THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-97-10

Parallel Integer Sorting

Andrew Tridgell, Richard Brent and Brendan McKay

May 1997

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-97-09 M. Manzur Murshed and Richard P. Brent. *Constant time algorithms for computing the contour of maximal elements on the Reconfigurable Mesh*. May 1997.

TR-CS-97-08 Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. *A mobile TCP socket*. April 1997.

TR-CS-97-07 Richard P. Brent. *A fast vectorised implementation of Wallace's normal random number generator*. April 1997.

TR-CS-97-06 M. Manzur Murshed and Richard P. Brent. *RMSIM: a serial simulator for reconfigurable mesh parallel computers*. April 1997.

TR-CS-97-05 Beat Fischer. *Collocation and filtering — a data smoothing method in surveying engineering and geodesy*. March 1997.

TR-CS-97-04 Stephen Fenwick and Chris Johnson. *HeROD flavoured oct-trees: Scientific computation with a multicomputer persistent object store*. February 1997.

Parallel Integer Sorting

Andrew Tridgell, Richard Brent and Brendan McKay
Department of Computer Science
Australian National University

December 13, 1995

Contents

1 Introduction

This paper presents algorithms and experiments for internal (in core) and external (secondary memory) parallel sorting. It concentrates on algorithms appropriate for medium scale MIMD parallel computers, with all experiments being performed on a 128 processor Fujitsu AP1000.

Data sizes ranging from a few hundred thousand to a few hundred million elements are considered, with all elements being either 64 bit or 128 bit integers.

The internal sorting algorithm is based on earlier work by Andrew Tridgell and Richard Brent[11], while the external sorting algorithm was developed for this paper.

The paper also takes a quick look at serial sorting algorithms, as they play an important part as subroutines in the parallel sorting algorithms.

2 Internal sorting

Internal parallel sorting just means sorting data which fits in the parallel machines primary memory without resorting to secondary memory. It is not difficult to come up with a parallel sorting algorithm that is asymptotically optimal (any local sort then merge algorithm will do) but it can be quite hard to find an algorithm that actually a achieves a reasonable percentage of the optimal P times speedup for reasonable data sizes.

This paper describes a local-sort/merge algorithm which uses a number of algorithmic tricks to achieve quite respectable performance over a wide range of parameters. It was first described in [11].

2.1 Overview of the algorithm

The internal sorting algorithm is basically a simple parallel merge sort. Each processor sorts its own elements using a local sorting algorithm then the processors perform a series of merge-exchange operations to bring the locally sorted lists into globally sorted order.

The novel parts of the algorithm are in the choice of the pattern of merge exchange operations, and the implementation details of the merge-exchange operation itself. Together these lead to a highly efficient algorithm that performs well under a wide range of circumstances.

The pattern of merge exchanges is split into two parts. In the first part, called the primary merge, a highly parallelisable hypercube algorithm is used that "almost sorts" the data. The algorithm produces a ordering of the data which is close to being completely sorted but needs some cleanup operations.

The second part, called the cleanup, consists of merge exchange operations following the pattern of Batcher's merge exchange sort[6]. This algorithm has the property that it completes very quickly for almost sorted data, and thus is ideal as a cleanup operation.

The merge exchange operation itself is also split into a number of parts. In the first part, which we call "find-exact", the two processors exchange individual elements in a bisection search to find which elements need to be transferred between the processors. The processors then do a bulk swap of elements based on the result of the find-exact operation.

Each processor taking part in the merge-exchange then has two lists of elements and needs to perform a local merge-exchange between them. This merge exchange can take advantage of the position of the lists after the bulk swap of elements to produce a fast memory efficient block-wise merge algorithm.

In the following sections each of the stages of the internal sorting algorithm is described. A more detailed and general account of the algorithm can also be found in [11].

2.2 Distribution requirements

The algorithm described below will assume that the elements start out distributed across the processors. It will also assume a particular packing of the elements on the processors. The packing that is required is described as follows

$$N_{p+1} \leq N_p$$

$$\text{If } N_{p+1} > 0 \text{ then } N_p = N_{p-1}$$

where N_p is the number of elements on processor p and p is restricted to suitable ranges.

This packing is a result of "infinity padding" where virtual infinity elements are added to the data to bring the total number of elements up to a multiple of the number of processors P. These virtual infinity elements must be initially placed in the highest cell numbers to guarantee that they will not have to move during the sorting process.

An algorithm for efficiently obtaining the required initial distribution is described in [11].

2.3 Local sorting

The choice of algorithm for sorting the elements within each cell is perhaps the simplest part of the process. As the results will show, however, it is critical to the overall performance of the algorithm.

A simple demonstration of the importance of the serial sorting phase can be made by calculating its expected contribution to the total time for the parallel sorting algorithm under some simple assumptions.

If we assume that the serial sorting algorithm takes time $N \log N$ then we would expect an ideal parallel sorting algorithm to take $\frac{N \log N}{P}$. The local sorting portion of the algorithm (executed in parallel) would take $\frac{N}{P} \log \frac{N}{P}$. This means we would expect the serial sorting phase of the algorithm to take $1 - \frac{\log P}{\log N}$ of the total time.

If we assume we have 2^7 processors and 2^{30} elements then we would expect the serial sorting phase to take more than 75% of the total sorting time. Obviously the choice of serial sorting algorithm is critical.

2.3.1 Quicksort

A very commonly used serial sorting algorithm is quicksort. Quicksort is a comparison based in-place serial sorting algorithm based around partitioning. It is often included (in a slightly modified form) in a standard C library function qsort(). The problem is that the C library implementations are typically very slow.

The main problem is that they call a user defined function for each comparison. A huge speed improvement can often be had by inlining the comparison function. Further improvements can be made by tuning the insertion sort threshold, which is the point at which the algorithm switches to a fast insertion sort for small numbers of elements.

For this paper the GNU quicksort implementation was investigated, using inlined comparisons. It was found to be much faster than the standard library routines.

2.3.2 Radix sort

When sorting integers or integer-like quantities a considerable advantage can be gained from the bitwise ordering of the elements. The best known algorithm to take advantage of this is radix sort. In its simplest form a radix sort is a bucket sort which first partitions using the least significant bits then progressively partitions for the more significant bits of the data.

Typically 8 bits are considered at one time for convenience of programming. This means that for 128 bit integers the algorithm requires 16 passes through the data, with each pass doing essentially random accesses. This leads to very poor performance. Radix sorts perform very well for small integers (up to around 32 bits or so) but degrade rapidly beyond that.

Radix sorts also typically require temporary storage of approximately equal size to the input data. As will become plain later, this is unacceptable for parallel sorting in many cases, especially for external parallel sorting.

