# AN IMPLEMENTATION OF A GENERAL-PURPOSE PARALLEL SORTING ALGORITHM

**Andrew Tridgell and Richard P. Brent**

**Computer Sciences Laboratory**
**Australian National University**
**TR-CS-93-01**
**February 1993**

**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 is examined on two MIMD machines, the Fujitsu AP1000 and the Thinking Machines CM5.

# Table of Contents

# 1 THE PARALLEL SORTING TASK

## 1.1 Introduction

Many papers have discussed the task of sorting on parallel computers. Most of these have dealt with the problem from a theoretical point of view, neglecting many issues which are important in a practical implementation of a parallel sorting algorithm [6]. This report introduces a practical parallel sorting algorithm which is suitable for efficient general-purpose internal sorting.

An overview of the algorithm is given in Section 2. Considerably more details are given in Section 3. Finally, the performance of our implementations on two MIMD parallel machines is discussed in Section 4.

## 1.2 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 $p$ ($0 \le p < P$). To avoid double subscripts we abbreviate $N_{p_j}$ to $N_j$ where no confusion should arise.

Elements within each node of the machine are referred to as $E_{p,i}$, for $0 \le i < N_p$ and $0 \le p < P$. We may write $E_{j,i}$ instead of $E_{p_j,i}$ if no confusion will arise.

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 $i,j$.

The speedup offered by a parallel algorithm for sorting $N$ elements is defined as the ratio of the time to sort $N$ elements with the fastest known serial algorithm (on one node of the parallel machine) to the time taken by the parallel algorithm on the parallel machine.

## 1.3 Aims of the Algorithm

In designing the algorithm we had several aims.

- Speed
- Good memory utilisation. The number of elements that can be sorted should closely approach the physical limits of the machine.

- Flexibility, so that no restrictions are placed on $N$ and $P$. In particular $N$ should not need to be a multiple of $P$ or a power of 2, which are common restrictions in parallel sorting algorithms [1].

In order for the algorithm to be truly general purpose we restricted ourselves to algorithms which relied only on binary comparisons between elements. This rules out methods such as radix sorting, which can be very fast when they are applicable, but are dependent on a short key length and good data distribution [7].

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 Unix 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.

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 the size of $N$ and $P$ are the physical constraints of the machine.

The algorithm presented here is similar in many respects to parallel shellsort [2], but contains a number of new features. For example, the memory overhead of the algorithm is considerably reduced.

## 1.4 Hardware

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

The first machine is a 128-node AP1000 produced by Fujitsu [3]. 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.

The second machine is a 32-node CM5 produced by Thinking Machines Corporation [8]. 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 implementation differences in the individual nodes of the two machines which are significant for our algorithm.

- 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 this topology, so communication patterns which would not cause network contention on a hypercube may cause contention on the CM5 or the AP1000. It turns out that this does not have a serious impact on performance (see Section 4.5).

# 2 OVERVIEW OF 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 2.1 to 2.5 below, we outline the purpose and implementation of each phase, and of 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.

## 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 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 the number of nodes is a power of 2.

The details of the pre-balance are discussed in Section 3.1, along with a method for reducing the cost of this process by tokenising the movement of the elements. In practice, the pre-balancing phase usually consumes only a small proportion of the overall sorting time.

## 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, the serial sorting method chosen was a combination of quicksort and insertion sort. 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().

## 2.3 Primary Merging

The aim of the primary merging phase of the algorithm is to almost completely sort the data in a very efficient manner. 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. 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 $\log P$ 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 to almost nil and allows the algorithm to achieve a high parallel efficiency.

It is possible to omit the primary merging phase, but it has been found that this increases the overall sorting time.

## 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 algorithm [4] (not to be confused with Batcher's bitonic sorting algorithm). The algorithm is actually a generalisation of Batcher's merge-exchange algorithm (as usually described), 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 [5]. The algorithm defines a pattern of merge-exchange operations which will 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 Section 3.6, the algorithm is very efficient if the data is almost sorted. Thus, in practice the cleanup is found to take only a small proportion of the total time (see Section 4.4).

## 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 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.

The number of elements in each of the nodes must be controlled for the algorithm to function correctly. The number is determined by a method called "infinity padding" (Section 3.1), which in practice leads

to only minor changes to the distribution produced by the pre-balancing phase.

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 $\log(N/P)$ 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.

Finally, the merge itself is performed. Although it is a trivial matter to merge two sorted lists into one if a generous amount of additional storage is assumed, it is more difficult to merge them with minimal additional storage. We developed an 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 Section 3.5.

# 3 IMPLEMENTATION DETAILS

In this Section we describe in more detail the implementation of each phase of the algorithm.

## 3.1 Infinity Padding

In order for a parallel sorting algorithm to be useful as a general-purpose routine, arbitrary restrictions on the number of elements that can be sorted must be removed. It is unreasonable to expect that the number of elements $N$ should be a multiple of the number of nodes $P$.

The proof given in [5] shows that sorting networks will correctly sort lists of elements provided the number of elements in each list is equal, and the comparison-exchange operation is replaced with a merge-exchange operation. The restriction to equal-sized lists is necessary, as simple examples show. However, a simple extension of the algorithm, which we call *infinity padding*, can remove this restriction.

