# A GENERAL-PURPOSE PARALLEL SORTING ALGORITHM*

ANDREW TRIDGELL and RICHARD P. BRENT
*Computer Sciences Laboratory*
*Australian National University*
*Canberra, ACT 0200, Australia*
{tridge,rpb}@cslab.anu.edu.au

ABSTRACT

A parallel sorting algorithm is presented for general purpose internal sorting on MIMD machines. The algorithm initially sorts the elements within each node using a serial sorting algorithm, then proceeds with 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 the Fujitsu AP1000 MIMD supercomputer is discussed.

*Keywords:* Batcher's merge-exchange sort, distributed memory, Fujitsu AP1000, nCUBE2, parallel sorting, sorting, Thinking Machines CM5

**1. Introduction.** Sorting of numeric or alphabetic data is required in many computing applications. Knuth [8] gives several examples in his classic work on serial sorting algorithms.

Many papers have discussed the task of sorting on parallel computers. See, for example, [1, 2, 12]. Most of these papers have dealt with the problem from a theoretical point of view, neglecting many issues that are important in a practical implementation of a parallel sorting algorithm [4, 10, 14]. This paper describes a practical parallel sorting algorithm which is suitable for efficient general-purpose internal sorting.

A description of the algorithm is given in Sections 2 and 3. In Section 4 the algorithm is compared with some other parallel sorting algorithms, such as radix sort and sample sort. Finally, the performance of our implementation on the Fujitsu AP1000 is discussed in Section 5.

The algorithm presented in this paper was first described in [16], which gives a more detailed account of the implementation of the algorithm and compares results on the AP1000 with results on the Thinking Machines CM5.

Knuth [8, p. 5] distinguishes between *internal sorting*, in which the records are kept in the computer's high-speed random-access memory, and *external sorting*, when there are more records than can be held in memory at once. In this paper we restrict our attention to internal sorting, although our algorithm could be extended in a straightforward way to handle external sorting.

**1.1. Nomenclature.** $P$ 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 $O(N/P)$.

The only operation assumed for 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 $i, j$.

* To appear in *Int. J. High Speed Computing* 7, 2 (1995), 285-301 (accepted 27 March 1994).
 rpb158 typeset using LaTeX

```
procedure hypercube_balance(integer base, integer num)
if num = 1 return
for all i in [0..num/2)
  pair_balance (base+i, base+i+(num+1)/2)
hypercube_balance (base+num/2, (num+1)/2)
hypercube_balance (base, num - (num+1)/2)
end
```

FIG. 1. *Pseudo-code for load balancing*

Logarithms are always to base 2, and division of integers in pseudo-code (e.g. Figures 1–2) is assumed to include an implicit truncation operation.

**2. The Sorting Algorithm.** The algorithm has four phases. The first and third phases are not strictly necessary, but are included because they greatly improve the efficiency of the algorithm. The second and third phases typically consume most of the processing time of the algorithm.

The first phase, called the pre-balancing phase, moves elements between the nodes without reference to the comparison function so as to achieve a more even distribution of the elements. This serves to reduce load imbalance problems later on. Pre-balancing can be omitted if the input data is known to be well-balanced.

The second phase is a serial-sorting phase, in which the elements of each node are sorted with an efficient serial sorting algorithm.

The third phase, called the primary merging phase, uses a pattern of merge-exchange operations between the nodes so that the elements are nearly sorted, in the sense that the sort may be completed with only a small amount of additional work.

The final phase, called the cleanup phase, uses a different pattern of merge-exchange operations which is guaranteed to complete the sort. The success of the primary merging phase in nearly sorting the data is judged by the amount of time the cleanup phase takes to complete.

Note that no post-balancing phase is required, because after the initial pre-balancing phase the algorithm is guaranteed to preserve a balanced distribution.

In Sections 2.1 to 2.4 below, we describe the purpose and implementation of each phase. In Section 3 we describe the merge-exchange operation that is used in both the primary merging and cleanup phases.

**2.1. Pre-Balancing.** The pre-balancing phase moves the elements between nodes so as to achieve a more even distribution of the elements. This phase is desirable to minimize 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 hypercube 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 $P$ is a power of 2 (see Lemma 1).

Pseudo-code for the pre-balancing algorithm is shown in Figure 1. When the algorithm is called, *base* is initially set to the index of the smallest node in the system, and *num* is set to $P$, the number of nodes. The algorithm operates recursively and takes $\lceil \log P \rceil$ steps to complete. When the number of nodes is not a power of 2, the effect is to have some of the nodes idle in some phases of the algorithm. Because the nodes that remain idle change with each step, all nodes take part in at least one pair-balance with another node. With some distributions of elements and values for $P$ it may be beneficial to iterate the pre-balancing algorithm until some balancing criterion is met.

