A Fast, Storage-Efficient Parallel Sorting Algorithm*

Richard P. Brent and Andrew Tridgell
Computer Sciences Laboratory
Australian National University
Canberra, ACT 0200
Australia

{rpb,tridge}@cslab.anu.edu.au

Abstract

A parallel sorting algorithm is presented for storage-efficient internal sorting on MIMD machines. The algorithm first sorts the elements within each node using a serial sorting algorithm, then uses a two-phase parallel merge. The algorithm is comparison-based and requires additional storage of order the square root of the number of elements in each node. Performance of the algorithm on two general-purpose MIMD machines, the Fujitsu AP1000 and the Thinking Machines CM5, is examined. The algorithm is suitable for implementation on special-purpose parallel machines, e.g. parallel database machines.

Key words and phrases.

Batcher's merge-exchange sort, distributed memory, Fujitsu AP1000, parallel sorting, sorting, Sparc, Thinking Machines CM5.

1: Introduction and Aims

There is a large literature on parallel sorting – see, for example, [1, 2, 9] and the references given there. Many of these papers have dealt with the problem from a theoretical point of view, neglecting issues which are important in a practical implementation of a parallel sorting algorithm [3, 7]. This paper describes a fast, practical parallel sorting algorithm which has been implemented on several MIMD machines and used in applications such as speech recognition and text retrieval. We aimed for an algorithm with the following properties.

1. Speed. The algorithm is competitive with the fastest known algorithms.
2. Good memory utilisation. The number of elements that can be sorted is close to the number that can be stored in the memory of the machine.
3. Flexibility. No restrictions are placed on the number of records to sort or the number of processors.
4. Determinism. The algorithm does not use a random number generator.
5. Comparison-based. The only operation used on keys is binary comparison.

Property 2 is especially important for special-purpose machines on which a single application may occupy the whole machine. This memory utilisation constraint rules out the usual implementations of radix sort and sample sort, which require workspace at least equal to the original data size. Property 4 rules out methods such as sample-sort, which depend on taking a pseudo-random sample of the input data. Property 5 rules out methods such as radix sort.

Our algorithm first sorts the records within each processor using a fast serial sorting algorithm, and then a two-phase parallel merge is performed. The first merge phase is very efficient and "almost" sorts the data. The second merge phase is guaranteed to sort, and is efficient when operating on data which is almost sorted. Working storage requirements are negligible (of the order of the square root of the number of elements in each node).

Our algorithm is similar in many respects to parallel shellsort [4], but contains a number of new features. For example, the memory overhead is considerably reduced.

An overview of the algorithm is given in Section 2. Because of space limitation, many details had to be omitted from this paper, but they may be found in [11], which is available by anonymous ftp. The performance of our implementations on two MIMD parallel machines, the Fujitsu AP1000 and the Thinking Machines CM5, is discussed in Section 3.

1.1: Notation

P is the number of nodes (also called cells or processors) available on the parallel machine, and N is the total number of elements to be sorted. N_p is the number of elements in a particular node p ($0 \le p < P$).

Elements within each node of the machine are referred to as $E_{p,i}$, for $0 \le i < N_p$ and $0 \le p < P$.

When giving "big O" time bounds we usually assume that P is fixed. Thus, we do not usually distinguish between $O(N)$ and $O(N/P)$.

Speedup and *efficiency* are defined as usual (see Section 3.1.).

1.2: Restrictions

The only operation assumed to be defined between elements is binary comparison, written with the usual comparison symbols. For example, $A < B$ means that element A precedes element B. The elements are considered sorted when they are in non-decreasing order in each node, and non-decreasing order between nodes. More precisely, this means that $E_{p,i} \le E_{p,j}$ for all relevant $i < j$ and p, and that $E_{p,i} \le E_{q,j}$ for $0 \le p < q < P$ and all relevant i, j.