First let us define $M$ to be the maximum number of elements in any one node. It is clear that it would

be possible to pad each node with $M–N_p$ dummy elements so that the total number of elements would become $M×P$. This would mean that each node has $M$ elements after padding which presents no problems to sorting networks. After sorting is complete these padding elements could be found and removed in the resulting sorted list.

Infinity padding is a variation on this theme. We notionally pad each node with $M–N_p$ "infinity" elements. These elements are assumed to have the property that they compare greater than any elements in any possible data set. If we now consider one particular step in the sorting algorithm then we shall see that these infinity elements need only be represented implicitly.

Say nodes $p_1$ and $p_2$ have $N_1$ and $N_2$ elements respectively before being merged in our algorithm, with node $p_1$ receiving the smaller elements. Then the addition of infinity padding elements will result in $M–N_1$ and $M–N_2$ infinity elements being added to nodes $p_1$ and $p_2$ respectively. We know that, after the merge, node $p_2$ must contain the largest $M$ elements, so we can be sure that it will contain all of the infinity elements up to a maximum of $M$. From this we can calculate the number of real elements which each node must contain after merging. If we designate the number of real elements after merging as $N'_1$ and $N'_2$ then we find that

$$N'_2 = MAX(0, N_1 + N_2 - M) \text{ and}$$
$$N'_1 = N_1 + N_2 - N'_2$$

This means that if at each merge phase we give node $p_1$ the first $N'_1$ elements and node $p_2$ the remaining elements, we have implicitly performed padding of the nodes with infinity elements, thus guaranteeing the correct behavior of the algorithm.

## 3.2 Balancing

The aim of the balancing phase of the algorithm is to produce a distribution of the elements on the nodes that approaches as closely as possible $N/P$ elements per node.

The algorithm chosen for this task is one which reduces to a hypercube for values of $P$ which are a

```
subroutine hypercube_balance(integer base,integer num)

if num = 1 stop

for all i in [0..num/2)
     call pair_balance(base+i,base+i+(num+1)/2)

call hypercube_balance(base+num/2,(num+1)/2)
call hypercube_balance(base,num - (num+1)/2)

endsubroutine
```

Figure 1. Pseudo-code for load balancing phase

power of 2. The pseudo-code for the algorithm is shown in Figure 1. When the algorithm is called, the *base* is initially set to the index of the smallest node in the system and *num* is set to the number of nodes, *P*. The algorithm operates recursively and takes log*P* steps to complete. When the number of nodes is not a power of 2, the effect is to have one of the nodes idle in some phases of the algorithm. Because the node which remains idle changes with each step, all nodes take part in a pair-balance with another node.

As can be seen from the code for the algorithm, the actual work of the balance is performed by another routine called *pair_balance*. This routine is designed to exchange elements between a pair of nodes so that both nodes end up with the same number of elements, or as close as possible. If the total number of elements shared by the two nodes is odd then the node with the lower node number gets the extra element. Consequently if the total number of elements *N* is less than the number of nodes *P*, then the elements tend to gather in the lower numbered nodes.

A slight modification can be made to the balancing algorithm in order to improve the performance of the merge-exchange phase of the sorting algorithm. As discussed in Section 3.1, infinity padding is used to determine the number of elements to remain in each node after each merge-exchange operation. If this results in a node having less elements after a merge than before then this can lead to complications in the merge-exchange operation and a loss of efficiency.

To ensure that this never happens we can take advantage of the fact that all merge operations in the primary merge and in the cleanup phase are performed in a direction such that the node with the smaller index receives the smaller elements, a property of the sorting algorithm used. If the node with the smaller index has more elements than the other node, then the virtual infinity elements are all required in the other node, and no transfer of real elements is required. This means that if a final balancing phase is introduced where elements are drawn from the last node to fill the lower numbered nodes equal to the node with the most elements, then the infinity padding method is not required and the number of elements on any one node need not change.

As the number of elements in a node can be changed by the *pair_balance* routine it must be possible for the node to extend the size of the allocated memory block holding the elements. This leads to a restriction in the current implementation to the sorting of blocks of elements that have been allocated using the standard memory allocation procedures. It would be possible to remove this restriction by allowing elements within one node to exist as two non-contiguous blocks of elements, and applying an un-balancing phase at the end of the algorithm. This idea has not been implemented because its complexity outweighs its relatively minor advantages.

In the current version of the algorithm elements may have to move up to log*P* times before reaching their destination. It might be possible to improve the algorithm by arranging that elements move only once in reaching their destination. Instead of moving elements between the nodes, tokens would be sent to represent blocks of elements along with their original source. When this virtual balancing was completed, the elements could then be dispatched directly to their final destinations. This tokenised balance has not been implemented, primarily because the balancing is sufficiently fast without it.

## 3.3 Serial Sorting

The aim of the serial sorting phase is to order the elements in each node in minimum time. For this task, the best available serial sorting algorithm has been used, subject to the restriction that the algorithm

must be comparison based.

If the number of nodes is large then another factor must be taken into consideration in the selection of the most appropriate serial sorting algorithm. A serial sorting algorithm is normally evaluated using its average case performance, or sometimes its worst case performance. The worst case for algorithms such as quicksort is very rare, so the average case is more relevant in practice. However, if there is a large variance, then the serial average case can give an over-optimistic estimate of the performance of a parallel algorithm. This is because a delay in any one of the nodes may cause other nodes to be idle while they wait for the delayed node to catch up.