As can be seen from the pseudo-code for the algorithm, the actual work of the balance is performed by another routine called *pair_balance*. This routine exchanges elements between a pair of nodes so that both nodes end up with as close as possible to the same number of elements. More precisely, if the nodes start with $N_j$ and $N_k$ elements, they end up with $\lfloor (N_j + N_k)/2 \rfloor$ and $\lceil (N_j + N_k)/2 \rceil$ elements.

Our statements about the pre-balancing phase can be made more precise by the following Lemma. The proof by induction on $d$ is straightforward, so it is omitted.

LEMMA 1. *If $P = 2^d$ and the number of elements in node $j$ is $N_j$ before the pre-balancing phase and $N'_j$ after the pre-balancing phase $(0 \le j < P)$, then*

$$\max_{0 \le j < k < P} |N'_j - N'_k| \le d \, .$$

```
procedure primary_merge(integer base, integer num)
if num = 1 return
for all i in [0..num/2)
  merge_exchange (base+i, base+i+(num+1)/2)
primary_merge (base+num/2, (num+1)/2)
primary_merge (base, num - (num+1)/2)
end
```

FIG. 2. *Pseudo-code for primary merge*

The worst case in Lemma 1 is illustrated by the example

$$N_j = \text{popcount}(j),$$

where popcount counts the number of 1-bits in the binary representation of its argument. In this example $N_0 = 0$, $N_{P-1} = d$, and the pre-balancing phase does nothing.

**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.

After some experimentation, a combination of quick-sort and insertion sort was chosen for serial sorting. This was found to give the best performance over a wide range of conditions on the AP1000.

As will be seen in Section 5, the time taken by the serial sorting phase of the algorithm dominates the total time taken by the sorting algorithm when the number of elements being sorted is large. It is thus very important that the serial sorting algorithm be optimized as much as possible. This optimization might include the use of an alternate serial sorting algorithm that can take advantage of particular forms of the comparison function between data elements. An example would be the use of a radix sorting algorithm for the sorting of ordinal data types such as integers.

**2.3. Primary Merging.** Since the aim of the primary merge phase of the algorithm is to "almost sort" the data in minimal time, significant improvements to the overall parallel efficiency could be attained by not employing a method guaranteed to fully sort the data; such methods as, for example, Batcher's merge-exchange sort [3], which is used in the cleanup phase, generally have a lower parallel efficiency.

The pattern of merge-exchange operations in the primary merge is identical to that used in the pre-balancing phase of the algorithm. The pseudo-code for the algorithm is given in Figure 2. When the algorithm is called, *base* is initially set to the index of the smallest node in the system, and *num* is set to $P$, the number of nodes.

This algorithm completes in $\lceil \log P \rceil$ steps per node, with each step consisting of a merge-exchange operation. As with the pre-balancing algorithm, if $P$ is not a power of 2 then some nodes may be left idle at each step of the algorithm.

If $P$ is a power of 2 and the initial ordering of the elements is random, then at each step of the algorithm each node has about the same amount of work to perform as the other nodes. In other words, the load balance between the nodes is very good. The symmetry is only broken by an unusual ordering of the original data, or if $P$ is not a power of 2. In both of these cases load imbalances may occur.

**2.4. Cleanup.** The cleanup phase of the algorithm is similar to the primary merge phase, but it must complete the sorting process. We chose Batcher's merge-exchange sort to achieve this, because the algorithm has some useful properties which make it ideal for a cleanup operation.

Pseudo-code for Batcher's merge-exchange sort is given in [8, Algorithm M, p. 112]. The method defines a pattern of comparison-exchange operations that sorts a list of elements of any length.

The way the algorithm is normally described, the comparison-exchange operation operates on two elements and exchanges the elements if the first element is greater than the second. In the application of the algorithm to the cleanup operation we generalize the notion of an element to include all elements in a node. This means that the comparison-exchange operation must make all elements in the second node greater than all elements in the first. This is identical to the operation of the merge-exchange algorithm, described in Section 3. A proof that it is possible to make this generalization while maintaining the correctness of the algorithm is given in [8, solution to problem 5.3.4.38]. This proof assumes that each of the nodes contains the same number of elements. A method for extending this result to the case of possibly different numbers of elements is given in Section 2.5.