We restricted ourselves to algorithms for sorting elements of a fixed size, because of the difficulties of pointer representations between nodes in a MIMD machine. In short, we were aiming to produce a parallel equivalent of the qsort() C library function.

To obtain good memory utilisation when sorting small elements, we avoided representations using linked lists. Thus, the lists of elements referred to below are implemented using arrays, without any storage overhead for pointers.

1.3: Hardware

For purposes of illustration we examine the performance of implementations of the parallel sorting algorithm on two parallel MIMD computers.

- Fujitsu AP1000 [5]. This machine contains 128 Sparc scalar nodes connected on an 8 by 16 torus. Node to node communication is performed by hardware, using wormhole routing. Each node has 16Mb of local memory and all are connected to a host workstation via a relatively slow connection.
- Thinking Machines CM5 [13]. This machine contains 32 Sparc scalar nodes connected by a communication network that has the topology of a tree. Each Sparc node has two vector processors which are time-sliced to emulate four virtual vector processors. Each virtual vector processor controls a bank of 8Mb of memory, giving the Sparc node access to a total of 32 Mb of memory. In our algorithm no use is made of the vector processors other than as memory controllers.

Both machines support a general message-passing model as well as a wealth of broadcast and other communications primitives. Our implementation of parallel sorting only uses a subset of message passing primitives common to both machines, and for this reason it should be relatively easy to port to other MIMD machines.

There are a number of small but significant implementation differences in the individual nodes of the two machines –

- The clock speed is 32 MHz on the CM5, and 25 MHz on the AP1000.
- The cache line size is 32 bytes on the CM5, and 16 bytes on the AP1000.
- The cache size is 64KB on the CM5, and 128KB on the AP1000.

It will be apparent from the description below that our algorithm is ideally suited to a machine with a hypercube topology. Neither the CM5 nor the AP1000 has a hypercube topology, so communication patterns which would not cause network contention on a hypercube may cause contention on the CM5 or the AP1000. This does not have a serious impact on performance (see Section 3).

2: The Algorithm

The algorithm has four distinct phases (pre-balancing, serial sorting, primary merging, and cleanup). The primary merging and cleanup phases both use the merge-exchange operation. In Sections 2.1 to 2.5 below, we outline the purpose and implementation of each phase, and describe the merge-exchange operation.

The pre-balancing and primary merging phases are logically unnecessary, and could in principle be omitted. They are included to improve the performance. Without them, the algorithm would still sort, but much more slowly.

The algorithm starts with a number of elements N assumed to be distributed over P processing nodes. No particular distribution of elements is assumed and the only restrictions on N and P are those imposed by the physical constraints of the machine.

2.1: Pre-Balancing

The pre-balancing phase moves elements between the nodes so as to achieve as close to an even distribution as possible. This phase is desirable to minimise the load imbalance between nodes in later phases of the algorithm. The balancing is achieved by exchanging elements between pairs of nodes. The communication pattern corresponds to the edges of a hyper-cube in the case that the number of nodes is a power of 2. This method produces approximately N/P elements in each node, with an error of order $\log P$ for each node if the number of nodes is a power of 2.

In applications where the distribution of elements across nodes is known in advance to be well-balanced, or on machines for which simulation of a hyper-cube is expensive (e.g. a 1D or 2D systolic array without wormhole routing), the pre-balancing phase could be omitted.

2.2: Serial Sorting

In the serial sorting phase there is no communication between nodes, but a fast comparison-based serial sorting algorithm is applied to the elements in each of the nodes. At the end of this phase the data is in the form of P sorted lists of elements, with approximately N/P elements in each list.

The best serial sort is machine-dependent. The method chosen (after some experimentation) for our implementation on Sparc nodes was a combination of quicksort and insertion sort[1].

2.3: Primary Merging

The data is considered almost sorted if it is possible to complete the sorting process in a small proportion of the overall time for the algorithm. The aim of the primary merging phase of the algorithm is to almost completely sort the data in a very efficient manner. This phase maintains the balancing of the lists between the nodes, and each of the lists remains sorted.