This suggests that it may be safest to choose a serial sorting algorithm such as heapsort, which has worst case equal to average case performance. However, we found that the parallel algorithm performed better on average when the serial sort was quicksort (for which the average performance is good and the variance small) than when the serial sort was heapsort.

Our final choice is a combination of quicksort and insertion sort. The basis for this selection was a number of tests carried out on implementations of several algorithms. The care with which the algorithm was implemented was at least as important as the choice of abstract algorithm.

Our implementation is based on code written by the Free Software Foundation for the GNU project. Several modifications were made to give improved performance For example, the insertion sort threshold was tuned to provide the best possible performance for the Sparc architecture.

## 3.4 Primary Merge

The aim of the primary merge phase of the algorithm is to almost sort the data in minimum time. For this purpose an algorithm with a very high parallel efficiency was chosen to control merge-exchange operations between the nodes. This led to significant performance improvements over the use of an algorithm with lower parallel efficiency that is guaranteed to completely sort the data (for example, Batcher's algorithm as used in the cleanup phase).

The pattern of merge-exchange operations in the primary merge is identical to that used in the

```
subroutine primary_merge(integer base,integer num)

if num = 1 stop

for all i in [0..num/2)
     call merge_exchange(base+i,base+i+(num+1)/2)

call primary_merge(base+num/2,(num+1)/2)
call primary_merge(base,num - (num+1)/2)

endsubroutine
```

Figure 2. Pseudo-code for primary merge phase

pre-balancing phase of the algorithm. The pseudo-code for the algorithm is given in Figure 2. When the algorithm is called the *base* is initially set to the index of the smallest node in the system and *num* is set to the number of nodes, *P*.

This algorithm completes in log*P* steps per node, with each step consisting of a merge-exchange operation. As with the load balancing algorithm, if *P* is not a power of 2 then a single node may be left idle at each step of the algorithm, with the same node never being left idle twice in a row.

If *P* is a power of 2 and the initial distribution 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 due to an unusual distribution of the original data, or if *P* is not a power of 2. In both these cases load imbalances may occur.

## 3.5 Merge-Exchange Operation

The aim of the merge-exchange algorithm is to exchange elements between two nodes so that we end up with one node containing elements which are all smaller than all the elements in the other node, while maintaining the order of the elements in the nodes. 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 instead of his merge-exchange algorithm.

Secondary aims of the merge-exchange operation are that it should be very fast for data that is almost sorted already, and that the memory overhead should be minimised.

```
subroutine merge(list destination, list source1, list source2)

while (source1 not empty) and (source2 not empty)
     if (top_of_source1 < top_of_source_2)
       put top_of_source1 into destination
     else
       put top_of_source2 into destination
     endif
endwhile

while (source1 not empty)
     put top_of_source1 into destination
endwhile

while (source2 not empty)
     put top_of_source2 into destination
endwhile

endsubroutine
```

Figure 3. Pseudo-code for a simple merge

Suppose that a merge operation is needed between two nodes, $p_1$ and $p_2$, which initially contain $N_1$ and $N_2$ elements respectively. We assume that the smaller elements are required in node $p_1$ after the merge.

In principle, merging two already sorted lists of elements to obtain a new sorted list is a very simple process. The pseudo-code for the most natural implementation is shown in Figure 3. This algorithm completes in $N_1+N_2$ steps, with each step requiring one copy and one comparison operation. The problem with this algorithm is the storage requirements implied by the presence of the destination array. This means that the use of this algorithm as part of a parallel sorting algorithm would restrict the number of elements that can be sorted to the number that can fit in half the available memory of the machine. The question then arises as to whether an algorithm can be developed that does not require this destination array.

In order to achieve this, it is clear that the 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.

The principle of infinity padding is used to determine how many elements will be required in each of the nodes at the completion of the merge operation. If the complete balance operation has been performed at the start of the whole algorithm then the result of this operation must be that the nodes end up with the same number of elements after the merge-exchange as before. We assume that infinity padding tells us that we require $N'_1$ and $N'_2$ elements to be in nodes $p_1$ and $p_2$ respectively after the merge.

### 3.5.1 Find-Exact Algorithm

When a node takes part in a merge-exchange with another node, it will need to be able to access the other nodes elements as well as its own. The simplest method for doing this is for each node to receive a copy of all of the other nodes 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 minimising 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 element $E_{1,A-1}$ and $E_{2,N'_1-A}$ then the result of the comparison determines whether node $p_1$ will require more or less than $A$ of its own elements in the merge. If $E_{1,A-1}$ is greater than $E_{2,N'_1-A}$ then the maximum number of elements that could be required to be kept by node $p_1$ is $A-1$, otherwise the minimum number of elements that could be required to be kept by node $p_1$ is $A$.