Batcher's merge-exchange sort is ideal for the cleanup phase because it is very fast for almost sorted data. This is a consequence of a unidirectional merging property: the merge operations always operate in a direction so that the lower numbered node receives the smaller elements. This is not the case for some other fixed sorting networks. Algorithms without the unidirectional merging property are poorer choices for the cleanup phase as they tend to un-sort the data, undoing the work done by the primary merge phase, before completely sorting it. In practice the cleanup time is only a small fraction of the total sort time if Batcher's merge-exchange sort is used and the merge-exchange operation is implemented efficiently (see Figure 5).

**2.5. $N$ not a multiple of $P$.** The proof given in [8, solution to problem 5.3.4.38] shows that comparison-exchange based sorting algorithms can be extended to sort lists of elements by replacing the comparison-exchange operation with a merge-exchange operation. Simple examples demonstrate that the condition that each of the nodes have the same number of elements is necessary.

In [16] we describe a method called "infinity padding" which allows the cleanup phase of our algorithm to operate correctly when each node does not have the same number of elements. The method works by adding implicit $\infty$-elements to each of the nodes and calculating the number of real elements that must remain in each node after each merge-exchange operation.

In this section we describe an alternative to infinity padding which operates not in the merge-exchange operation but instead as an additional balancing phase coming directly after the pre-balancing phase of the algorithm. The effect of this change is to simplify the merge-exchange operation.

Let the maximum number of elements in any one node be $M$. We transfer $M - N_i$ elements from the highest numbered node to each of the lower numbered nodes in turn, starting with node 0. If the highest numbered node runs out of elements then we continue the process with the next highest numbered node, and so on.

Now, when a merge-exchange operation is performed between nodes $p$ and $q$ with $0 \leq p < q < P$, we can be sure that either $N_p = M$ and $N_q \leq N_p$, or that $N_q = 0$. This means that if we pad the nodes that have $N_p < M$ elements with $M - N_p$ $\infty$-elements, then we can guarantee that these $\infty$-elements will not move, and thus do not have to be explicitly represented.

This method relies on the fact that the sorting algorithm used for the cleanup phase is unidirectional in order to guarantee that the implicit $\infty$-elements always remain in the higher numbered nodes.

**3. Merge-Exchange.** The purpose of the merge-exchange operation is to exchange elements between two nodes so that we end up with one node containing elements that are all smaller than all the elements in the other node, while maintaining the order of the elements within the nodes. In practice the merge-exchange operation, along with the initial serial sort, takes the bulk of the processing time for a sort.

This section explains the merge-exchange portion of our overall parallel sorting algorithm, along with some special-case optimizations that are important for good performance.

To be useful in our sorting algorithm the merge-exchange operation must be very fast for data that is already nearly sorted so that the cleanup phase can complete quickly. The memory overhead must also be minimal so that the algorithm can efficiently utilize available memory.

In our implementation of parallel sorting we always require the node with the smaller node number to receive the smaller elements. This would not be possible if we used Batcher's bitonic algorithm (as in [5]) instead of his merge-exchange algorithm for the cleanup phase.

Suppose that a merge operation is needed between two nodes, $p$ and $q$, which initially contain $N_p$ and $N_q$ elements, respectively. For the discussion below (and without loss of generality) let us assume that the smaller elements are required in node $p$ after the merge.

In principle, merging two already-sorted lists of elements to obtain a completely sorted list is a very simple process. The most obvious implementation takes $N_p + N_q$ steps, with each step requiring one copy and one comparison operation [5, 8]. The problems with this approach are the storage requirements implied by the presence of a destination array. Using this algorithm as a component of a parallel sort would be restricted to lists whose maximum length fits in half the available memory of the machine. A more space-efficient algorithm can be developed. It is clear that such an algorithm must re-use the space that is freed by moving elements from the two source lists. We now describe how this can be done. The algorithm has several parts, each of which is described separately.

Our merge operation is similar to known algorithms for in-place merging [6, 9], but is also considerably simpler and faster due to its use of a temporary storage area of size $O(\sqrt{N/P})$. In practice the size of this temporary storage area is insignificant.

**3.1. Find-Exact Algorithm.** During a merge-exchange between two nodes, they will need access to each other's elements. The simplest method for doing this is for nodes to receive a complete copy of each other's elements before the merge begins.

A much better approach is to first determine exactly how many elements from each node will be required to complete the merge, and to transfer only those elements. This reduces the communications cost by minimizing the number of elements transferred, and at the same time reduces the memory overhead of the merge.

The find-exact algorithm allows each node to determine exactly how many elements are required from another node in order to produce the correct number of elements in a merged list.