The communication pattern of the primary merging phase is similar to that of the pre-balancing phase. A merge-exchange operation is performed between nodes in a pattern that reduces to the edges of a hypercube if P is a power of 2. This means that each node must perform $\log_2 P$ merge-exchange operations. The use of this hypercube pattern of merging guarantees that each node has about the same amount of work to do at each step. In practice this reduces the load imbalance between the nodes almost to nil and allows the algorithm to achieve a high parallel efficiency.

It is possible to omit the primary merging phase, but (at least on the AP1000 and CM5) this increases the overall sorting time.

On a machine for which simulation of a hypercube is expensive, e.g. a 2D systolic array without wormhole routing, a different communication pattern could be used. There is a wide choice because there is no need to guarantee that the output is sorted after the primary merging phase.

[1] The implementation is a highly optimised adaptation of code written by the Free Software Foundation for the GNU project. It was found to perform up to twice as fast as the standard C library function qsort().

2.4: Cleanup

The aim of the cleanup phase is to guarantee that the data is completely sorted, while consuming very little time for data that is almost sorted. The algorithm chosen was Batcher's merge-exchange[2] algorithm [6, Sec. 5.2.2]. The algorithm is actually a generalisation of Batcher's merge-exchange algorithm, in that it operates on lists of elements rather than on single elements. The generalisation is straightforward, and the proof of its correctness is given in [6, problem 5.3.4.38]. It is important to note that the proof requires the lists to have equal sizes, and small examples show that this restriction is necessary. However, a device known as *infinity-padding* (described in detail in [11]) allows us to circumvent this restriction by implicitly padding the lists with "∞ elements".

The cleanup algorithm defines a pattern of merge-exchange operations which merge already-sorted lists of elements into completely sorted order. The algorithm takes $O((\log P)^2)$ steps on each of the nodes, and uses the same merge-exchange algorithm that is used for the primary merging phase.

For reasons described in [11] the algorithm is very efficient if the data is almost sorted (this would not be true if we used Batcher's bitonic sort). In practice the cleanup is found to take only a small proportion of the total time (see Section 3.4).

On some architectures Batcher's merge-exchange algorithm may not be the best choice. For example, on a systolic array it would be preferable to use an algorithm which required only nearest-neighbour communication. One such variation is described in [12].

2.5: Merge-Exchange

Suppose that $p_1 < p_2$. A merge-exchange between nodes p_1 and p_2 results in node p_1 having all its elements less than or equal to those in the node p_2, while maintaining the ordering of elements within the nodes. The efficiency of the merge-exchange algorithm has a large influence on the overall efficiency of the parallel sorting algorithm.

It is important that the merge-exchange algorithm should not use an excessive amount of temporary storage, which would severely limit the number of elements that could be sorted on a given hardware configuration. Our algorithm requires $3\sqrt{N/P}$ elements of temporary storage, which is a trivial amount in practice.

The first part of the merge-exchange algorithm is to determine exactly how many elements from node p_2 will be required by node p_1 and vice versa. This is completed in at most $\log_2(N/P)$ steps, where each step requires one comparison and the transfer of one element from node p_2 to p_1.

The next part is to transfer the elements between the nodes. This must be done so that the space freed by moving elements from p_1 to p_2 can be used to contain the elements coming from p_2. The results of the first part allow this to be performed without the allocation of additional memory.

In order to minimise working storage requirements, we devised a merge algorithm which operates on lists of blocks of elements. This algorithm requires approximately N/P memory movements and $3\sqrt{N/P}$ elements of additional storage. An important special case occurs when the sizes of the two lists are very different. Our algorithm is designed to be particularly fast in this case. Details of this algorithm are discussed in [11], and a possible improvement is given in [12].

[2] Not to be confused with Batcher's bitonic sorting algorithm [6, Sec. 5.3.4].