2.3.3 American flag sort

Another class of radix sorting algorithms are the forward radix sorts. These examine the most significant bits first and do a successive partitioning moving towards the less significant bits. This has the big advantage that the sort can stop when enough bits have been examined to sort the data (for N elements this will typically be around $logN$ bits for random data). This makes it much more attractive for sorting 128 bit elements.

The problem with forward radix sorts is that they commonly produce lots of empty buckets in the later passes of the sort which leads to large overheads in bucket management.

The American flag sort[12] is a fine tuned forward radix sort. It uses algorithmic and programming tricks to largely avoid the problems of empty buckets, and also manages to produce a in-place radix sort using a sophisticated tail-chasing algorithm. It performs very well over a very wide range of parameters.

The standard American flag sort is designed for NULL terminated, variable length strings, and uses an array of pointers to the strings as the "elements". For this paper the algorithm was modified to sort fixed length integers without the use of an array of pointers. The changes required to do this were quite small.

The American flag sort was chosen as the principle serial sorting algorithm for the results later in this paper.

2.4 Merge exchange operation

The aim of the merge-exchange operation is to exchange elements between two processors so that we end up with one processor containing elements which are all smaller than all the elements in the other processor, while maintaining the order of the elements in the processors. In our implementation of parallel sorting we always require the processor with the smaller processor number to receive the smaller elements. [1]

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. [2]

Suppose that a merge operation is needed between two processors, 1 and 2, which initially contain N_1 and N_2 elements respectively. We assume that the smaller elements are required in processor 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 1

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.

2.4.1 The Find-Exact Algorithm

When a processor takes part in a merge-exchange with another processor, it will need to be able to access the other processors elements as well as its own. The

[1] This would not be possible if we used Batcher's bitonic algorithm instead of his merge-exchange algorithm for the cleanup phase. The bitonic algorithm is not a unidirectional algorithm, it requires elements to be moved away from their final destination at some stages.

[2] The minimisation of the memory overhead becomes important when the internal parallel sorting algorithm is later used as a subroutine for the external algorithm, where memory is at a premium

```
procedure merge(list dest, 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 dest
  else
    put top_of_source2 into dest
  endif
endwhile
while (source1 not empty)
  put top_of_source1 into dest
endwhile
while (source2 not empty)
  put top_of_source2 into dest
endwhile
end
```

Figure 1: Pseudo-code for a simple merge

simplest method for doing this is for each processor to receive a copy of all of the other processors elements before the merge begins.

A much better approach is to first determine exactly how many elements from each processor 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 processor to determine exactly how many elements are required from another processor in order to produce the correct number of elements in a merged list.

Say we have two processors, called processor 1 and processor 2, which each have N elements.[3] We want to find which of these $2N$ elements will need to end up in the first cell and which in the second.

If we label the two lists of elements $E_{1,i}$ and $E_{2,i}$ then when a comparison is made between element $E_{1,A-1}$ and $E_{2,N-A}$ then the result of the comparison determines whether processor 1 will require more or less than A of its own elements in the merge. If $E_{1,A-1}$ is greater than $E_{2,N-A}$ then the maximum number of elements that could be required to be kept by processor 1 is $A-1$, otherwise the minimum number of elements that could be required to be kept by processor 1 is A.

The proof that this is correct relies on counting the number of elements that

[3] A generalisation to the case where the number of elements in each cell is not the same is given in [11]

could be less than $E_{1,A-1}$. If $E_{1,A-1}$ is greater than $E_{2,N-A}$ then we know that there are at least $N - A + 1$ elements in processor 2 that are less than $E_{1,A-1}$. If these are combined with the $A - 1$ elements in processor 1 that are less than $E_{1,A-1}$, then we have at least N elements less than $E_{1,A-1}$. This means that the number of elements that must be kept by processor 1 must be at most $A-1$.

A similar argument can be used to show that if $E_{1,A-1} \leq E_{2,N-A}$ then the number of elements to be kept by processor 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 $\log N$ steps by successively halving the range of possible values for the number of elements required to be kept by processor 1.

Once this result is determined it is a simple matter to derive from this the number of elements that must be sent from processor 1 to processor 2 and from processor 2 to processor 1.

On a machine with a high message latency, this algorithm could be costly, as a relatively large number (ie. up to $logN$) 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 processors 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 target machine.

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 processor 1 must retain L elements and must transfer $N - L$ elements from processor 2.

2.4.2 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 processor 1 is much larger than the number of elements to be transferred from processor 2.

In this case the task which processor 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 is much greater than $N - L$. This may occur if the data is almost sorted, for example, near the end of the cleanup phase.[4] We proceed as follows.

First we determine, for each of the $N-L$ elements that have been transferred from 1, where it belongs in the list of length L. This can be done with at most $(N - L)\log L$ comparisons using a method similar to the find-exact algorithm.

[4]Experiments have shown that this is a very common occurrence, and the implementation of this as a special case improves the overall performance of the parallel sorting algorithm significantly

As $N - L$ is small, this number of comparisons is small, and the results take only $O(N - L)$ 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 N element copies, but in practice it often takes only about $2(N-L)$ copies. This is explained by the fact that when only a small number of elements are transferred between processors there is often only a small overlap between the ranges of elements in the two processors, 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.

2.4.3 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 $O(\sqrt{N})$, which is small relative to the memory available on each processor. To simplify the exposition we also assume, for the time being, that L and $N - L$ 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

```
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
```

Figure 2: Pseudo-code for primary merge

if the sorting algorithm is to attain its true potential then this would become necessary.

As mentioned earlier, it was assumed that L and $N-L$ were both multiples of B. In general this is not the case. If L 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 $L \bmod B$ 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 processor 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 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.

2.5 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 consists of merge-exchange operations along the edges of a hyper-cube. The pseudo-code for the algorithm is given in Figure 2. When the algorithm is called the *base* is

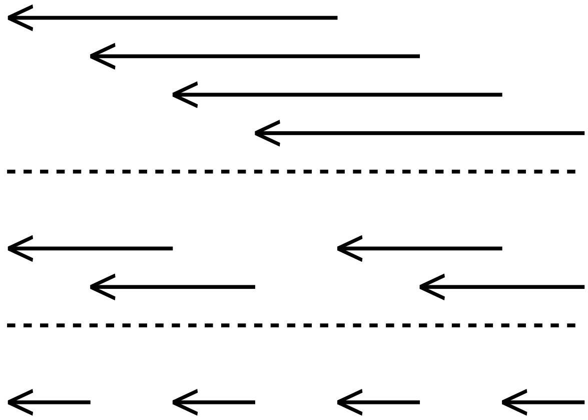

Figure 3: The parallelisation of the primary merge for $P = 8$

initially set to the index of the smallest node in the system and *num* is set to the number of nodes, P.

Note that the algorithm is only a true hyper-cube algorithm for when P is a power of 2. In other cases the algorithm can be thought of as a incomplete hyper-cube, and will lead to less parallel efficiency than can be achieved with a power of 2.