When a comparison is made between elements $E_{p,A-1}$ and $E_{q,N_p-A}$ then the result of the comparison determines whether node $p$ will require more or less than $A$ of its own elements in the merge. If $E_{p,A-1}$ is greater than $E_{q,N_p-A}$, then the maximum number of elements that could be required to be kept by node $p$ is $A-1$, otherwise the minimum number of elements that could be required to be kept by node $p$ is $A$.

The proof of correctness relies on counting the number of elements that could be less than $E_{p,A-1}$. If $E_{p,A-1}$ is greater than $E_{q,N_p-A}$, then there are at least $N_p - A + 1$ elements in node $q$ that are less than $E_{p,A-1}$. If these are combined with the $A-1$ elements in node $p$ that are less than $E_{p,A-1}$, then we have at least $N_p$ elements less than $E_{p,A-1}$. This implies that the number of elements that must be kept by node $p$ is at most $A-1$.

A similar argument can be used to show that, if $E_{p,A-1} \leq E_{q,N_p-A}$, then the number of elements to be kept by node $p$ must be at least $A$. Combining these two results leads to a "bisection" algorithm that can find the exact number of elements required in at most $\lceil \log N_p \rceil$ steps by successively halving the range of possible values for the number of elements required to be kept by node $p$. Once this result is determined, it is a simple matter to derive the number of elements that must be sent from node $p$ to node $q$, and from node $q$ to node $p$.

On a machine with a high inter-processor communications latency, this algorithm could be costly, as a relatively large number of small messages are transferred. The number of messages could be reduced, but with a penalty of increased message size and algorithm complexity. One method for doing this is described in [17]. On the AP1000 the cost of the find-exact algorithm was found to be very small, due to the low inter-processor communications latency of the machine [7, 13], so the method of [17] was not implemented.

We assume for the remainder of the discussion on the merge-exchange algorithm that the find-exact algorithm has determined that node $p$ must retain $L_1$ elements and must receive $L_2$ elements from node $q$.

**3.2. Transferring Elements.** After the exact number of elements to be transferred has been determined, the transfer takes the form of an exchange of elements between the two nodes. The elements that are sent from node $p$ leave behind them spaces which must be filled with the incoming elements from node $q$. The reverse happens on node $q$, so the transfer process must be careful not to overwrite elements that have not yet been sent.

After the transfer is complete, the elements on node $p$ are stored in two contiguous sorted lists, of lengths $L_1$ and $L_2$ where $L_2$ is just $N_p - L_1$. In the remaining steps of the merge-exchange algorithm we merge these two lists so that all the elements are in order.

**3.3. Unbalanced Merging.** Before considering the algorithm for memory efficient merging, it is worth considering a special case where the result of the find-exact algorithm determines that the number of elements to be kept on node $p$ is much larger than the number of elements to be received from node $q$. In this case the task that node $p$ must undertake is to merge two lists of very different sizes. There is a very efficient algorithm for this special case, which may occur if the data is almost sorted, e.g., near the end of the cleanup phase.

Suppose that $L_1$ is much greater than $L_2$. First we determine where each of the $L_2$ elements that have been transferred from $q$ belongs in the list of length $L_1$. This can be done with at most $L_2\lceil \log L_1 \rceil$ comparisons using a binary search. As $L_2$ is small, this number of comparisons is small, and the results take only $O(L_2)$ storage.

Once this is done we can copy all the elements in list 2 to a temporary storage area and begin the process of moving elements from list 1 and list 2 to their proper destinations. This takes at most $L_1 + L_2$

element copies, but in practice it often takes only about $2L_2$ copies. This is because when only a small number of elements are transferred between nodes there is often only a small overlap between the ranges of elements in the two nodes, and only the elements in the overlap region have to be moved.

The unbalanced merge is faster in practice than a general purpose merge algorithm, and the overall performance of the sorting procedure is significantly better than it would be if we did not take advantage of this special case.

**3.4. Blockwise Merging.** The blockwise merge is a solution to the problem of merging two sorted lists of elements into one, while using only a small amount of additional storage. The first phase in the operation is to break the two lists into blocks of an equal size $B$. The exact value of $B$ is unimportant for the correctness of the algorithm but does influence the efficiency and memory usage. We assume that $B$ is $O(\sqrt{L_1 + L_2})$, which is small relative to the memory available on each node. To simplify the exposition we also assume, for the time being, that $L_1$ is a multiple of $B$.

The merge takes place by merging from the two blocked lists of elements into a destination list of blocks. When a block from one of the two source lists becomes empty it is added to the list of destination blocks. The destination list is initially primed with two empty blocks.

