Efficient Algorithms for Sorting and Synchronization

Andrew Tridgell

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

July 1999

Except where otherwise indicated, this thesis is my own original work.

Andrew Tridgell
4 July 1999

Till min kära fru, Susan

Acknowledgments

The research that has gone into this thesis has been thoroughly enjoyable. That enjoyment is largely a result of the interaction that I have had with my supervisors, colleagues and, in the case of rsync, the people who have tested and used the resulting software.

I feel very privileged to have worked with my supervisors, Richard Brent, Paul Mackerras and Brendan McKay. To each of them I owe a great debt of gratitude for their patience, inspiration and friendship. Richard, Paul and Brendan have taught me a great deal about the field of Computer Science by sharing with me the joy of discovery and investigation that is the heart of research.

I would also like to thank Bruce Millar and Iain Macleod, who supervised my initial research in automatic speech recognition. Through Bruce, Iain and the Computer Sciences Laboratory I made the transition from physicist to computer scientist, turning my hobby into a career.

The Department of Computer Science and Computer Sciences Laboratory have provided an excellent environment for my research. I spent many enjoyable hours with department members and fellow students chatting about my latest crazy ideas over a cup of coffee. Without this rich environment I doubt that many of my ideas would have come to fruition.

The Australian Telecommunications and Electronics Research Board, the Commonwealth and the Australian National University were very generous in providing me with scholarship assistance.

Thanks also to my family who have been extremely understanding and supportive of my studies. I feel very lucky to have a family that shares my enthusiasm for academic pursuits.

Finally I'd like to thank my wonderful wife, Susan, who has encouraged me so much over the years. Many people told us that having two PhD students under the one roof is a recipe for disaster. Instead it has been fantastic.

Abstract

This thesis presents efficient algorithms for internal and external parallel sorting and remote data update. The sorting algorithms approach the problem by concentrating first on highly efficient but incorrect algorithms followed by a cleanup phase that completes the sort. The remote data update algorithm, rsync, operates by exchanging block signature information followed by a simple hash search algorithm for block matching at arbitrary byte boundaries. The last chapter of the thesis examines a number of related algorithms for text compression, differencing and incremental backup.

Contents

Introduction

While researching the materials presented in this thesis I have had the pleasure of covering a wider range of the discipline of Computer Science than most graduate students. My research began with the study of automatic speech recognition using Hidden Markov Models, decision trees and large Recurrent Neural Networks[Tridgell et al. 1992] but I was soon drawn into studying the details of parallel sorting algorithms when, while implementing my Neural Network code on a CM5 multicomputer, I discovered that suitable parallel sorting algorithms were unavailable.

I was surprised to find that practical parallel sorting algorithms offered a lot of scope for active research, perhaps more than the rather densely populated field of automatic speech recognition. As I have always enjoyed working on new algorithms, particularly algorithms where efficiency is of primary concern, I jumped at the chance to make a contribution in the field of parallel sorting. This led to the research presented in the first chapter of this thesis and away from my original field of study.

While completing this research I had the opportunity to participate in a cooperative research center research project called the PIOUS project[Tridgell 1996] in which I worked on a parallel filesystem for multicomputers. This research led to the creation of a parallel filesystem called HiDIOS[Tridgell and Walsh 1996] which led me to investigate the natural extension of the problem of internal parallel sorting – external parallel algorithm. The result is the research presented in the second chapter of this thesis.

The rest of the thesis is dedicated to the rsync algorithm which provides a novel method of efficiently updating files over slow network links. The rsync algorithm was a direct result of my work on parallel filesystems and external parallel sorting algorithms. The link is a simple text searching algorithm[Tridgell and Hawking 1996] that I developed and which is used by an application running on the HiDIOS parallel filesystem. It was this text searching algorithm which provided the seed from which the rsync algorithm was able to grow, although the algorithm presented in this thesis bears little resemblance to that seed.

This thesis is a testament to the multi-faceted nature of computer science and the many and varied links within this relatively new discipline. I hope you will enjoy reading it as much as I enjoyed the research which it describes.

