with a single "store with auto–increment" instruction, if every VP is assigned a different but contiguous portion of the count array. The arrays `count[]` and `pos[]` are used one at a time and can be coalesced into a single `cp[]` array, which saves a few address computations. Because there are no data dependencies between different iterations of loop 2, all `PROCS` virtual processors are mapped round robin onto the `keys` and `cp` arrays and synchronously execute loops 2 and 3. Two successive

```
(a) elem1 = M( k_ptr1+=2*PROCS);    elem2 = M( k_ptr2+=2*PROCS);
    cp_ptr1 = elem1 + cp_base;      cp_ptr2 = elem2 + cp_base;
    syncadd( cp_ptr1, 1);           syncadd( cp_ptr2, 1);
(b) elem1 = M( cp_ptr1+=2*PROCS);   elem2 = M( cp_ptr2+=2*PROCS);
    elem1 = mpadd( ranksum, elem1); elem2 = mpadd( ranksum, elem2);
    M( cp_ptr1) = elem1;            M( cp_ptr2) = elem2;
```

Fig. 1. Interleaved loop bodies: (a) Loop 2, and (b) Loop 3.

loop bodies are interleaved in order to fill all load delay slots (see assembly code in Figure 1.a). The first instruction combines the incrementing of a properly initialized, private key pointer `k_ptr` by 2·`PROCS` with the loading of the next key. In the second instruction of the loop body, the pointer `cp_ptr` is computed as `cp[]` indexed by `key`, and the `syncadd` operation (`mpadd` without result) increments this entry. Figure 1.b lists the assembly code for the third loop, which also employs the interleaving technique. In the first instruction, the pointer `cp_ptr` is adjusted and an entry of `cp[]` is read. Then, the accumulated rank is computed in the multi–prefix add on `ranksum`, and the result is written back to the `cp` array. This loop relies on synchronous execution, because the order of `mpadd` instructions among VPs is crucial.

Loop 2 contributes most instructions to the timed portion of the code, because it is executed `N` times whereas loops 1 and 3 are executed only $B_{max} = \frac{N}{16}$ times. Thus, CPE(ranking) $= \frac{1}{16} + 3 + \frac{3}{16} = 3.25$ is optimal for the SB–PRAM. The benchmark implemented using assembly language macros and 64–fold unrolled loop achieves 3.30 CPE. A similar tuned 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.

We have run the benchmark on our instruction–level simulator as well as on the real 4 processor machine. The run times differ by less than 0.1%, a fact that validates our network simulations which predicted that memory stalls are extremely unlikely. The run time obtained by a 128 processor simulation is only 1% slower than $\frac{1}{128}$th of the one processor run time, i.e. speedup is nearly linear. 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 High–Performance–PRAM HPP would gain a factor of 5.6 in absolute speed, but the CPE value would be worse, because the machine can issue memory requests at only a third of the instruction rate. For a fairer comparison, the CPU clock should be replaced by the memory request rate.

Tera MTA. The Tera MTA system has a sophisticated parallelizing compiler, which processes the sequential source code and automatically generates threads when needed. The assembler output of loop 2, which is listed in [2], shows a 2 instruction loop body that is 5-fold unrolled. The memory operations are "load next key" and "fetch & increment count". An integer addition plus the loop

assume: V1 ← 0..63	
(a) 1 : V2 ← key[V1 + k]	(b) 2.1 : count[V2] ← V1
2 : V3 ← count[V2] + 1	2.2 : V4 ← count[V2] - V1
3 : count[V2] ← V3	2.3 : if(V4 ≠ 0) check

Fig. 2. Loop 2 vector code: (a) straight forward, (b) correction

overhead are hidden in the non–memory operations of the ten instructions of the loop body. Note, that the CPU clock and the memory request rate are equal in the super–scalar Tera MTA design.