As the merge proceeds there are always exactly $2B$ free spaces in the three lists. This implies that there must always be at least one free block on the destination list whenever a new destination block is required.

The algorithm actually takes no more steps than the simple merge mentioned earlier. Each element moves only once. The drawback, however, is that the elements end up in a blocked list structure rather than in a simple linear array.

The simplest method for resolving this problem, at the expense of additional memory accesses, is to rearrange the blocks into standard form. This is what has been done in our current implementation. It would be possible to modify the algorithm so that the elements are always kept in a block list form. This would improve performance at the cost of some additional coding complexity.

So far, we have assumed that $L_1$ is a multiple of $B$. If $L_1$ is not a multiple of $B$ then blocks in the second source list cannot be used as destination blocks without leaving a gap in the final list. To overcome this problem we copy $L_1 \bmod B$ elements from the tail of list 1 and use this copy as a final source block. Then we offset the blocks when transferring them from source list 2 to the destination list so that they end up aligned on the proper boundaries. Finally, we increase the amount of temporary storage to $3B$ and prime the destination list with three blocks to account for the fact that we cannot use the partial block from the tail of list 1 as a destination block.

**4. Comparison with other algorithms.** When designing the parallel sorting algorithm we wished to produce a general-purpose algorithm, similar in functionality to the many general-purpose serial sorting algorithms that already exist. Specifically, we aimed for the following properties:

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

A parallel algorithm based on radix sorting has been shown to perform very well [4], but it relies on the ordinal properties of the data type being sorted. This limits its application as it cannot be used to sort with an arbitrary comparison function. Another problem is memory utilization. Radix sorting, in its usual implementation, requires the use of additional workspace at least as large as the data being sorted. This limits the number of elements that can be sorted to half that which fits in the machine's memory.

An advantage of radix sorting is that it can be adapted relatively easily to take advantage of any vector processing capabilities of the processors. This can give a large performance gain in situations where radix sorting is possible. We have achieved some success in combining the general purpose algorithm presented in this paper with a radix sorting algorithm. The performance improvement which results comes at a cost of decreased generality.

The parallel sample sort algorithm, as with radix sorting, has a problem with high memory requirements [4]. It also fails to satisfy property 4 due to its reliance on taking pseudo-random samples of the input data.

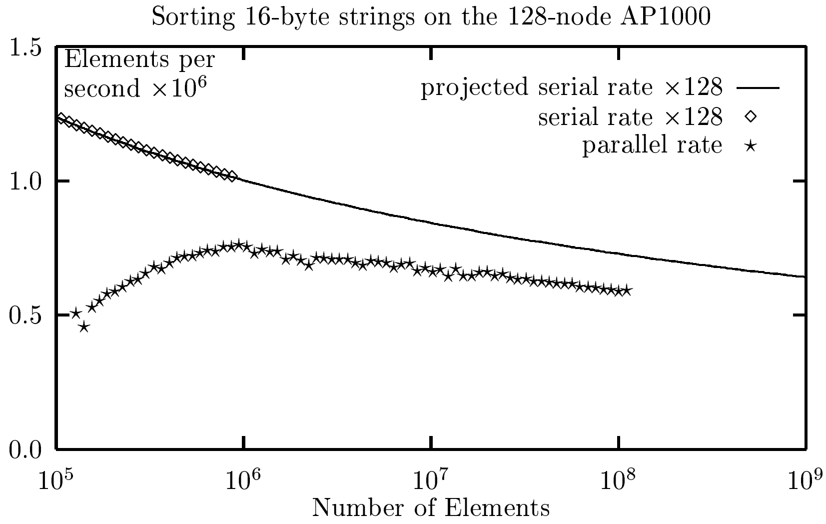


FIG. 3. *Sorting 16-byte strings on the AP1000*

Parallel versions of radix sort [4] and sample sort [4, 14] also require "all-to-all" communications, which is much more costly in terms of communication overhead on current parallel machines than the transfer of large contiguous blocks of elements as is used in our algorithm.

Several methods that rely on the number of elements being a multiple of the number of nodes or on the number of nodes being a power of 2 do not satisfy property 3 (see [2]).

**5. Performance.** Several runs were made on a AP1000 [7] to examine the performance of our implementation of the sorting algorithm under a variety of conditions. The AP1000 used contains 128 nodes connected in an 8 by 16 torus network topology, node to node communication on which is performed in hardware using wormhole routing. Each node comprises a 25 MHz Sparc 1+ processor with 16MByte of local memory. Not all of the local memory is available to applications, due to the presence of a minimal operating system on each node. In practice just over 12MByte is available for program use. The AP1000 does not use virtual memory.