3: Performance

3.1: Estimating the Speedup

Speedup is usually defined to be the ratio of the time taken to solve the problem on a single node (using the best known algorithm) to the time taken by the parallel algorithm. *Efficiency* is defined to be speedup divided by the number of nodes. There is a difficulty in measuring the speedup in practice, due to the limited memory available on each node. The parallel sorting algorithm only performs at its best for values of N which are far beyond that which a single node on the CM5 or AP1000 can hold. To overcome this difficulty, we have extrapolated the timing results of the serial algorithm to larger N. Thus the speedup is measured by comparing the parallel algorithm with the best serial algorithm running on a hypothetical node with very large memory.

The quicksort/insertion-sort algorithm has an asymptotic average run time of order $N \log N$. However, there are significant contributions of lower asymptotic order to the run time. To estimate these contributions we performed a least squares fit of the form:

$$\text{time}(N) = a + b \log N + cN + dN \log N.$$

The results of this fit are used to extrapolate the run time to problems which are too large to fit on a single node, and thus to estimate the speedup.

3.2: Timing Results

Several runs have been made on the AP1000 and CM5 to examine the performance of the sorting algorithm. Figure 1 shows the performance of the algorithm for sorting random 32-bit integers on the 128-node AP1000. The sorting speed (in millions of elements per second) is plotted versus the total number (N) of 32-bit integers being sorted. N ranges from values which would be easily dealt with on a workstation, to those at the limit of the AP1000s memory capacity (2 Gbyte). For this example the comparison function was put inline to reduce function call overheads.

Shown on the same graph is the performance of a hypothetical serial computer that operates P times as fast as the P individual nodes of the parallel computer. For small N, this performance is calculated by sorting the elements on a single node and multiplying the observed speed by P. For large N, extrapolation is used, as described in Section 3.1.

The graph shows that the performance of the sorting algorithm increases up to $N \simeq 4 \times 10^6$, and slowly falls off for larger N. The roll-off point corresponds to the number of elements that can be held in the 128KB cache of each node (i.e. 4 Mbytes overall cache). The algorithm achieves an efficiency of about 75% for large N.

A similar result for sorting 16-byte random strings is shown in Figure 2. In this case the comparison function is the C library function strcmp(). The roll-off point is now $N \simeq 10^6$, again corresponding to the cache size.

The performance for 16-byte strings is approximately 6 times worse than for 32-bit integers. This is because each data item is 4 times larger, and the cost of the function call to strcmp() is much higher than the cost of an inline integer comparison. The speedup is higher than that achieved for the integer sorting. The efficiency is close to 85% for large N.