Chapter 1

Internal Parallel Sorting

My first introduction to the problem of parallel sorting came from a problem in the implementation of an automatic speech recognition training program. A set of speech data needed to be normalized in order to be used as the input to a recurrent neural network system and I decided that a quick-and-dirty way of doing this would be to sort the data, then sample it at regular intervals to generate a histogram.

This is a terribly inefficient way of normalizing data in terms of computational complexity but is a quite good way in terms of software engineering because the availability of easy to use sorting routines in subroutine libraries makes the amount of coding required very small. I had already used this technique in the serial version of the speech recognition system so it seemed natural to do the same for the parallel version.

With this in mind I looked in the subroutine library of the parallel machine I was using (a Fujitsu AP1000 [Ishihata et al. 1991] running CellOS) and discovered that there was a problem with my plan – the library totally lacked a parallel sorting routine. I had been expecting that there would be a routine that is the parallel equivalent of the ubiquitous qsort() routine found in standard C libraries. The lack of such a routine was quite a surprise and prompted me to look into the whole issue of parallel sorting, totally ignoring the fact that I only wanted a parallel sorting routine in order to solve a problem that didn't really require sorting at all.

A survey of the literature on parallel sorting routines was rather disappointing. The routines were not at all general purpose and tended to place unreasonable restrictions on the amount of data to be sorted, the type of data and the number of CPUs in the parallel system. It became clear that the world (or at least my corner of it) needed a fast, scalable, general-purpose parallel sorting routine.

The rest of this chapter details the design, implementation and performance of just such a routine.

1.1 How fast can it go?

A question that arises when considering a new algorithm is "How fast can it go?". It helps greatly when designing an algorithm to understand the limits on the algorithm's efficiency. In the case of parallel sorting we can perform some simple calculations which are very revealing and which provide a great deal of guidance in the algorithm design.

It is well known that comparison-based sorting algorithms on a single CPU require $\log N!$ time[1], which is well approximated[Knuth 1981] by $N \log N$. This limitation arises from the fact the the unsorted data can be in one of $N!$ possible arrangements and that each individual comparison eliminates at most half of the arrangements from consideration.

An ideal parallel sorting algorithm which uses P processors would reduce this time by at most a factor of P, simply because any deterministic parallel algorithm can be simulated by a single processor with a time cost of P. This leads us to the observation that an ideal comparison-based parallel sorting algorithm would take time $\frac{N}{P} \log N$.

Of course, this totally ignores the constant computational factors, communication costs and the memory requirements of the algorithm. All those factors are very important in the design of a parallel sorting algorithm and they will be looked at carefully in a later part of this chapter.

1.1.1 Divide and conquer

We now consider parallel sorting algorithms which are divided into two stages. In the first stage each processor sorts the data that happens to start in its local memory and in the second stage the processors exchange elements until the final sorted result

[1] Throughput this thesis $\log x$ will be used to mean $\lceil \log_2 x \rceil$.

is obtained. It seems natural to consider dividing the algorithm in this manner as efficient algorithms for the first stage are well known.

How long would we expect each stage to take? The first stage should take $O(\frac{N}{P}\log\frac{N}{P})$ simply by using an optimally efficient serial sorting algorithm on each processor[2]. This clearly cannot be improved upon[3].

The more interesting problem is how long the second stage should take. We want the overall parallel sorting algorithm to take $O(\frac{N}{P}\log N)$ time which means we would ideally like the second stage to take $O(\frac{N}{P}\log P)$ time. If it turns out that this is not achievable then we might have to revisit the decision to split the algorithm into the two stages proposed above.

1.1.2 The parallel merging problem

The second stage of the parallel sorting algorithm that is now beginning to take shape is to merge P lists of N/P elements each stored on one of P processors. We would like this to be done in $O(\frac{N}{P}\log P)$ time.

1.1.2.1 The two processor problem

Let us first consider the simplest possible parallel merging problem, where we have just two processors and each processor starts with $\frac{N}{2}$ elements. The result we want is that the first processor ends up with all the small elements and the second processor ends up with all the large elements. Of course, "small" and "large" are defined by reference to a supplied comparison function.

We will clearly need some communication between the two processors in order to transmit the elements that will end up in a different processor to the one they start in, and we will need some local movement of elements to ensure that each processor ends up with a locally sorted list of elements[4].