The parallel sort algorithm was implemented in the C language using a library of message passing routines and was compiled with the GNU C compiler (version 2.4.5). The code was written so that it could be easily ported to other parallel machines such as the Thinking Machines CM5 [18].

An evaluation of the algorithm has been performed on the CM5. The results are given in [16]. They are comparable to the results obtained on the AP1000, though with some differences due to the different machine architectures and operating systems.

Our code has also been ported to the nCUBE2 [11]. The results for this machine show a greater parallel speedup than for either the AP1000 or CM5. This is due to the hypercube connections of the nCUBE2, which suit our algorithm, along with a high ratio of network speed to processing speed.

When we refer to the parallel speedup of the algorithm we mean the ratio of the time the sorting task takes to complete with the best available serial algorithm on a single node of the parallel machine to the time for our parallel algorithm. The fact that our algorithm naturally reduces to the best available serial algorithm when run on one node of the parallel machine makes the comparison simpler.

Because the algorithm is comparison-based its performance is not dependent on the statistical distribution or entropy [15] of the test data. It is influenced only by the initial ordering of the elements and the number of elements on each node. This simplifies the task of generating suitable test data sets, as a random element generator supplying a uniform distribution can be used.

**5.1. Timing Results.** The first results are shown in Figure 3. This figure shows the performance of the algorithm on the 128-node AP1000 as the number of elements, $N$, spans a wide range of values, from values which would be easily dealt with on a workstation, to those at the limit of the AP1000's memory capacity (2GByte). The elements are 16-byte random strings. The comparison function is the C library function `strcmp()`.

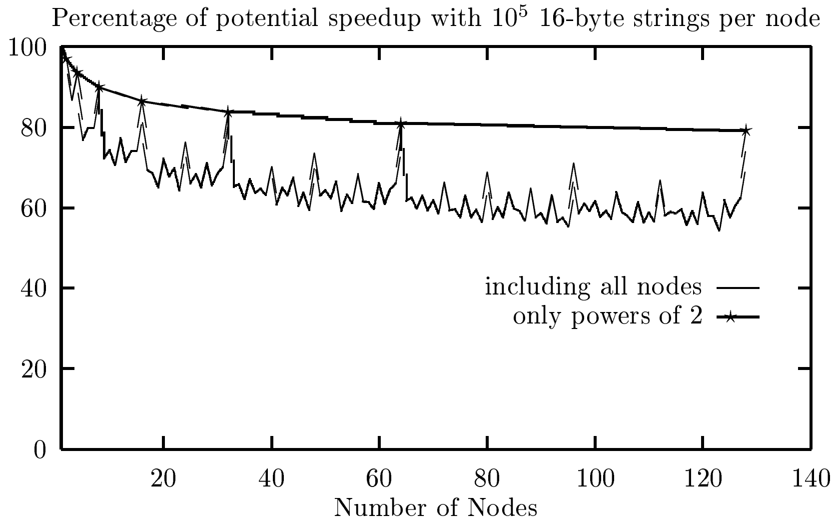


FIG. 4. *Scalability of sorting on the AP1000*

The results give the number of elements that can be sorted per second of real time. This time includes all phases of the algorithm and gives an overall indication of performance.

Shown on the same graph is the performance of a hypothetical serial computer that operates $P$ times as fast as one of the $P$ individual nodes of the AP1000. The hypothetical sorting performance is calculated as the product of $P$ and the measured, single-node performance. Due to memory limitations on the individual nodes an extrapolation is necessary to estimate the time a single node would take to perform large sorts. A least squares fit was used to obtain the values of $a$, $b$, $c$ and $d$ in the function:

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

This function was chosen because it covers the significant contributions in the quick-sort algorithm and was found empirically to extrapolate very accurately on machines with large amounts of single processor memory.

The graph shows that the performance of the sorting algorithm increases quickly as the number of elements approaches 1 million. After this a slow fall-off occurs, closely following the profile of the ideal parallel speedup. The roll-off point of 1 million elements corresponds to the number of elements that can be held in the 128KByte cache of each node. This indicates the importance of caching to the performance of the algorithm.

It is encouraging to note how close the algorithm comes to the ideal speedup of $P$ for a $P$-node machine. The algorithm achieves 85% of the ideal performance for a 128-node machine. The dependence of this result on the value of $P$ is discussed in the next section.