This algorithm completes in $\log P$ steps per node, with each step consisting of a merge-exchange operation between two processors as described above. 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.

Figure3 shows the parallelisation of the primary merge for $P = 8$. The figure is layed out with time on the vertical axis. On the horizontal axis are evenly spaced points for each processor. An arrow going from processor 4 to processor 0 means that a merge exchange takes place between those two processors with the smaller elements ending up in processor 0.

The dashed line separates stages of the sort which can be completed completely in parallel. Thus the primary merge for $P = 8$ takes 3 parallel steps to complete. In fact, this algorithm will always take $\lceil \log_2 P \rceil$ parallel steps.

2.6 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 [6]. 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 [7].

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 [4]. 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.

Figure4 shows the parallelisation of the cleanup operation for $P = 8$. It clearly has a much lower parallel efficiency than the primary merge phase for the same P, taking 6 parallel steps instead of 3, and leaving many processors idle at some steps. Each step will, however, be much cheaper than the steps of the primary merge as most of the work will have already been done.

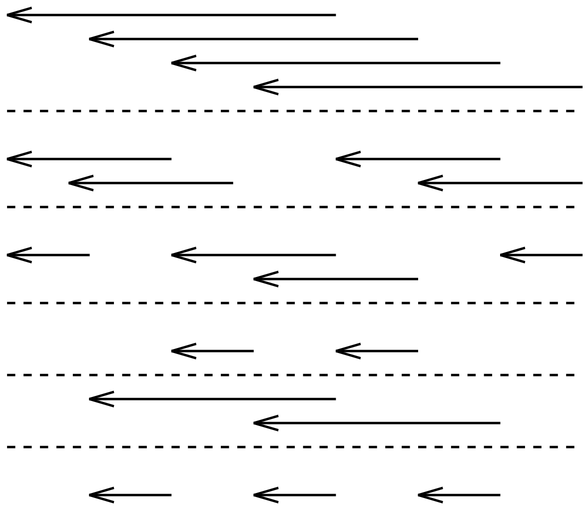

Figure 4: The parallelisation of the cleanup for $P = 8$

3 External sorting

External sorting involves sorting more data than can fit in the combined memory of all the processors on the machine. The sorting algorithm needs to make extensive use of secondary memory (disk in our case). For this paper we will characterise the external sorting using a parameter k where $k = \lceil \frac{N}{M} \rceil$. This means that k equal to 1 implies internal sorting (as the data can fit in memory) and any value of k greater than 1 implies external sorting.

3.1 Lower Limits on I/O

Lower limits on the computational requirements for parallel external sorting are the same as those for internal sorting. It is, however, instructive to look at the lower limits on I/O operations required for external sorting as quite clearly the number of I/O operations required will impact significantly on the performance of the algorithm.

If we assume that we have a I/O device which supports read/write operations in a similar fashion to todays file systems, then we can argue for at least 2 read operations on each element and two write operations on each element for large values of k. The argument to support this assertion is not a strict lower bound, but is based on practical arguments. The argument also relies on not having an insert operation on files.

The first read operation would be needed to completely characterise the distribution of the elements. Say we were to read only $N-1$ elements and attempt to place a single element into its correct position in the final sorted ordering, then the unknown value of the Nth element would prevent this placement with any certainty. This means we cannot place any elements into sorted order (with any certainty) until we have read all elements.

The first write operation is required to store the characterisation of the distribution of the elements. As argued above we need this characterisation of the distribution before we can write any elements in their final position. For uncompressible data we will require storage of the order of N to hold a precise characterisation of the input distribution. This must be placed in secondary storage as $N > M$. [5]

The second read operation is required to load elements before the final write and the second write operation is required to write the elements into their final positions.

The algorithm presented below comes close to this limit on average for practical values of k, M and N.

3.2 Overview of the algorithm

The external sorting algorithm works by mapping the file onto a 2 dimensional $k \times k$ grid in a snake like fashion. The columns and rows of the grid are then alternatively sorted using the internal parallel sorting algorithm described above. It can be demonstrated that the upper limit on the number of column and row sorts required is $\lceil \log k \rceil + 1$[13].

This basic algorithm is augmented with a number of optimisations which greatly improve the efficiency of the algorithm in the average case. In particular the grid squares are further divided into *slices* which contain enough elements to fill one processor. The top and bottom element in each slice is kept in memory allowing the efficient detection of slices for which all elements are in their correct final position. In practice this means that most slices are completely sorted after just one column and row sort, and they can be ignored in the following operations.

The algorithm also uses a dynamic allocation of processors to slices, allowing for an even distribution of work to processors as the number of unfinished slices decreases.

The proof that the algorithm does indeed sort and the upper limit on the number of column and row sorts comes from the shear-sort algorithm[13].

3.3 Partitioning

The external sorting algorithm starts off by partitioning the file to be sorted into a two dimensional grid of size $k \times k$. Each grid square is further subdivided

[5]Strictly we would only require $\log N!$ bits of storage for this distribution, but in practice this limits us to approximately N as computing permutations would be impractical for large N

0 (0,0)	1 (0,0)	2 (0,1)	3 (0,1)	4 (0,2)	5 (0,2)
10 (1,0)	11 (1,0)	8 (1,1)	9 (1,1)	6 (1,2)	7 (1,2)
12 (2,0)	13 (2,0)	14 (2,1)	15 (2,1)	16 (2,2)	17 (2,2)

Figure 5: Partitioning with $k = 3$ and $P = 6$

into a number of slices, so that the number of elements in a slice does not exceed the number that can be held in one processors memory.

A example of this partitioning is shown in Figure 5 for $k = 3$ and $P = 6$. Each slice in the figure is labeled with two sets of coordinates. One is its slice number and the other is its (row,column) coordinates. The slice number uniquely defines the slice whereas the (row,column) coordinates are shared between all slices in a grid square.

The snake like ordering of the slice numbers on the grid squares is essential to the sorting process. Note that in the diagram the slices are labeled in snake-like fashion between the grid squares but are labeled left-right within a grid square. The ordering within a grid square is not essential to the sorting process but is for convenience and code simplicity.

One difficulty in the partitioning is achieving some particular restrictions for on the numbers of slices, elements, processors and grid squares. The restrictions are:

- The number of slices in a row or column must be less than or equal to the number of processors P
- The number of elements in a slice must be less than the amount of available memory in a processor M [6]

[6] Here and elsewhere in this paper we measure memory in units of elements, so one memory unit is assumed to hold one element

These restrictions arise from the need to load all of a slice into a single processor and the need to be able to load a complete row or column into the total internal memory. To achieve these requirements a iterative procedure is used which first estimates k and assigns slices then increases k if the restrictions are not met.