[2] I shall initially assume that the data is evenly distributed between the processors. The problem of balancing will be considered in the detailed description of the algorithm.

[3] It turns out that the choice of serial sorting algorithm is in fact very important. Although there are numerous "optimal" serial sorting algorithms their practical performance varies greatly depending on the data being sorted and the architecture of the machine used.

[4] As is noted in Section 1.2.12 we don't strictly need to obtain a sorted list in each cell when this two processor merge is being used as part of a larger parallel merge but it does simplify the discussion.

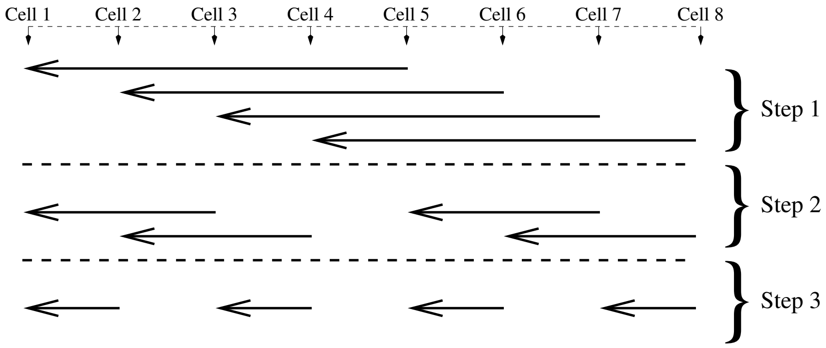

Figure 1.1: The parallelisation of an eight way hypercube merge

Both of these operations will have a linear time cost with respect to the number of elements being dealt with. Section 1.2.8 gives a detailed description of an algorithm that performs these operations efficiently. The basis of the algorithm is to first work out what elements will end up in each processor using a $O(\log N)$ bisection search and then to transfer those elements in a single block. A block-wise two-way merge is then used to obtain a sorted list within each processor.

The trickiest part of this algorithm is minimizing the memory requirements. A simple local merge algorithm typically requires order N additional memory which would restrict the overall parallel sorting algorithm to dealing with data sets of less than half the total memory in the parallel machine. Section 1.2.12 shows how to achieve the same result with $O(\sqrt{N})$ additional memory while retaining a high degree of efficiency. Merging algorithms that require less memory are possible but they are quite computationally expensive[Ellis and Markov 1998; Huang and Langston 1988; Kronrod 1969].

1.1.2.2 Extending to P processors

Can we now produce a P processor parallel merge using a series of two processor merges conducted in parallel? In order to achieve the aim of an overall cost of $O(\frac{N}{P}\log P)$ we would need to use $O(\log P)$ parallel two processor merges spread across the P processors.

The simplest arrangement of two processor merges that will achieve this time cost is a hypercube arrangement as shown for eight processors in Figure 1.1.

This seems ideal, we have a parallel merge algorithm that completes in $\log P$ parallel steps with each step taking $O(\frac{N}{P})$ time to complete.

There is only one problem, *it doesn't work*!

1.1.3 Almost sorting

This brings us to a central idea in the development of this algorithm. We have so far developed an algorithm which very naturally arises out of a simple analysis of the lower limit of the sort time. The algorithm is simple to implement, will clearly be very scalable (due to its hypercube arrangement) and is likely to involve minimal load balancing problems. All these features make it very appealing. The fact that the final result is not actually sorted is an annoyance that must be overcome.

Once the algorithm is implemented it is immediately noticeable that although the final result is not sorted, it is "almost" sorted. By this I mean that nearly all elements are in their correct final processors and that most of the elements are in fact in their correct final positions within those processors. This will be looked at in Section 1.3.6 but for now it is good enough to know that, for large N, the proportion of elements that are in their correct final position is well approximated by $1 - P/\sqrt{N}$.

This is quite extraordinary. It means that we can use this very efficient algorithm to do nearly all the work, leaving only a very small number of elements which are not sorted. Then we just need to find another algorithm to complete the job. This "cleanup" algorithm can be designed to work for quite small data sets relative to the total size of the data being sorted and doesn't need to be nearly as efficient as the initial hypercube based algorithm.