**5.2. Scalability.** An important aspect of a parallel algorithm is its scalability, which is a measure of the algorithm's ability to utilize additional nodes. Figure 4 depicts the scalability on the AP1000 (as a percentage of the potential speedup) of sorting 100,000 16-byte strings per node versus the number of nodes. The number of elements per node is kept constant to ensure that caching factors do not influence the result.

The points in the graph are connected because this shows more clearly the pattern of dependence on the value of $P$. No significance should be inferred from the interpolation of the points.

The left-most data point shows the speedup for a single node. This is equal to 1 as the algorithm reduces to our optimized quick-sort when only a single node is used. As the number of nodes increases, the proportion of the ideal speedup decreases, as communication costs and load imbalances begin to appear. The graph flattens out for larger numbers of nodes, which indicates that the algorithm should have a good efficiency when the number of nodes is large.

The two curves in the graph 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 2 in the algorithm. It is also clear that certain values for $P$ are preferred to

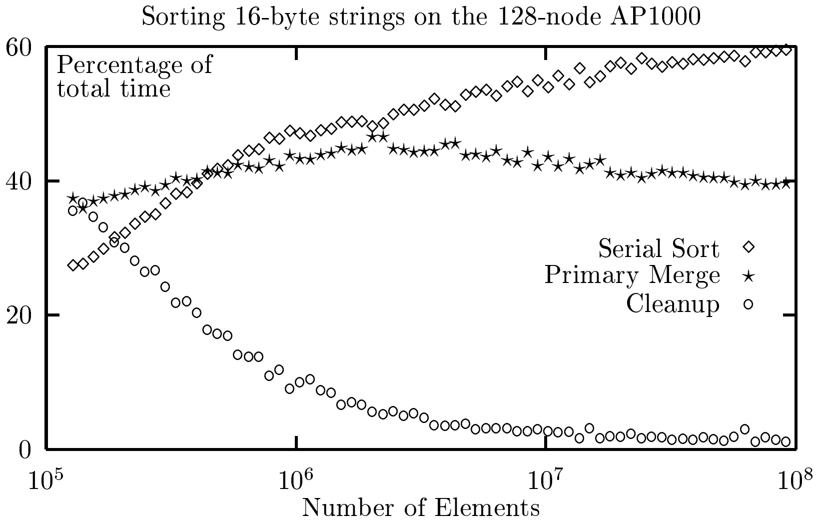


FIG. 5. *Timing breakdown by phase*

others. In particular, the parallel sort algorithm performs better on even numbers of nodes than on odd ones. Sums of adjacent powers of 2 also seem to be preferred, so that when $P$ takes on values of 24, 48 and 96 the efficiency is quite high.

**5.3. Where The Time Goes.** It is interesting to look at the proportion of the total time spent in certain phases of the sort, as is shown in Figure 5 sorting 16-byte strings on the AP1000 over a wide range of list lengths $N$. The figure depicts results for the initial serial sort, primary merge, and cleanup phases of the algorithm.

This graph shows that as $N$ increases to a significant proportion of the available memory of the machine the greatest time is taken by the initial serial sorting of the elements in each cell. This is the case because this phase of the algorithm takes time $O(N \log N)$ whereas all other phases of the algorithm take time $O(N)$ or lower. It is the fact that this component of the algorithm is able to dominate the time while $N$ is still a relatively small proportion of the capacity of the machine, which leads to the practical efficiency of the algorithm. Many sorting algorithms are asymptotically optimal in the sense that their speedup approaches $P$ for large $N$, but few can get close to this speedup for values of $N$ that are of interest in practice [10].

It is also interesting to note the small impact that the cleanup phase has for larger values of $N$. This demonstrates that the primary merge phase does indeed produce an almost sorted data set and that the cleanup phase can take advantage of this.

The time spent by the overall algorithm in various tasks can also be measured. In this case we look at what kind of operation each of the nodes is performing, which provides a finer division of the time.

Figure 6 exhibits the timings for the various tasks on the 128-node AP1000 for sorting a wide range of list lengths $N$. Again it is clear that the serial sorting dominates for large values of $N$, for the same reasons as before. What is more interesting is that the proportion of time spent idling while waiting for messages and in actual communication, decreases steadily as $N$ increases. From the point of view of the parallel speedup of the algorithm, such tasks waste valuable time, which needs to be kept to a minimum.

**6. Conclusions.** We have presented a practical general-purpose parallel internal sorting algorithm that, in at least one implementation, comes close to achieving the best possible speedup over an optimized serial algorithm.

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.