The file itself is mapped onto the slices in slice order. This means that each slice represents a particular static region of the file. Whenever a slice is loaded or saved the offset into the file is calculated from the slice number and the block of elements at that offset is loaded or saved to a single processors memory.

Note that the mapping of slices to file position is static, but the mapping of slices to processors is dynamic. This is discussed further in section 3.6.

3.4 Column and row sorting

The heart of the external sorting algorithm is the alternate column and row sorting. To sort a row or column all slices with the required row or column number are loaded into memory, with one slice per processor, then the internal sorting algorithm described previously is used to sort the row or column into slice order.

The algorithm cycles through all columns on the grid, sorting each column in turn, then cycles through each row in a similar fashion. This continues until the sort is complete. The detection of completion is discussed below.

The only change to the internal sorting algorithm previously presented is to eliminate the local sorting phase except the first time a slice is used. Once a slice has taken part in one row or column sort its elements will be internally sorted and thus will not need to be sorted in later passes.

3.5 Completion

An important part of the algorithm is the detection of completion of the sort. Although it would be possible to use the known properties of shear-sort to guarantee completion by just running the algorithm for $\lceil \log k + 1 \rceil$ passes, it is possible to improve the average case enormously by looking for early completion.

Early completion detection is performed on a slice by slice basis, rather than on the whole grid. Completion of the overall sorting algorithm is then defined to occur when all slices have completed.

The completion of a slice is detected by first augmenting each slice number with a copy of the highest and lowest element in the slice. The last processor to write the slice holds these elements.

At the end of each set of column or row sorts these sentinel elements are then gathered in one processor using a simple tree based gathering algorithm. This processor then check to see if the following two conditions are true to determine if the slice has completed

- the smallest element in the slice is larger than or equal to the largest element in all preceding slices.
- the largest element in each slice is smaller than or equal to the smallest element in each of the following slices

If these two conditions are true then all elements in the slice must be in their correct final positions. In this case the elements need never be read or written again and the slice is marked as finished. When all slices are marked as finished the sort is complete. [7]

3.6 Processor allocation

As slices are marked complete the number of slices in a row or column will drop below the number of processors P. This means that if a strictly sequential sorting of the rows or columns was made then processors would be left idle in rows or columns which have less than P slices remaining.

To take advantage of these additional processors the allocation of slices to processors is made dynamically. Thus while one row is being sorted additional processors can begin the task of sorting the next row. Even if not enough additional processors are available to sort the next row a significant saving can be made because the data for the slices in the next row can be loaded, overlapping I/O with computation.

This dynamical allocation of processors can be done without additional communication costs because the result of the slice completion code is made available to all processors through a broadcast from one cell. This means that all processors know what slices are not completed and can separately calculate the allocation of processors to slices.

3.7 Large k

The algorithm described above has a limit of $k = P$. Above this point the allocation of slices to processors becomes much trickier. The simplest way of addressing this problem is to allocate multiple grid squares to a processor. A more complex alternative would be to sort recursively, so that rows and columns are further subdivided into sections that can be handled by the available processors and memory.

A further problem that comes with very large values of k is that the worst case of $\lceil \log k + 1 \rceil$ passes becomes a more significant burden. To overcome this problem alternative grid sorting algorithms to shear-sort may be used which work well for much larger values of k. For example, reverse-sort[14] has a worst

[7] The determination of the two conditions can be accomplished in linear time by first calculating the cumulative largest and cumulative smallest elements for each slice.

case of $\lceil \log \log k \rceil$ for large k. For smaller k, however, it has no advantage over shear-sort.

It is difficult to properly investigate these large k parameter ranges as available hardware (disk sizes) limits the available range of k to much less than P for current super-computer configurations. Values of k larger than P were not attempted in this paper.

3.8 Other partitionings

It may not be obvious from the above why a 2 dimensional partitioning was chosen. Other partitionings were considered as they lacked essential requirements or were slower.

In particular many of the obvious one-dimensional sorting algorithms would require that either far fewer or far more than $1/k$ of the elements be in memory at any one time. To have more than $1/k$ in memory would be impossible, and to have fewer would be inefficient as the obviously valuable resource of memory would be wasted.

Higher dimensional partitionings were also rejected after consideration. One that particularly appealed was a k dimensional hyper-cube, but it was simple to prove that its average case would be equal to the worst case of the 2 dimensional partitioning. It could not take advantage of the shortcuts which are so easy to produce in the 2 dimensional case.

This is not to say that there isn't a better partitioning than the 2 dimensional one proposed here. There may well be, but we haven't come up with one.

It should also be noted that if you were to choose k such that $k = \sqrt{N}$ and also assume that the serial sorting algorithm scales as $N \log N$ then one pass of the algorithm would take $N \log N$ time. The problem, however, is that $k = \sqrt{N}$ would provide a much too fine grained algorithm, which would suffer badly from overheads and latencies. The use of $k = N/M$ is the best that can be done in terms of the granularity of the algorithm.

4 Performance

The "proof of the pudding" is the speed. It is quite easy to create a parallel sorting algorithm that is asymptotically optimal, but it is much harder to create one which is fast over realistic data sizes.

In this section the performance of the above algorithms is examined for a range of data sizes and machine configurations.

4.1 The test environment

The principle test environment was a 128 processor Fujitsu AP1000 [5]. This machine contains 128 Sparc scalar processors connected on an 8 by 16 torus. Inter-processor communication is performed by hardware, using wormhole routing. Each processor has 16Mb of local memory and all are connected to a host workstation via a relatively slow connection.

A 0.5GB SCSI disk is attached to 32 of the processors. Each disk is capable of a maximum transfer rate of 2MB/sec under ideal conditions. More typical throughput for large transfers is 1.5MB/sec when the filesystem overheads are taken into account. The filesystem used on the AP1000 was the HiDIOS filesystem[15]. This gives a single filesystem view across all the processors rather than each processor seeing only its own files.

The AP1000 was programmed in C, using the MPI libraries for communications. The code should be portable to other MPI implementations. The GNU C compiler version 2.6.3 was used.

The serial sorting routines were also tested on a Intel Pentium 90 PC running Linux.

4.2 Serial sorting

As was pointed out earlier the algorithm used for the serial sorting components of the parallel sorting algorithm can have a very large impact on the overall performance. It is important to choose a serial sorting algorithm which is well suited to the architecture of the target machine and to the distribution properties of the data.

Figure 6 shows the speed in elements per second of sorting 64-bit integers on the Intel CPU as the number of elements to be sorted is varied. Two sorting algorithms are shown, the top one is the American flag sort, modified for inplace integer sorting. The bottom one is the GNU quicksort routine with inlined comparison function.