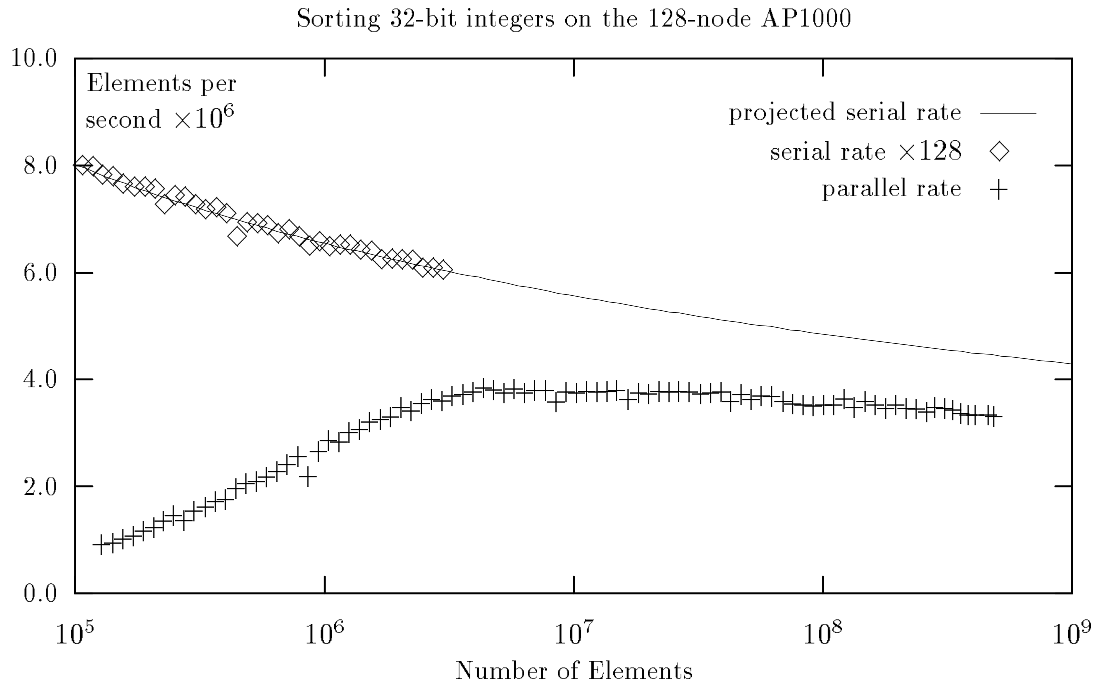

Figure 1: **Sorting 32-bit integers on the AP1000**

3.3: Scalability

Figure 3 shows the efficiency for sorting $10^5 P$ 16-byte strings on the AP1000 as the number of nodes P is varied. Because the number of strings per node is constant, the cache size does not influence the shape of the graph.

The efficiency decreases as P increases, because communication costs and load imbalances become significant. The graph flattens out for larger P, which indicates that the algorithm should have a high efficiency when $P > 128$.

The two curves in Figure 3 show the trend when all configurations are included, and when only configurations with P a power of 2 are included. The difference between these two curves clearly shows the preference for powers of two in the algorithm. There are also slight preferences for P the sum of adjacent powers of two (e.g. $P = 48$), and for even P over odd P.

3.4: Time Breakdown

Figure 4 shows the proportion of time used by each phase when sorting N 16-byte strings on the AP1000, for a range of values of N. The prebalancing phase is not shown in Figure 4, since it takes a negligible time unless the data is initially very badly balanced.

For large N the serial sort of the elements in each cell is the most time-consuming phase. This is as expected, because this phase of the algorithm has a theoretical average time of $\Theta(N \log N)$, whereas all other phases of the algorithm are $O(N)$. This observation also explains the high efficiency of the algorithm.