The cleanup algorithm chosen for this algorithm is Batcher's merge-exchange algorithm[Batcher 1968], applied to the processors so that comparison-exchange operations are replaced with merge operations between processors. Batcher's algorithm is a sorting network[Knuth 1981], which means that the processors do not need to communicate in order to determine the order in which merge operations will be performed. The sequence of merge operations is predetermined without regard to the data being sorted.

1.1.4 Putting it all together

We are now in a position to describe the algorithm as a whole. The steps in the algorithm are

- distribute the data over the P processors
- sort the data within each processor using the best available serial sorting algorithm for the data
- perform $\log P$ merge steps along the edges of a hypercube
- find which elements are unfinished (this can be done in $\log(N/P)$ time)
- sort these unfinished elements using a convenient algorithm

Note that this algorithm arose naturally out of a simple consideration of a lower bound on the sort time. By developing the algorithm in this fashion we have guaranteed that the algorithm is optimal in the average case.

1.2 Algorithm Details

The remainder of this chapter covers the implementation details of the algorithm, showing how it can be implemented with minimal memory overhead. Each stage of the algorithm is analyzed more carefully resulting in a more accurate estimate of the expected running time of the algorithm.

The algorithm was first presented in [Tridgell and Brent 1993]. The algorithm was developed by Andrew Tridgell and Richard Brent and was implemented by Andrew Tridgell.

1.2.1 Nomenclature

P is the number of nodes (also called cells or processors) available on the parallel machine, and N is the total number of elements to be sorted. N_p is the number of elements in a particular node p ($0 \le p < P$). To avoid double subscripts N_{p_j} may be written as 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 $0 \le p < P$. $E_{j,i}$ may be used instead of $E_{p_j,i}$ if no confusion will arise.

When giving "big O" time bounds the reader should assume that P is fixed so that $O(N)$ and $O(N/P)$ are the same.

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 $E_{p,i} \leq E_{p,j}$ for all relevant $i < j$ and p, and that $E_{p,i} \leq E_{q,j}$ for $0 \leq p < q < P$ 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.

1.2.2 Aims of the Algorithm

The design of the algorithm had several aims:

- Speed.
- Good memory utilization. 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 two. These are common restrictions in parallel sorting algorithms [Ajtai et al. 1983; Akl 1985].

In order for the algorithm to be truly general purpose the only operator that will be assumed is binary comparison. This rules out methods such as radix sort [Blelloch et al. 1991; Thearling and Smith 1992].

It is also assumed that elements are of a fixed size, because of the difficulties of pointer representations between nodes in a MIMD machine.

To obtain good memory utilization when sorting small elements linked lists are avoided. 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 some respects to parallel shellsort [Fox et al. 1988], but contains a number of new features. For example, the memory overhead of the algorithm is considerably reduced.

1.2.3 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 [Knuth 1981, solution to problem 5.3.4.38] 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 small examples show[5]. However, a simple extension of the algorithm, which will be referred to as *infinity padding*, can remove this restriction[6].

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 $M \times P$. After sorting is complete the padding elements could be found and removed from the tail of the 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, we 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

[5] A small example where unequal sized lists fails with Batchers's merge-exchange sorting network is a 4 way sort with elements ([1] [0] [1] [0 0]) which results in the unsorted data set ([0] [0] [1] [0 1]).

[6] A method for avoiding infinity-padding using balancing is given in Section 1.2.5 so infinity-padding is not strictly needed but it provides some useful concepts nonetheless.

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

Figure 1.2: Pseudo-code for load balancing

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

$$N_2' = \max(0, N_1 + N_2 - M)$$

and

$$N_1' = N_1 + N_2 - N_2'$$

This means that if at each merge step 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.

1.2.4 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 power of 2. Pseudo-code is shown in Figure 1.2[7].

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

[7] The pseudo-code in this thesis uses the C convention of integer division.

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.

1.2.5 Perfect Balancing

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 1.2.3, 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 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.

1.2.6 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 should be used, subject to the restriction that the algorithm must be comparison-based[8].