Since the Tera MTA lacks synchronous execution, the third loop cannot be parallelized as on the SB–PRAM. Instead, every processor works on a contiguous block of the count array, computes the local prefix sums and outputs the global sum of its elements. In a second step, the prefix sum of the global sums is computed. Then, each processor adds the appropriate global prefix sum to its local elements. This procedure requires 3 instructions plus a small logarithmic term. For the class A benchmark it is less than 4 instructions. The Tera MTA requires less than $\frac{1}{16}+2+\frac{4}{16} = 2.3125$ CPE. The improvement compared to the SB–PRAM comes from the ability to execute one memory operation and two other (integer, jump, ...) operations per instruction.

We do not have access to a Tera MTA ourselves. The performance figures contained in this section are derived from [2, 5] and personal communication with P. Briggs (Tera Computer Corporation) and L. Carter (SDSC). The numbers reported in [2] are 1.53 seconds and $5.25 < \text{CPE} < 5.3125$ for NPB 1. Assuming a CPE value of 5.25, the 294 MHz machine has an overhead of at least $\frac{1.53 \cdot 294 \cdot 10^6}{5.25 \cdot 10 \cdot 2^{23}} - 1 = 2.14$ %, i.e. hiding memory latency works well for the one processor prototype. The fourth inner loop of the benchmark, which is not present in NPB 2, accounts for 3 CPE. If we attribute all overhead to the three loops of IS(NPB 2) and conservatively assume CPE(NPB 2)=2.3125, we end up with a run time of < 0.68 seconds. Carter [5], who ran the NPB 2 benchmark on a 145 MHz machine, achieved a run time of 2.05 seconds, which corresponds to an overhead of 50 percent. He believes that on the early machine flipped bits in the network packets induced retransmissions, because the memory overhead figures for other benchmarks dropped as the hardware became more stable.

4 Vector Computers

In [5], a Tera MTA node is compared to a Cray T90 vector processor node on the NPB 2–serial sample code of IS. Vector processors have instructions for computing prefix sums on vector registers. Thus, the third benchmark loop can be parallelized in the same way as on the Tera MTA. The difficult part is loop 2, where even a single vector processor encounters problems.

Figure 2.a lists the straightforward, but wrong, implementation of loop 2. Successive, contiguous stripes of `key[]` are read into vector register V2. Then,

Table 2. IS (class A) benchmark results

Machine Name	CPU clock [MHz]	minimum #procs	minimum time	minimum CPE	maximum #procs	maximum time	maximum CPE
SB-PRAM	7	1	39.55	3.30	4	9.89	3.30
SB-PRAM simulation					128	0.31	3.33
Tera MTA	294	1	0.68	2.32			
Cray T90	440	1	1.11	5.82			
SB-PRAM (MPI)	7	2	160.8	26.85			
IBM SP2 WN	66	2	29.1	27.38	128	0.6	60.42
SGI Origin 2000	195	2	20.5	95.30	32	1.6	119.02
Cray T3E-900	450	2	12.7	136.26	256	0.4	258.15

V3 is filled by reading the corresponding `count[]` entries and incrementing them. Afterwards V3 is written back. If a specific key value v is contained twice in V2 at positions k_1 and k_2, `count[`v`]` is incremented only once, because V3[k_2], which finally ends up in `count[`v`]`, contains $\texttt{count}_{old}\texttt{[}v\texttt{]+1}$ and not the incremented V3[k_1] value.

The Cray compiler can cure the situation using a patented algorithm [6] invented by Booth.[1] Before writing V3 back in line 3, the code listed in Figure 2.b tests whether duplicates occur, and, if necessary, processes them in the scalar unit. The run time of IS(NPB 2), as reported in [5], is 1.11 s, which corresponds to 5.82 CPE. The NPB 2.3–serial sample code contains an unnecessary store that possibly has not been detected by the compiler. Hence, the optimal CPE for a single vector computer could be lower by 1 CPE.