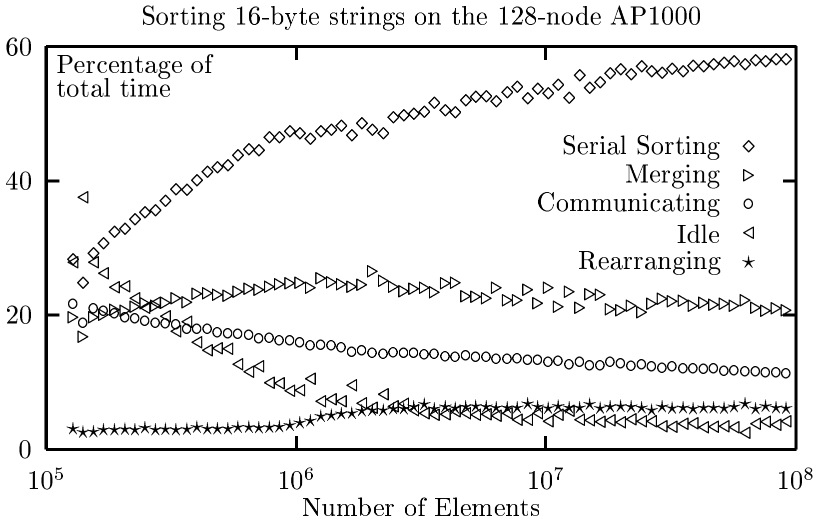


FIG. 6. *Timing breakdown by task*

**7. Availability.** Source code for the algorithm is available via anonymous ftp from `nimbus.anu.edu.au` (Internet number 150.203.15.21) in the directory `pub/tridge/sorting/par_sort`. Also included in this directory is a technical report [16]. The authors may be contacted by e-mail at `tridge@cslab.anu.edu.au` and `rpb@cslab.anu.edu.au` .

**Acknowledgments.** We thank the referees for their help in improving the exposition. Support by Fujitsu Laboratories, Fujitsu Limited, and Fujitsu Australia Limited via the Fujitsu-ANU CAP Project is gratefully acknowledged. Support of Andrew Tridgell via an ATERB postgraduate scholarship is also gratefully acknowledged.

REFERENCES

[1] M. AJTAI, J. KOMLÓS AND E. SZEMEREDI, *Sorting in $c \log n$ parallel steps*, Combinatorica 3, 1983, 1-19.
[2] S. G. AKL, *Parallel Sorting Algorithms*, Academic Press, Toronto, 1985.
[3] K. E. BATCHER, *Sorting networks and their applications*, Proc. AFIPS Spring Joint Computer Conference 32, 1968, 307–314.
[4] 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.
[5] 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.
[6] B.-C. HUANG AND M. A. LANGSTON, *Practical in-place merging*, Communications of the ACM 31, 1988, 348-352.
[7] H. ISHIHATA, T. HORIE AND T. SHIMIZU, *Architecture for the AP1000 highly parallel computer*, Fujitsu Sci. Tech. J. 29, 1993, 6-14.
[8] D. E. KNUTH, *The Art of Computer Programming, Volume 3: Sorting and Searching* (second edition), Addison-Wesley, Menlo Park, 1981.
[9] M. A. KRONROD, *An optimal ordering algorithm without a field of operation*, Dokl. Akad. Nauk SSSR 186, 1969, 1256-1258 (in Russian).
[10] L. NATVIG, *Logarithmic Time Cost Optimal Parallel Sorting is Not Yet Fast in Practice!*, Proc. Supercomputing '90, IEEE Press, 1990, 486-494.
[11] T. RASHID, personal communication, March 1993.
[12] J. H. REIF AND L. G. VALIANT, *A logarithmic time sort for linear size networks*, J. ACM 34, 1987, 60-76.
[13] T. SHIMIZU, T. HORIE AND H. ISHIHATA, *Performance evaluation of the AP1000*, Fujitsu Sci. Tech. J. 29, 1993, 15-24.
[14] H. SHU AND J. SCHAEFFER, *Parallel sorting by regular sampling*, J. Parallel and Distributed Computing 14, 1992, 361-372.

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

[16] A. Tridgell and R. P. Brent, *An Implementation of a General-Purpose Parallel Sorting Algorithm*, Report TR-CS-93-01, Computer Sciences Lab, Australian National University, February 1993, 24 pp. Available by anonymous ftp.

[17] B. B. Zhou, R. P. Brent and A. Tridgell, *Efficient Implementation of Sorting Algorithms on Asynchronous Distributed-Memory Machines*, Report TR-CS-93-06, Computer Sciences Lab, Australian National University, March 1993, 7 pp.

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