The graph clearly shows the speed advantage of the forward radix sorting used by the American flag sort. A speed difference of up to a factor or 5 is shown. The American flag sort also varies in speed much more across the range tested. It is quite feasible that the quicksort will come close to the American

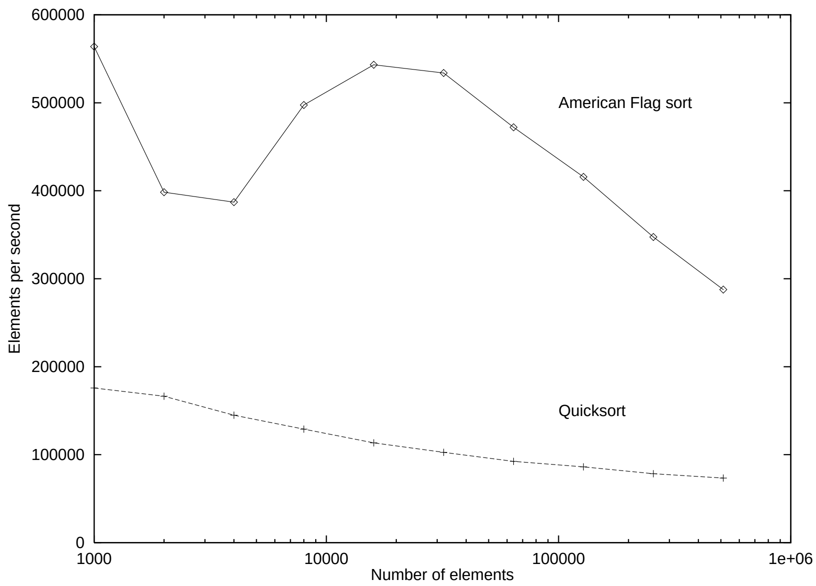

Figure 6: Serial sorting 64 bit integers on a Pentium 90

flag sort for data sizes beyond what was tested here.[8]

Figure 7 shows the same graph for a AP1000 processor. In this case the speed difference between the American flag sort and the quicksort is not nearly as large. This reflects the different processor architecture of the Sparc1+ and clearly demonstrates the importance of selecting a sorting algorithm that is appropriate for the target architecture.[9]

Figure 8 shows the speed of sorting 128 bit elements on the Intel CPU. It is interesting to note that the speed has dropped by a factor much less than 2. If the sorting algorithm was "ideal" then a factor of 2 decrease would be expected as data movement costs would dominate. This suggests that there is room for further optimisation of these routines to reduce overheads.

Figure 9 shows the same graph for a AP1000 processor. Again the speed has dropped by a factor much less than 2, and the difference between the two algorithms is not nearly as large as it is for the Intel CPU.

For the remainder of the testing in this paper the American flag sort is used for serial sorting components.

[8]The amount of memory available on the machine severely limits the range of data sizes that can be tested

[9]It should be noted that the Sparc1+ CPUs in the AP1000 are quite old technology. More recent machines in the same line use SuperSparc CPUs which have much better performance