The proof that this is correct relies on counting the number of elements that could be less than $E_{1,A-1}$. If $E_{1,A-1}$ is greater than $E_{2,N'_1-A}$ then we know that there are at least $N'_1-A+1$ elements in node $p_2$ that are less than $E_{1,A-1}$. If these are combined with the $A-1$ elements in node $p_1$ that are less than $E_{1,A-1}$, then we have at least $N'_1$ elements less than $E_{1,A-1}$. This means that the number of elements

that must be kept by node $p_1$ must be at most $A$–1.

A similar argument can be used to show that if $E_{1,A-1}$ is less than or equal to $E_{2,N'_1-A}$ then the number of elements to be kept by node $p_1$ must be at least $A$. Combining these two results leads to an algorithm that can find the exact number of elements required in at most $\log N_1$ steps by successively halving the range of possible values for the number of elements required to be kept by node $p_1$.

Once this result is determined it is a simple matter to derive from this the number of elements that must be sent from node $p_1$ to node $p_2$ and from node $p_2$ to node $p_1$.

On a machine with a high message latency, this algorithm could be costly, as a relatively large number of small messages are transferred. The cost of the algorithm can be reduced, but with a penalty of increased message size and algorithm complexity. To do this the nodes must exchange more than a single element at each step, sending a tree of elements with each leaf of the tree corresponding to a result of the next several possible comparison operations. This method has not been implemented as the practical cost of the find-exact algorithm was found to be very small on the CM5 and AP1000.

We assume for the remainder of the discussion on the merge-exchange algorithm that after the find exact algorithm has completed it has been determined that node $p_1$ must retain $L_1$ elements and must transfer $L_2$ elements from node $p_2$.

### 3.5.2 Transferring Elements

After the exact number of elements to be transferred has been determined, the actual transfer of elements can begin. The transfer takes the form of an exchange of elements between the two nodes. The elements that are sent from node $p_1$ leave behind them spaces which must be filled with the incoming elements from node $p_2$. The reverse happens on node $p_2$ so the transfer process must be careful not to overwrite elements that have not yet been sent.

The implementation of the transfer process was straightforward on the CM5 and AP1000 because of appropriate hardware/operating system support. On the CM5 a routine called CMMD_send_and_receive does just the type of transfer required, in a very efficient manner. On the AP1000 the fact that a non-blocking message send is available allows for blocks of elements to be sent simultaneously on the two nodes, which also leads to a fast implementation.

If this routine were to be implemented on a machine without a non-blocking send then each element on one of the nodes would have to be copied to a temporary buffer before being sent. The relative overhead that this would generate would depend on the ratio of the speeds of data transfer within nodes and between nodes.

After the transfer is complete, the elements on node $p_1$ are in two contiguous sorted lists, of lengths $L_1$ and $N'_1$–$L_1$. In the remaining steps of the merge-exchange algorithm we merge these two lists so that all the elements are in order.

### 3.5.3 Unbalanced Merging

Before considering the algorithm that has been devised for minimum memory 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 $p_1$ is much larger than the number of elements to be transferred from node

$p_2$.

In this case the task which node $p_1$ must undertake is to merge two lists of very different sizes. There is a very efficient algorithm for this special case.

Suppose that $L_1$ is much greater than $L_2$. This may occur if the data is almost sorted, for example, near the end of the cleanup phase. We proceed as follows.

First we determine, for each of the $L_2$ elements that have been transferred from $p_1$, where it belongs in the list of length $L_1$. This can be done with at most $L_2 \log L_1$ comparisons using a method similar to the find-exact algorithm. 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 slotting elements from list 1 and list 2 into 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 explained by the fact that 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. Thus the unbalanced merge performs very quickly in practice, 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.

### 3.5.4 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 size of $B$ is unimportant for the functioning of the algorithm and only makes a difference to the efficiency and memory usage of the algorithm. 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 $L_1$ and $L_2$ are multiples of $B$.

The merge takes place by merging from the two blocked lists of elements into a destination list of blocks. The destination list is initially primed with two empty blocks which comprise a temporary storage area. As each block in the destination list becomes full the algorithm moves on to a new, empty block, choosing the next one in the destination list. As each block in either of the two source lists becomes empty they are added to the destination list.

As the merge proceeds there are always exactly $2B$ free spaces in the three lists. This means that there must always be at least one free block for the algorithm to have on the destination list, whenever a new destination block is required. Thus the elements are merged completely with them ending up in a blocked list format controlled by the destination list.

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

The simplest method for resolving this problem is to go through a re-arrangement phase of the blocks to put them back in the standard form. This is what has been done in the implementation of our parallel sorting algorithm. It would be possible, however, to modify the whole algorithm so that all references

to elements are performed with the elements in this block list format. At this stage the gain from doing this has not warranted the additional complexity, but if the sorting algorithm is to attain its true potential then this would become necessary.

As mentioned earlier, it was assumed that $L_1$ and $L_2$ were both multiples of $B$. In general this is not the case. If $L_1$ is not a multiple of $B$ then this introduces the problem that the initial breakdown of list 2 into blocks of size $B$ will not produce blocks that are aligned on multiples of $B$ relative to the first element in list 1. To overcome this problem we must make a copy of the $L_1 \bmod B$ elements on the tail of list 1 and use this copy as a final source block. Then we must 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 must 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.