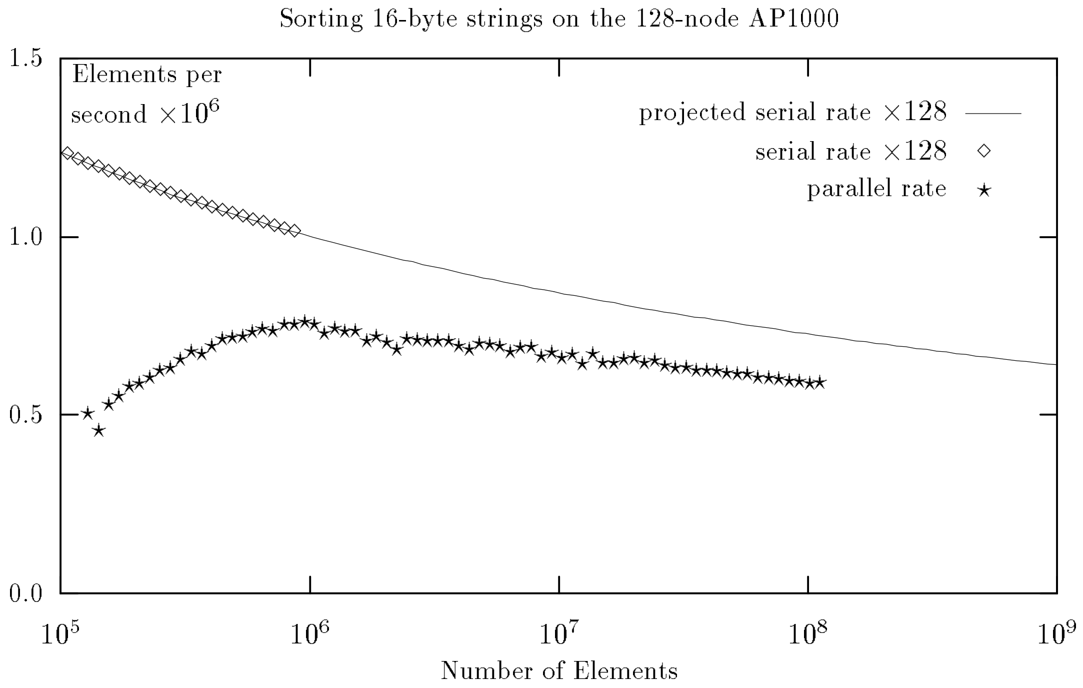

Figure 2: **Sorting 16-byte strings on the AP1000**

It is interesting to observe the small proportion of time taken by the cleanup phase for large values of N. This demonstrates that the primary merge produces an almost sorted data set, and that the cleanup phase takes advantage of this.

3.5: CM5 vs AP1000

The results presented so far are for the 128-node AP1000. It is interesting to compare this machine with the CM5 to see if the relative performance is as expected. To make the comparison fairer, we compare the 32-node CM5 with a 32-node AP1000 (the other 96 nodes are physically present but not used). Since the CM5 vector units are not used (except as memory controllers), we effectively have two rather similar machines. The same C compiler was used on both machines.

The AP1000 is a single-user machine and the timing results obtained on it are very consistent. However, it is difficult to obtain accurate timing information on the CM5. This is a consequence of the time-sharing capabilities of the CM5 nodes. Communication-intensive operations produce timing results which vary by a large factor from run to run. To overcome this problem, the times reported here are for runs with a very long time quantum for the time sharing, and with only one process on the machine at one time. Even so, we have ignored occasional anomalous results which take much longer than usual. This means that the results are not strictly representative of results that are regularly achieved in a real application.

Table 1 shows the time taken by the various parts of the sorting algorithm on the 32-node AP1000 and CM5. In this example we are sorting 8 million 32-bit integers.