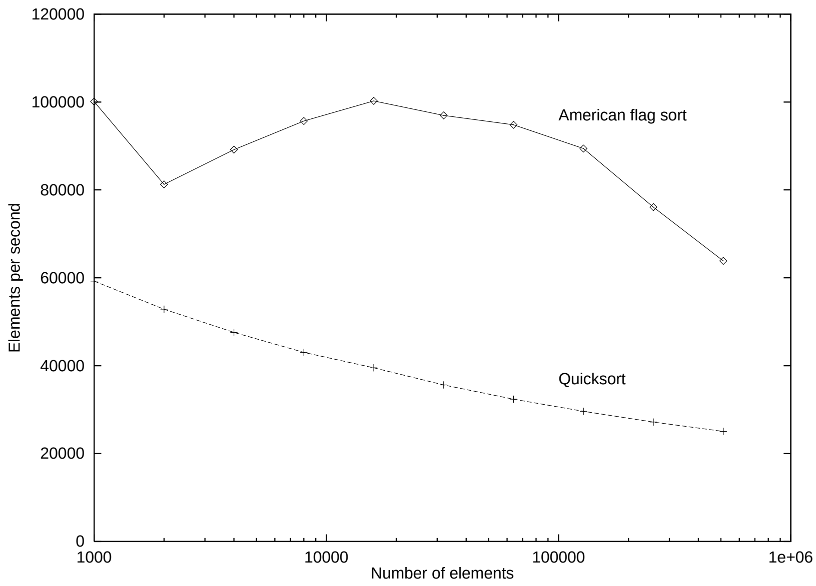

Figure 7: Serial sorting 64 bit integers on a AP1000 processor

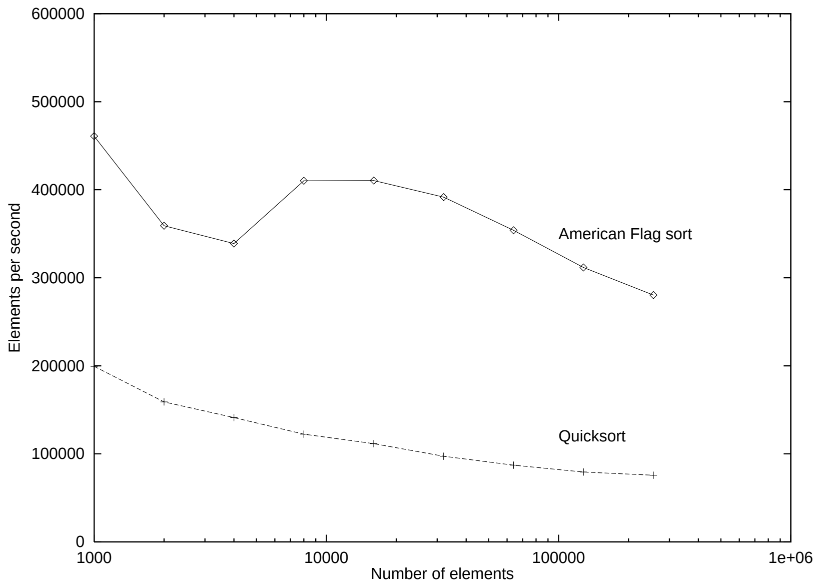

Figure 8: Serial sorting 128 bit integers on a Pentium 90

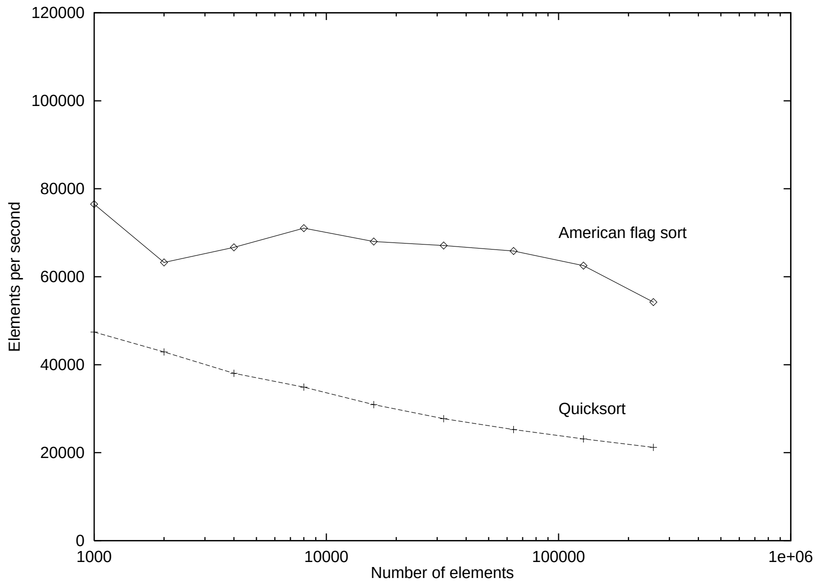

Figure 9: Serial sorting 128 bit integers on a AP1000 processor

4.3 Internal parallel sorting

Figure 10 shows the speed of the internal sorting algorithm as the data size is varied. The two graphs show the speed for 64 bit and 128 bit integers.

It is difficult to extract from these results a accurate speedup (parallel efficiency) figure. The problem is that the serial sorting algorithm results shown in Figure 7 and Figure 9 have no obviously good extrapolation. If, however, we use a very simple extrapolation then we might estimate the serial sorting rate for large data sizes would be 50000 elements per second for 64 bit integers and 40000 elements per second for 128 bit integers.

Given these estimates we get an approximate speedup of 50 and 38 respectively. This is much less than the ideal 128 times speedup for this machine. The reason for this less than ideal speedup comes from the extremely optimised nature of the american flag sort. In a previous paper[11] a much higher speedup was found for the same algorithm when using the GNU quicksort algorithm for serial sorting.

A breakdown by time of the internal sorting algorithm is shown in Figure 11 for 64 bit integers and Figure 12 for 128 bit integers. These graphs show that the time is dominated by the primary merge phase of the algorithm.

The cleanup phase becomes insignificant as the data size rises, showing that the primary merge is doing its job of almost sorting the data.

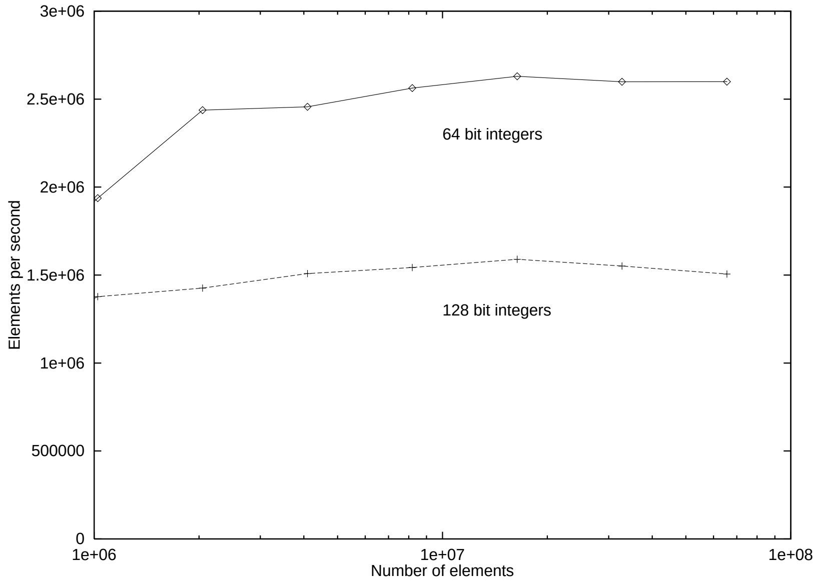

Figure 10: Internal sorting 64 and 128 bit integers

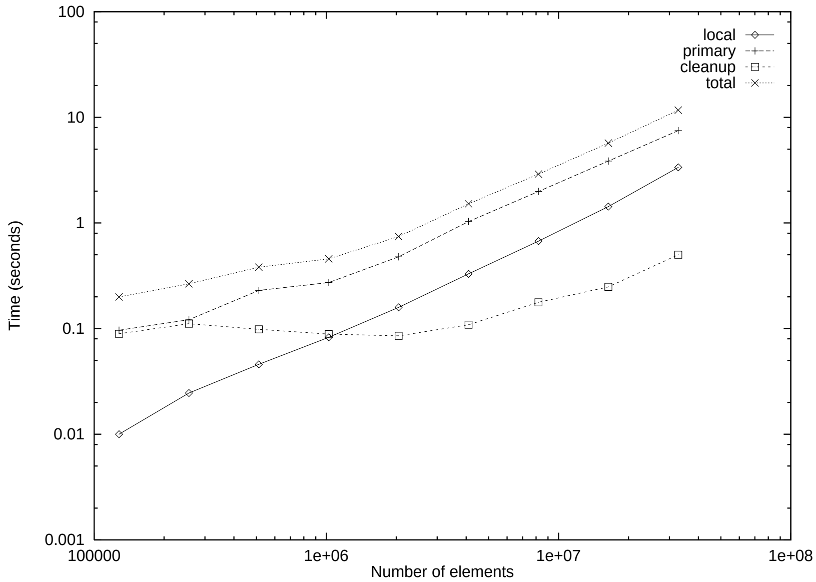

Figure 11: Internal sorting breakdown for 64 bit integers

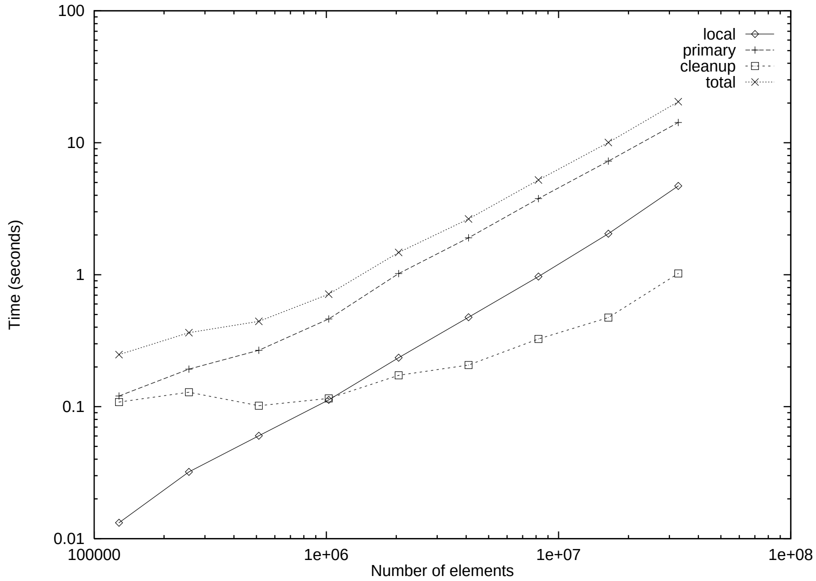

Figure 12: Internal sorting breakdown for 128 bit integers

4.4 External sorting

The problem with measuring the speed of external sorting is finding enough disk space. The AP1000 has 2GB of internal memory, which means you need to sort considerably more than 2GB of data to reach reasonable values of k. Unfortunately only 10GB (of a possible 16GB) is allocated to the HiDIOS filesystem, so a k above 5 is not really possible. In practice a k of above 2 is difficult to achieve as the disks are invariably quite full.