Consideration must finally be given to the fact that infinity padding may result in a gap between the elements in list 1 and list 2. This can come about if a node is keeping the larger elements and needs to send more elements than it receives. Handling of this gap turns out to be a trivial extension of the method for handling the fact that $L_1$ may not be a multiple of $B$. We just add an additional offset to the destination blocks equal to the gap size and the problem is solved.

## 3.6 Cleanup

The cleanup phase of the algorithm is similar to the primary merge phase, but it must be guaranteed to complete the sorting process. The method that has been chosen to achieve this is Batcher's merge-exchange algorithm. This algorithm has some useful properties which make it ideal for a cleanup operation.

The pseudo-code for Batcher's merge-exchange algorithm is given in [4]. The algorithm defines a pattern of comparison-exchange operations which will sort 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 generalise 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 first node greater than all elements in the second. This is identical to the operation of the merge-exchange algorithm. A proof that it is possible to make this generalisation while maintaining the correctness of the algorithm is given in [5].

Batcher's merge-exchange algorithm 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, such as the bitonic algorithm [2]. Algorithms that do not have the unidirectional merging property are a poor choice for the cleanup phase as they tend to unsort the data (undoing the work done by the primary merge phase), before sorting it. In practice the cleanup time is of the order of 1 or 2 percent of the total sort time if Batcher's merge-exchange algorithm is used and the merge-exchange operation is implemented efficiently.

# 4 PERFORMANCE

## 4.1 Estimating the Speedup

An important characteristic of any parallel algorithm is how much faster the algorithm performs than an algorithm on a serial machine. Which serial algorithm should be chosen for the comparison ? Should it be the same as the parallel algorithm (running on a single node), or the best known algorithm ?

The first choice gives what which we call the parallel efficiency of the algorithm. This is a measure of the degree to which the algorithm can take advantage of the parallel resources available to it.

The second choice gives the fairest picture of the effectiveness of the algorithm itself. It measures the advantage to be gained by using a parallel approach to the problem. Ideally a parallel algorithm running on $P$ nodes should complete a task $P$ times faster than the best serial algorithm running on a single node of the same machine. It is even conceivable, and sometimes realisable, that caching effects could give a speedup of more than $P$.

A problem with both these choices is apparent when we attempt to time the serial algorithm on a single node. If we wish to consider problems of a size for which a the use of a large parallel machine is worthwhile, then it is likely that a single node cannot complete the task, because of memory or other constraints.

This is the case for our sorting task. The parallel 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 problem we have extrapolated the timing results of the serial algorithm to larger $N$.

The quicksort/insertion-sort algorithm which we have found to perform best on a serial machine is known to have an asymptotic average run time of order $N\log N$. There are, however, contributions to the run time that are of order 1, $N$ and $\log N$. To estimate these contributions we have performed a least squares fit of the form:

$$time(N) = a + b\log N + cN + dN\log N$$

The results of this fit are used in the discussion of the performance of the algorithm to estimate the speedup that has been achieved over the use of a serial algorithm.

## 4.2 Timing Results

Several runs have been made on the AP1000 and CM5 to examine the performance of the sorting algorithm under a variety of conditions. The aim of these runs is to determine the practical performance of the algorithm and to determine what degree of parallel speedup can be achieved on current parallel computers.

The results of the first of these runs are shown in Figure 4. This figure shows the performance of the algorithm on the 128-node AP1000 as $N$ spans a wide range of values, from values which would be easily dealt with on a workstation, to those at the limit of the AP1000s memory capacity (2 Gbyte). The elements are 32-bit random integers. The comparison function has been put inline in the code, allowing the function call cost (which is significant on the Sparc) to be avoided.