The above code for loop 2 can be easily parallelized by maintaining a private copy $\texttt{count}_i\texttt{[]}$ of the count array on each node computer. When p denotes the number of participating processors, the third loop becomes a parallel prefix computation $\texttt{pos[i]} = \sum_{k=0}^{i-1}\sum_{l=0}^{p-1} \texttt{count}_l\texttt{[k]}$. Memory consumption as well as the execution time of loop 3 scale with p. If p becomes very large, a two pass radix sort, like described by Zagha [20], can improve performance.

5 Comparison with MPI–based sample code

Table 2 lists the benchmark results discussed so far and compares them with run times of MPI–based sample code. We implemented the necessary MPI library functions on the SB–PRAM and optimized the sample code to the extent that an optimizing compiler could also achieve [11]. The times for the other machines have been taken from benchmark results published by NAS. The IBM SP2 is a distributed memory machine (DMM) with an extremely fast interconnection network, the SGI Origin 2000 a cache–coherent NUMA, and the Cray T3E a UMA shared memory machine (SMM) without cache coherence.

[1] Personal Communication with Larry Carter and Max Dechantsreiter.

Sample code implementation. The keys are distributed on all participating p computers before the timing starts. In a first local step, each computer sorts its keys using only the b most significant bits into 2^b buckets. Because the keys are put into a sorted order, this step alone requires more instructions than IS(NPB 1). Then, a mapping of buckets to processors, which balances the load, is computed. Afterwards, all local bucket parts are sent to their destinations in a single, global, all–to–all communication step. Now, every processor owns a distinct, contiguous key range and can count its keys locally. The `pos[]` entries can be computed from `count[]` like in the Tera MTA implementation.

The sample code algorithm is guided by the idea of doing one, central all–to–all communication. The price for this procedure is to rearrange the keys locally, such that keys that are sent to the same destination are stored contiguously in memory. A plus of the implementation is the good cache efficiency: the key arrays are accessed sequentially, so that the initial cache miss time is amortized over the complete cache line, and the count arrays fit into second level cache.

With less than $p = 16$ processors another strategy with fewer communication overhead would be possible: every processor could count its keys in a local count array and `pos[]` could be computed like on vector processors after transposing the $p \times B_{max}$ count matrix. If $p < 16$, then only $p \cdot B_{max} < 16 \cdot \frac{N}{16}$ data elements had to be sent over the network. However, the sample code is written for a large number of processors and slow communication networks.

The CPE of the SB–PRAM is 26.85, of which 9 cycles belong to memory instructions operating on key elements. The SB–PRAM issues one instruction per cycle and encounters no memory waits. The other machines have super–scalar processors, but suffer from memory waits. The impact of wait states on the CPE is the higher, the higher the clock rate of the specific machine is. Table 2 highlights this fact, which limits the benefit of cache–based architectures from future technological improvements compared to vector computers or multi–threaded machines.

6 Other Algorithms

The MPI–based sample code is tailored for distributed memory machines. We now investigate, if there are better algorithms for CC–NUMA machines or the Cray T3E.

CC–NUMA. Cache–coherent non–uniform–memory–access SMMs [13], 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.

We now investigate, if the direct approach of locking the `count[]` elements for updating could improve performance so much, that the results of multi–threaded SMMs could be reached. Therefore, we calculate the average memory

latency introduced by accessing `count[keys[i]]` in loop 2. We assume that on a p–processor machine $\frac{1}{p}$ of the `count[]` 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

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

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

Cray T3E. A Cray T3E node computer consists of a DEC Alpha processor with up to 2 GByte local RAM. Additionally, the system logic provides a notion of a logically addressable shared memory through the E–registers [16]. Up to 512 E–registers can be accessed by the user to trigger shared memory accesses. In addition to load and store, E–registers also support fetch & add and an especially fast fetch & increment operation.

Although the machine does not support multi–threading in hardware, memory wait conditions can be avoided by using multiple E–registers to simulate multi–threading in software. With this technique, alternate key load and count increment operations can be performed at the maximal network clock of 75 MHz. The third loop of IS can also be parallelized in the same way as on the Tera MTA. Thus, less than 3 network cycles per key element are possible.