To overcome this problem the following experiments limit the amount of ram available to the program to much less than the full 2GB. This allows a much wider range of external sorting parameters to be explored.

Figure 13 shows the speed in elements per second of a external sort where the available memory per processor has been fixed at 1MB (excluding operating system overheads). This means that k increases with the number of elements to be sorted. The left most point on the graph is in fact an internal sort as there is sufficient memory available for $k = 1$, meaning the data can fit in internal memory. This graph includes the I/O overhead of loading and saving the data, which is why the internal point is considerably slower than the previous internal results.

The alternate high-low structure of the graph is due to the number of elements not being close to a multiple of k^2 for the lower results. This means that many grid squares on the $k \times k$ grid are a long way from being completely filled,

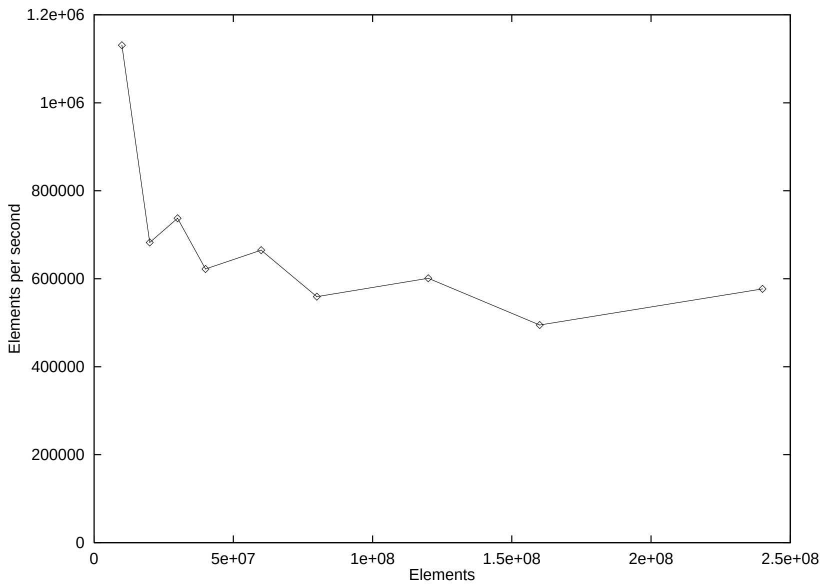

Figure 13: External sorting of 64 bit integers with fixed memory

which leads to load imbalance.

Figure 14 shows the speed of sorting a fixed set of 50 million 64 bit integers while the amount of available memory on each processor is changed. The graph shows clearly the degradation of the speed of the sort for small memory sizes [10]

This degradation arises from several causes. One cause is the inefficiency of disk reads/writes for small I/O sizes. A more significant cause is the overheads for each of the internal parallel sorts, with the number of internal sorts required increasing as k (which increases as the inverse of the available memory).

Figure 15 gives a breakdown of the costs associated with the previous figure. It demonstrates that the overheads of the sorting process increase greatly for small memory sizes, while the serial sorting component stays roughly static.

4.4.1 IO Efficiency

It is interesting to see how close the algorithm comes to the practical limit of 2 reads and writes per element for external sorting. The worst case is $2(\log k + 1)$ reads and writes per element but the early completion detection in the algorithm allows much smaller values to be reached.

Figure 16 shows the number of reads and writes required per element for

[10] It should be noted that the smaller values on the graph do represent extremely small amounts of available memory. 50kB is about 0.3% of the memory normally available to each processor!

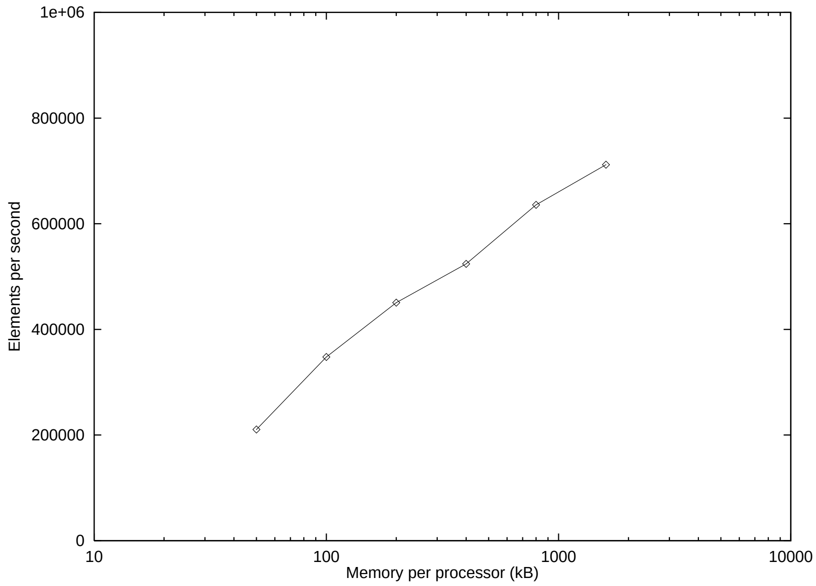

Figure 14: External sorting of 50 million 64 bit integers with varying amounts of available memory

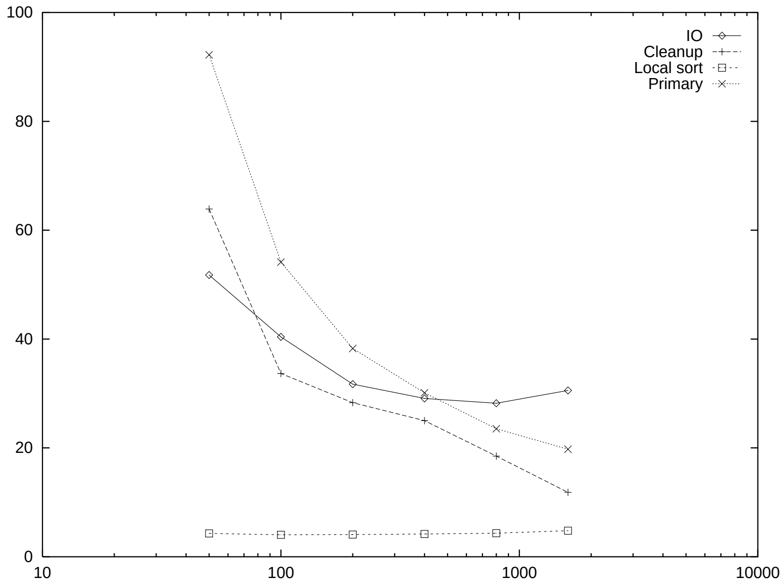

Figure 15: Breakdown of cost of external sorting as amount of available memory varies