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, experiments showed 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)

[8]The choice of the best serial sorting algorithm is quite data and architecture dependent.

```
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 1.3: Pseudo-code for primary merge

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[GNU 1998]. 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 used in the AP1000[Ishihata et al. 1993].

1.2.7 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[Batcher 1968] as used in the cleanup phase).

The pattern of merge-exchange operations in the primary merge is identical to that used in the pre-balancing phase of the algorithm. The pseudo-code for the algorithm is given in Figure 1.3. 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 pre-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.

1.2.8 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[Fox et al. 1988] 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 minimized.

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 1.4.

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

```
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 1.4: Pseudo-code for a simple merge

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.

1.2.9 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 node's 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 node's elements before the merge begins.

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

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

When a comparison is made between 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 $E_{1,A-1} \le E_{2,N_1'-A}$ 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[Zhou et al. 1993]. 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.

1.2.10 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.

1.2.11 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, i.e. 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.

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.

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.

1.2.12 Block-wise Merging

The block-wise merge is a solution to the problem of merging two sorted lists of elements into one, while using only a small amount of additional storage. The first phase in the operation is to break the two lists into blocks of an equal size B. The exact value of B is unimportant for the functioning of the algorithm and only makes a difference to the efficiency and memory usage of the algorithm. It will be assumed that B is $O(\sqrt{L_1 + L_2})$, which is small relative to the memory available on each node. To simplify the exposition 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 my implementation of the 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.

1.2.13 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 [Knuth 1981, Algorithm M, page 112]. 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 generalize the notion of an element to include all elements in a node. This means that the comparison-exchange operation must make all elements in the first node less than all elements in the second. This is identical to the operation of the merge-exchange algorithm. The fact that it is possible to make this generalization while maintaining the correctness of the algorithm is discussed in Section 1.2.3.

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 [Fox et al. 1988]. 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.

1.3 Performance

In this section the performance of my implementation of the parallel sorting algorithm given above will be examined, primarily on a 128 node AP1000 multicomputer. While the AP1000 is no longer a state of the art parallel computer it does offer a reasonable number of processors which allows for good scalability testing.

1.3.1 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 that which is called 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 realizable, 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 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, I have extrapolated the timing results of the serial algorithm to larger N.

The quicksort/insertion-sort algorithm which I 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

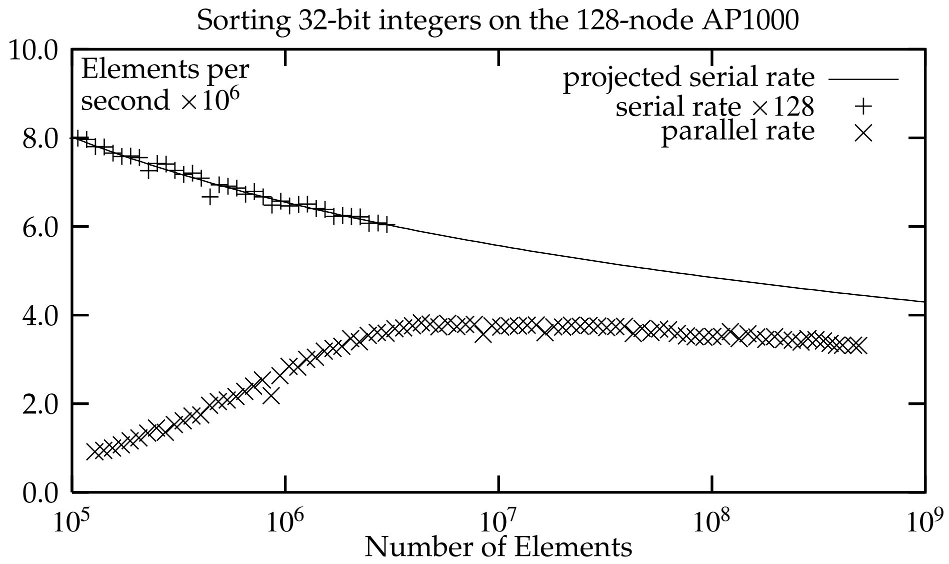

Figure 1.5: Sorting 32-bit integers on the AP1000

estimate these contributions I have performed a least squares fit of the form:

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

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.

1.3.2 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 1.5. 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 AP1000's 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.

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 1.3.1.

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 roll-off 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 for large N.

A similar result for sorting of 16-byte random strings is shown in Figure 1.6. 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.