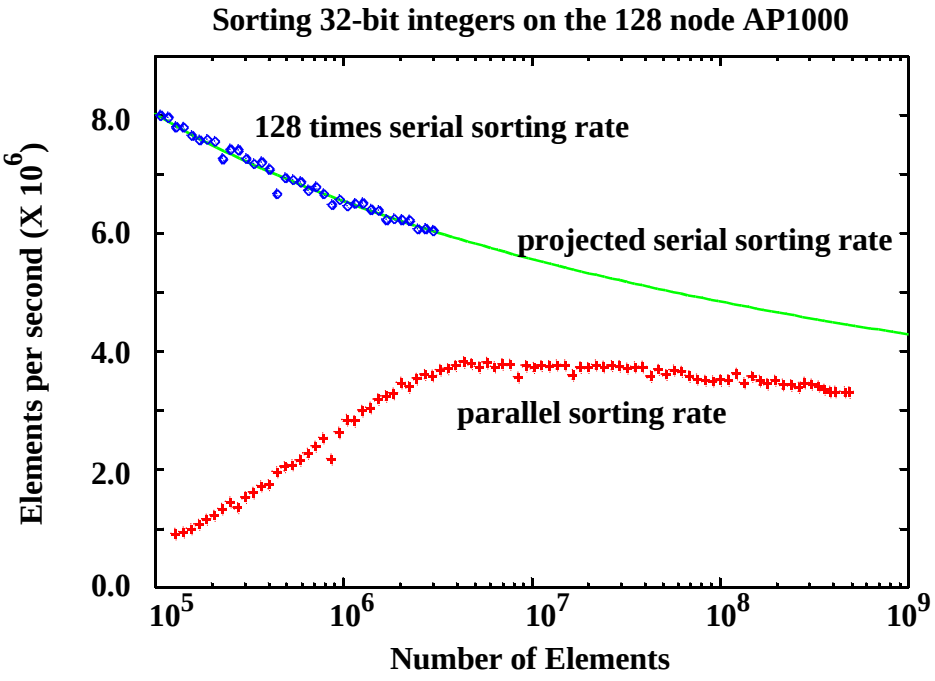


Figure 5. Sorting 32-bit integers on the AP1000

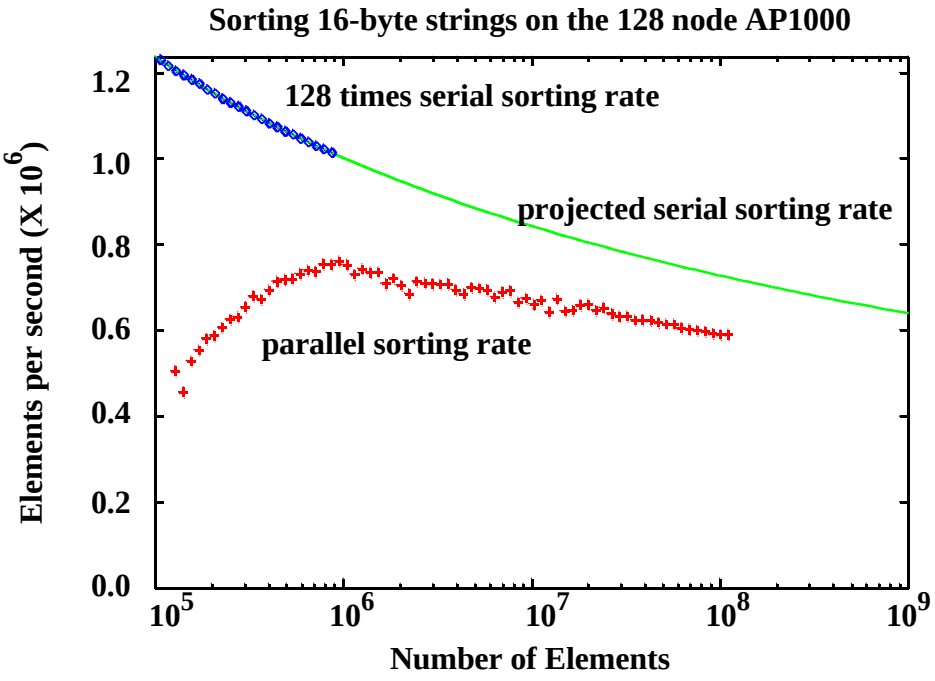


Figure 4. Sorting 16-byte strings on the AP1000

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 the $P$ individual nodes of the parallel computer. This performance is calculated by sorting the elements on a single node and multiplying the resulting elements per second result by $P$. An extrapolation of this result to larger values of $N$ is also shown using the least squares method described in Section 4.1.

The graph shows that the performance of the sorting algorithm increases quickly as the number of elements approaches 4 million, after which a slow falloff occurs which closely follows the profile of the ideal parallel speedup. The rolloff point of 4 million elements corresponds to the number of elements that can be held in the 128Kb 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 75% of the ideal performance for a 128-node machine.

A similar result for sorting of 16-byte random strings is shown in Figure 5. In this case the comparison function is the C library function strcmp(). The roll-off point for best performance in terms of elements per second is observed to be 1 million elements, again corresponding to the cache size on the nodes.

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 the strcmp() function is much higher than an inline integer comparison. The parallel speedup, however, is higher than that achieved for the integer sorting. The algorithm achieves 85% of the (theoretically optimal) $P$ times speedup over the serial algorithm for large $N$.

## 4.3 Scalability

An important aspect of a parallel algorithm is its scalability, which depends on the ability of the algorithm to utilise additional nodes.

Shown in Figure 6 is the result of sorting 100,000 16-byte strings per node on the AP1000 as the number of nodes is varied. The percentages refer to the proportion of the ideal speedup $P$ that is achieved. The number of elements per node is kept constant to ensure that caching factors do not influence the result.

The left-most data point shows the speedup for a single node. This is equal to 1 as the algorithm reduces to our optimised quicksort 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 two in the algorithm. Also clear is that certain values for $P$ are preferred to others. In particular even numbers of nodes perform better than odd numbers. Sums of adjacent powers of two also seem to be preferred, so that when $P$ takes on values of 24, 48 and 96 the efficiency is quite high.