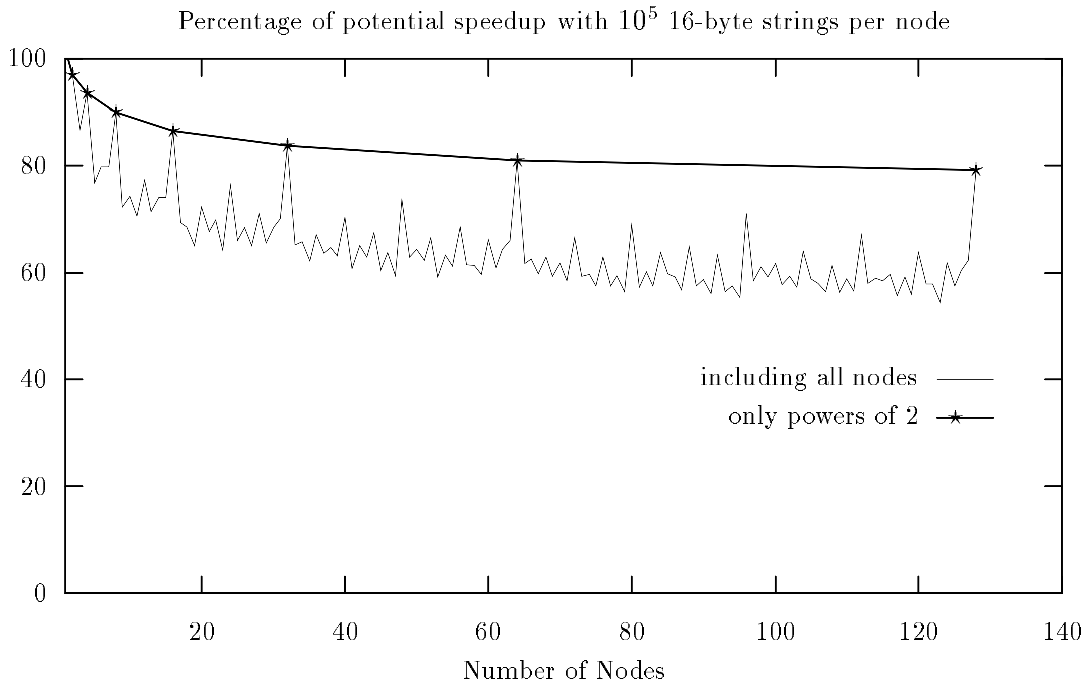

Figure 3: **Scalability of sorting on the AP1000**

The communication and idle times are about equal, but the CM5 is faster than the AP1000 on CPU-intensive operations. This is because of the higher clock speed (32MHz versus 25MHz), the different cache line sizes, and the higher memory bandwidth of the CM5. For a more detailed discussion, see [11].

The results illustrate the significance of minor architectural differences. On the other hand, the different connection topologies of the two machines are not reflected in significantly different communication times. This suggests that the parallel sorting algorithm presented here can perform well on a variety of parallel machine architectures with different communication topologies. Our code has recently been ported to a 64-node NCUBE2, which has a hypercube topology, and is reported to perform well [8].

Task	CM5 time	AP1000 time
Idle	0.22	0.23
Communicating	0.97	0.99
Merging	0.75	1.24
Serial Sorting	3.17	4.57
Rearranging	0.38	0.59
Total	5.48	7.62