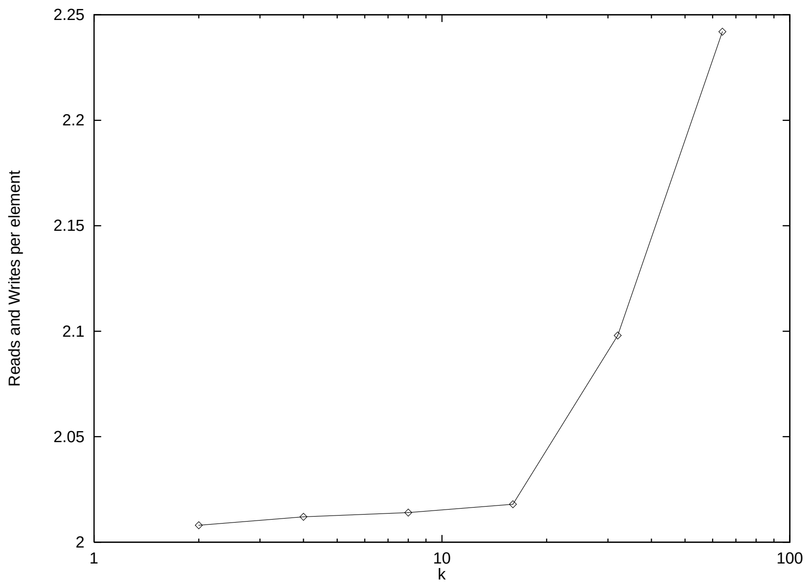

Figure 16: Number of reads and writes per element for a range of k values

random 64 bit integers. Note the narrow y scale. The graph shows that the algorithm achieves close to the ideal 2 reads and writes per element over a wide range of k, with a slight increase for larger k. This demonstrates that the early completion detection part of the algorithm is functioning well.

4.4.2 Worst case

An important property of a sorting algorithm is the worst case performance. Ideally the worst case is not much worse than the average case, but this is hard to achieve.

For the internal sorting the worst case is achieved when the data has a distribution which gives the worst case for the American flag sort. This means the distribution must be such that all bytes are required to sort the data, so that the early completion detection cannot be used. The worst histogram for the American flag sort is one which covers a wide range of byte values, as it uses an optimisation which relies on the byte values covering less than the full range. This means the worst case is when each byte of the data is randomly 0 or 255 with equal probability.

A experiment with 10 million 64 bit elements and a k of 1 (internal sorting) gave a slowdown for the worst case compared to random data of about a factor of 2.

The worst case for the external sorting algorithm is when the data starts out sorted by the column number that is assigned after the partitioning stage. This will mean that the initial column sort will not achieve anything. The worst case of $\log k + 1$ passes can then be achieved. Additionally the data distribution required for the worst case of the internal sorting algorithm should be used.

A experiment with 10 million 64 bit elements and a k of 7 gave a slowdown for the worst case compared to random data of about a factor of 4.

5 Conclusions

This paper has presented algorithms for internal and external parallel integer sorting. The algorithms have a high parallel efficiency and are well suited to massively parallel machines such as the AP1000.

The in-core sorting experiments demonstrated a sorting speed of over 2 million elements per second for sorting more than 1 million 64 bit integers on a 128 processor AP1000. For 128 bit integers 1.5 million elements per second was achieved for more than 50 million elements.

The secondary memory sorting experiments demonstrated a speed of 7 hundred thousand elements per second for 64 bit integers with a limited amount of memory available per processor. The strong dependence on the amount of available memory was also demonstrated.

Parameters	Millions of elements per second
in core, 64 bit, 1M elements	2
in core, 128 bit, 50M elements	1.5
external, 64 bit, 50M elements	0.7

There is probably still considerable room for improvement in the detailed implementation of the algorithms, in particular the reduction of overheads associated with each internal sort in the external sorting algorithm. A much finer grained profiling will be needed to identify the bottlenecks.

6 Source code

The full source code (using C and MPI) is available from the authors. It should be easily portable to other systems which support the MPI interface. It consists of a bit under 3000 lines of code.

The main program in main.c is merely a convenient test harness. It can be used to generate and sort data under a variety of conditions, controlled by command line options. For a real application it is likely that the code in par_sort.c would be used as the basis for a library routine.

6.1 Using the test program

The test program can generate and sort data under a wide variety of conditions. On the AP1000 it would be used like this:

```
mpirun -nohost capsort [options] infile outfile
```

If the input and output files are the same then the file is sorted in place. If two files are specified then the input file is first copied to the output file then the data is again sorted in place in the output file.

There are a large number of options that can be applied. Here is a brief summary:

-p causes the data to be printed to standard output before and after sorting. This is useful for debugging.

-v causes the main program to verify the sort by calling qsort() within one processor and comparing the result. This will only work if the data fits in one processors memory and the input and output files are not the same file.

-D causes the main program to delete the file after sorting. This is useful for cleaning up after a test run.

-l NUM causes the main program to loop, calling the sort routine NUM times. At the end of each loop the parameters are changed. How they are changed is currently controlled by code changes in the main loop. This is useful for generating graphs.

-w causes the code to try to produce the worst case.

-g NUM causes the main program to generate NUM items of random data in the input file.

-s NUM seeds the random number generator with NUM.

-m NUM sets the memory limit per processor to NUM kB.

More detailed information on running the program can be obtained by reading the source code.

References

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

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

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

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

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

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

[7] *ibid*, solution to problem 5.3.4 (38).

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

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

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

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

[12] D. McIlroy, P. McIlroy and K. Bosti, *Engineering radix sort* Computing systems, vol 6, no 1, winter 1993

[13] Scherson, Sen and Shamir, *Shear sort: A True Two Dimensional Sorting Technique for VLSI Networks*, Tech report, Dept of Electrical and Computer Engineering, University of California, 1985

[14] C. P. Schnorr, *An optimal Sorting Algorithm For Mesh Connected Computers*, Proc. ACM Synmposium on the Theory of Computation, 1986

[15] A. Tridgell and D. Walsh , *The HiDIOS Filesystem, Proceedings of the Fujitsu CAP Workshop, Imperial College, London, October 1995.*