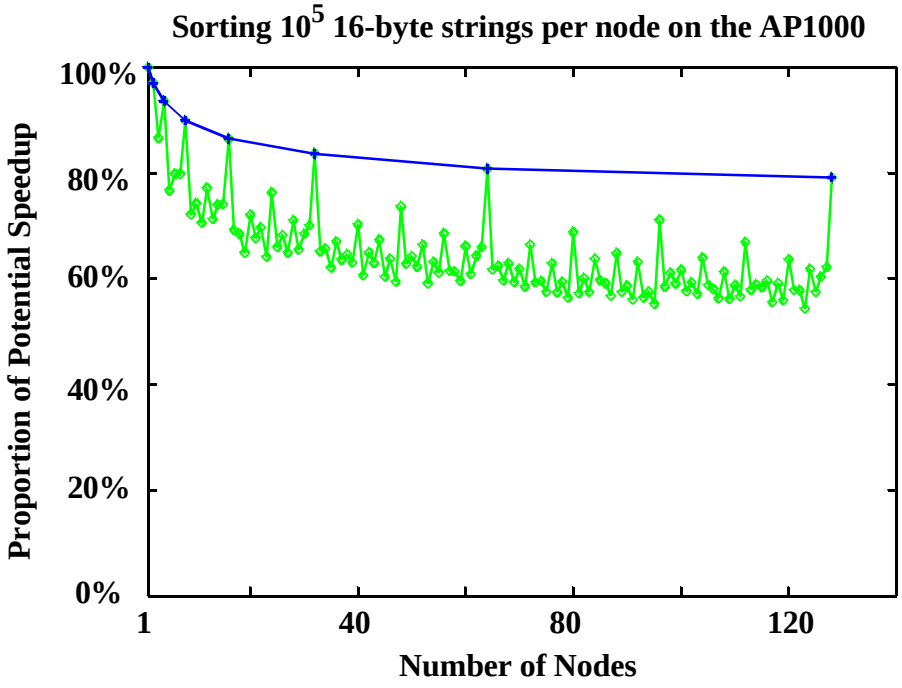


Figure 6. Scalability of sorting on the AP1000

## 4.4 Where Does The Time Go?

In evaluating the performance of a parallel sorting algorithm it is interesting to look at the proportion of the total time spent in each of the phases of the sort. In Figure 7 this is done over a wide range of values of $N$ for sorting 16-byte strings on the AP1000. The three phases that are examined are the initial serial sort, the primary merging and the cleanup phase.

This graph shows that as $N$ increases to a significant proportion of the memory of the machine the dominating time is the initial serial sort of the elements in each cell. This is the case because this phase of the algorithm is $O(N\log N)$ whereas all other phases of the algorithm are $O(N)$ 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 $N$ which are of of interest in practice [6].

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

A second way of splitting the time taken for the parallel sort to complete is by task. In this case we look at what kind of operation each of the nodes is performing, which provided a finer division of the time.

Figure 8 shows the result of this kind of split for the sorting of 16-byte strings on the 128-node AP1000, over a wide range of values of $N$. 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 actually communicating decreases steadily as $N$ increases. From the