Table 1: **Sort times (seconds) for 8 million integers**

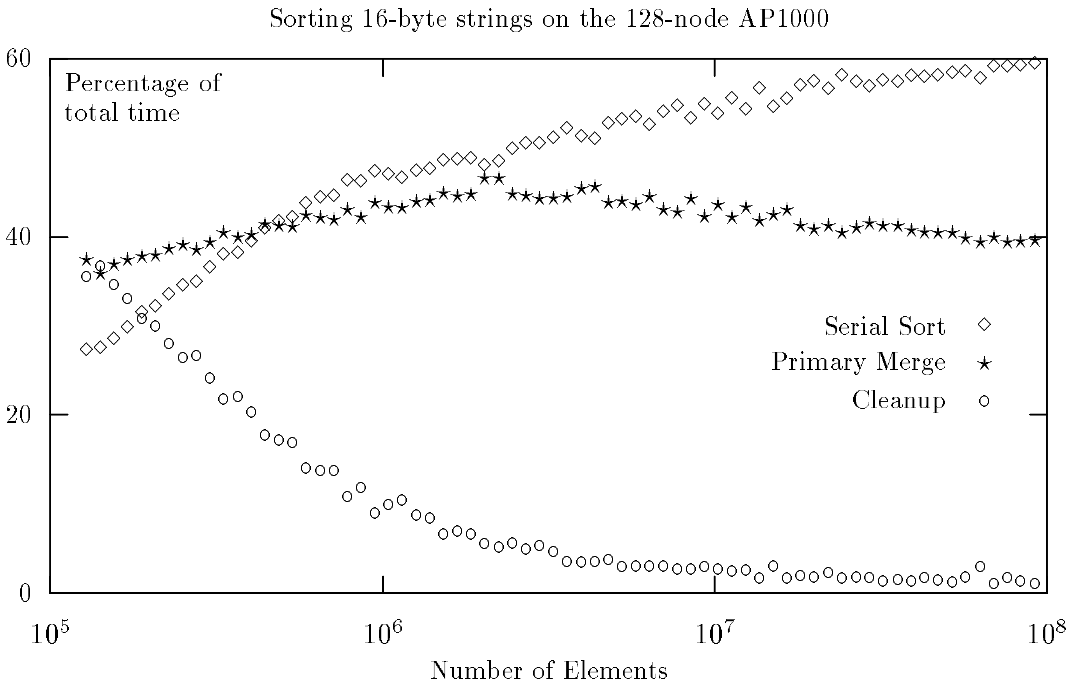

Figure 4: **Timing breakdown by phase**

4: Conclusions

We have presented a practical general-purpose parallel internal sorting algorithm that comes close to achieving the best possible speedup over an optimised serial algorithm. The algorithm is suitable for implementation on special-purpose, application-specific machines, or on general-purpose parallel machines. An implementation of the algorithm on two commercial machines has been discussed.

The algorithm derives its generality from the fact that it is comparison-based, and allows for a user-supplied comparison function. This corresponds to the commonly available serial sorting procedures that are the mainstay of internal sorting on serial computers.

The algorithm is frugal in its memory requirements, which allows data to be sorted almost to the limit of a parallel machine's memory. This is important, because it is unreasonable to expect data sets being sorted on a parallel machine to occupy only a small fraction of the machine's memory.

Acknowledgements

Support from Fujitsu Laboratories, Fujitsu Limited, and Fujitsu Australia Limited via the Fujitsu-ANU CAP Project is gratefully acknowledged. Andrew Tridgell was supported by an ATERB postgraduate scholarship.

References

[1] M. Ajtai, J. Kolmos and E. Szermeredi, "Sorting in $c \log n$ parallel steps", *Combinatorica* 3, 1983, 1-19.

[2] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, Toronto, 1985.

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith and M. Zagha, "A comparison of sorting algorithms for the Connection Machine CM-2", *Proc. Symposium on Parallel Algorithms and Architectures*, Hilton Head, South Carolina, July 1991.

[4] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors, Volume 1*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[5] H. Ishihata, T. Horie, S. Inano, T. Shimizu and S. Kato, "CAP-II Architecture", *Proc. First Fujitsu-ANU CAP Workshop* (edited by R. P. Brent and M. Ishii), Fujitsu Research Laboratories, Kawasaki, Japan, November 1990.

[6] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (second edition), Addison-Wesley, Menlo Park, 1981.

[7] L. Natvig, "Logarithmic Time Cost Optimal Parallel Sorting is Not Yet Fast in Practice!", *Proc. Supercomputing 90*, IEEE Press, 1990, 486-494.

[8] T. Rashid, personal communication, March 1993.

[9] H. H. Reif and L. G. Valiant, "A logarithmic time sort for linear size networks", *J. ACM* 34, 1987, 60-76.

[10] K. Thearling and S. Smith, "An Improved Supercomputing Sorting Benchmark", *Proc. Supercomputing 92*, IEEE Press, 1992, 14-19.

[11] A. Tridgell and R. P. Brent, *An Implementation of a General-Purpose Parallel Sorting Algorithm*, Report TR-CS-93-01, CS Lab, ANU, February 1993, 24 pp. Available by anonymous ftp from `andosl.anu.edu.au` (Internet number 150.203.15.95) in the directory `pub/tridge/sorting/par_sort`, and from `dcssoft.anu.edu.au` in the directory `pub/Brent`.

[12] B. B. Zhou, R. P. Brent and A. Tridgell, *Efficient Implementation of Sorting Algorithms on Asynchronous Distributed-Memory Machines*, Technical Report TR-CS-93-06, CS Lab, ANU, March 1993. Available by anonymous ftp from `dcssoft.anu.edu.au` in the directory `pub/Brent`.

[13] *CM-5 Technical Summary*, Thinking Machines Corporation, October 1991.