7 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 most of todays computers. Besides multi–threading, both machines take advantage of a fetch–and–add instruction to update shared data structures conflict–free. These two properties lead to a performance that is an order of magnitude higher compared to published results of other scalable parallel computers, both DMMs and SMMs.

In particular, the experimental 7 MHz SB–PRAM prototype can compete with modern parallel computers on this benchmark. The Tera MTA makes use of expensive top technology, and achieves better performance figures than non–scalable top vector processors. Although CC–NUMA SMMs could possibly improve performance compared to DMMs, they are still an order of magnitude slower than multi–threaded machines. Alone the Cray T3E, which supports and atomic fetch & increment operation could challenge the performance of the Tera MTA by emulating multi–threading in software.

Acknowledgments. The authors would like to thank Larry Carter, Max Dechantsreiter, and Preston Briggs for helpful discussions and all people who contributed to the SB–PRAM project [15].

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. R. Alverson, P. Briggs, S. Coatney, S. Kahan, and R. Korry. Tera Hardware–Software Cooperation. In *Proc. of Supercomputing '97*. San Jose, CA, November 1997.
3. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The TERA Computer System. In *Proc. of Intl. Conf. on Supercomputing*, June 1990.
4. P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grün, and C. Lichtenau. Building the 4 Processor SB–PRAM Prototype. In *Proc. of the 30th Hawaii International Conference on System Sciences*, pages 14–23, January 1997.
5. J. Boisseau, L. Carter, K.S. Gatlin, A. Majumdar, and S. Snavely. NAS Benchmarks on the Tera MTA. In *Proc. of Workshop on Multi-Threac'ec' Execution, Architecture anc' Compilation (M-TEAC 98)*, Las Vegas, February 1998.
6. M. Booth. US Patent 5247696. see `http://www.patents.ibm.com`, September 1993.
7. C. Engelmann and J. Keller. Simulation–based comparison of hash functions for emulated shared memory. In *Proc. PARLE (Parallel Architectures anc' 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, March 1994. see also `http://science.nas.nasa.gov/Software/NPB`.
9. A. Formella, T. Grün, and C.W. Kessler. The SB–PRAM: Concept, Design and Construction. In *Draft Proceec'ings of 3rc' International Working Conference on Massively Parallel Programming Moc'els (MPPM-97)*, November 1997. see also `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 *Proceec'ings of the 2nc' Europar*, volume II of *LNCS 1124*, pages 425–434. Springer, August 1996.
11. T. Grün and M. A. Hillebrand. NAS Integersort on the SB–PRAM. Manuscript, available via `http://www-wjp/~tgr/NASIS`, May 1998.
12. T. Grün, T. Rauber, and J. Röhrig. Support for Efficient Programming on the SB–PRAM. *International Journal of Parallel Programming*, 26(3):209–240, June 1998.
13. D.E. Lenoski and W.-D. Weber. *Scalable Sharec'–Memory Multiprocessing.* Morgan Kaufmann Publishers, 1995.
14. NAS Parallel Benchmarks Home Page. `http:// science.nas.nasa.gov/Software/NPB/`.
15. SB–PRAM Home Page. `http:// www-wjp.cs.uni-sb.de/sbpram`.
16. S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proc. of the VII ASPLOS (Architectural Support for Programming Languages anc' Operating Systems) Conference*, pages 26–36. ACM, October 1996.
17. Tera Computer Corporation Home Page. `http://www.tera.com/`.

18. J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume A, pages 869–941. Elsevier, 1990.
19. T. Walle. *Das Netzwerk der SB–PRAM.* PhD thesis, University of the Saarland, 1997. in German.
20. M. Zagha and G.E. Belloch. Radix Sort for Vector Multiprocessors. In *Proc. of Supercomputing '91*, pages 712–721, New York, NY, November 1991.