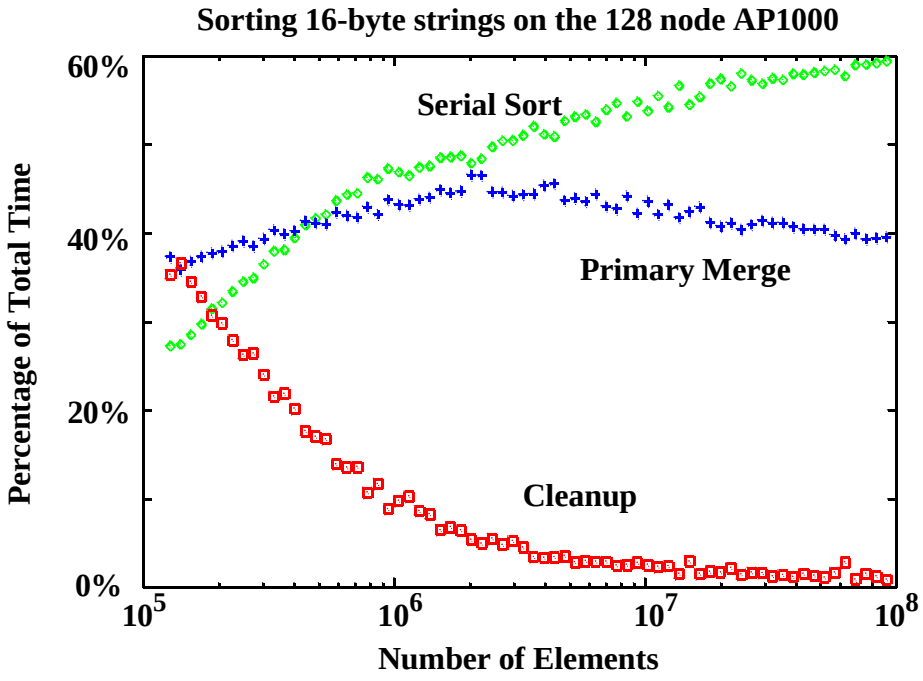


Figure 7. Timing breakdown by phase

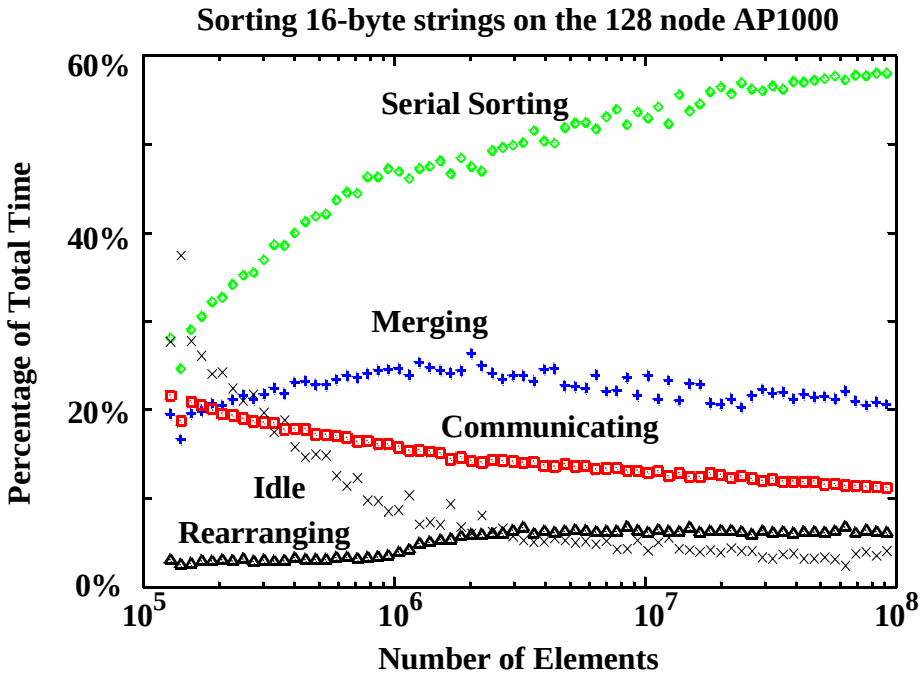


Figure 8. Timing breakdown by task

point of view of the parallel speedup of the algorithm these tasks are wasted time and need to be kept to a minimum.

## 4.5 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 representative of results that are regularly achieved in a real application. It is hoped that, with improvements in the CM5 operating system, the times shown here would be achieved as a matter of course.

In Figure 9 the speed of the various parts of the sorting algorithm are shown for the 32-node AP1000 and CM5. In this example we are sorting 8 million 32-bit integers.

For the communications operations the machines both achieve very similar timing results. For each of the computationally intensive parts of the sorting algorithm, however, the CM5 achieves times which are between 60% and 70% of the times achieved on the AP1000.

An obvious reason for the difference between the two machines is the difference in clock speeds of the individual scalar nodes. There is a ratio of 32 to 25 in favor of the CM5 in the clock speeds. This explains most of the performance difference, but not all. The remainder of the difference is due to the fact that sorting a large number of elements is a very memory-intensive operation.

A major bottleneck in the sorting procedure is the memory bandwidth of the nodes. When operating on blocks which are much larger than the cache size, this results in a high dependency on how often a

| Task | CM5 Time (seconds) | AP1000 time (seconds) |
|---|---|---|
| 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 TIME | 5.48 | 7.62 |

Figure 9. Comparison of CM5 and AP1000 times

cache line must be refilled from memory and how costly the operation is. Thus, the remainder of the difference between the two machines may be explained by the fact that cache lines on the CM5 consist of 32 bytes whereas they consist of 16 bytes on the AP1000. This means a cache line load must occur only half as often on the CM5 as on the AP1000.

The results illustrate how important minor architectural differences can be for the performance of complex algorithms. At the same time the vastly different network structures on 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.

## 4.6 Optimisations

Several optimisation “tricks” have been used to obtain faster performance. It was found that these optimisations played a surprisingly large role in the speed of the algorithm, producing an overall speed difference of about 50%.

The first optimisation was to replace the standard C library routine memcpy() with a much faster version. At first a faster version written in C was used, but this was eventually replaced by a version written in Sparc assembler.

The second optimisation was the tuning of the block size of sends performed when elements are exchanged between nodes. This optimisation is hidden on the CM5 in the CMMD_send_and_receive() routine, but is under the programmer’s control on the AP1000.

The value of the $B$ parameter in the blockwise_merge routine is important. If it is too small then overheads slow down the program, but if it is too large then too many copies must be performed and the system might run out of memory. The value finally chosen was $4\sqrt{L_1+L_2}$.

The method of rearranging blocks in the blockwise_merge routine can have a big influence on the performance as a small change in the algorithm can mean that data is far more likely to be in cache when referenced, thus giving a large performance boost.

A very tight kernel for the merging routine is important for good performance. With loop unrolling and good use of registers this routine can be improved enormously over the obvious simple implementation.

It is quite conceivable that further optimisations to the code are possible and would lead to further improvements in performance.

# 5 CONCLUSIONS

## 5.1 What Has Been Achieved?

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. An implementation of the algorithm on two real 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 be only a small fraction of the machine's capacity.

## 5.2 Availability Of Code

The source code for the algorithm presented in this report is available via anonymous ftp from andosl.anu.edu.au in the directory pub/tridge/sorting/par_sort.

# 6 REFERENCES


[1] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, Orlando, 1985.

[2] G. Fox et al, *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems*, Prentice-Hall, New Jersey, 1988.

[3] H. Ishihata, T. Horie, S. Inano, T. Shimizu and S. Kato, "CAP-II Architecture", *Proceedings of the First Fujitsu-ANU CAP Workshop*, Fujitsu Research Laboratories, Kawasaki, Japan, November 1990.

[4] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (second edition), Addison-Wesley, Menlo Park, 1981, 112-113.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (second edition), Addison-Wesley, Menlo Park, 1981, solution to problem 5.3.4 (38).

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

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

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