1.3.3 Scalability

An important aspect of a parallel algorithm is its scalability, which measures the ability of the algorithm to utilize additional nodes. Shown in Figure 1.7 is the result of sorting 100,000 16-byte strings per node on the AP1000 as the number of nodes is var-

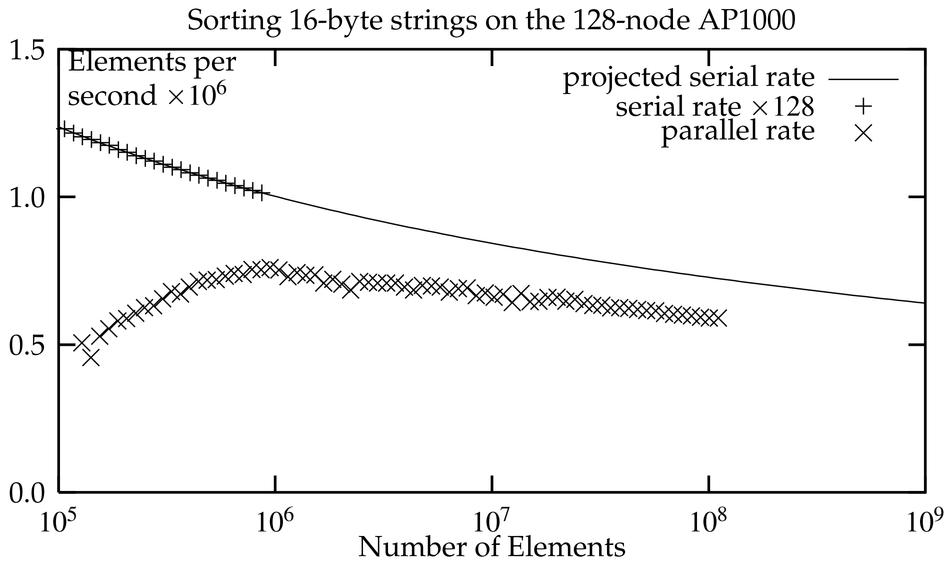

Figure 1.6: Sorting 16-byte strings on the AP1000

ied. 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 optimized 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.

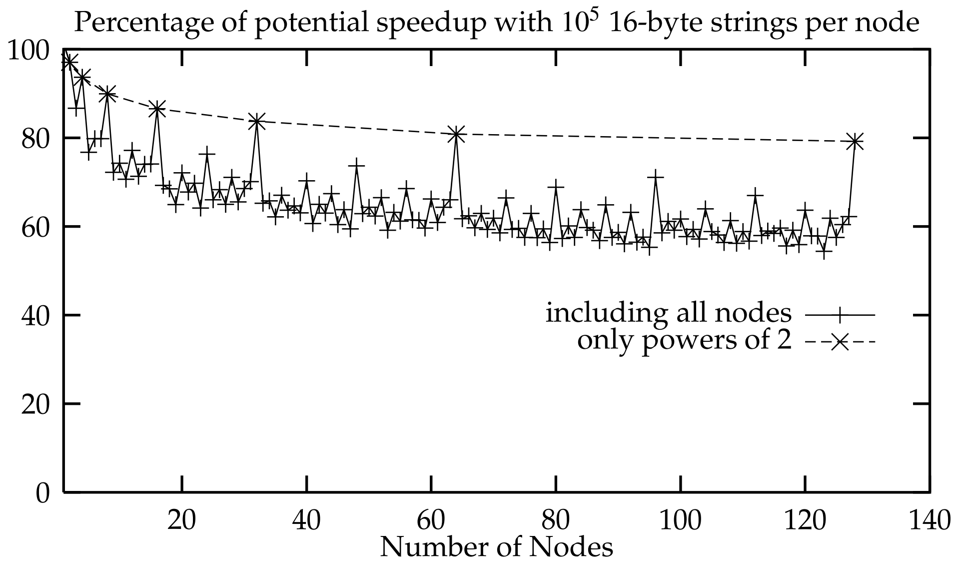

Figure 1.7: Scalability of sorting on the AP1000

1.3.4 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 1.8 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 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 [Natvig 1990].

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.

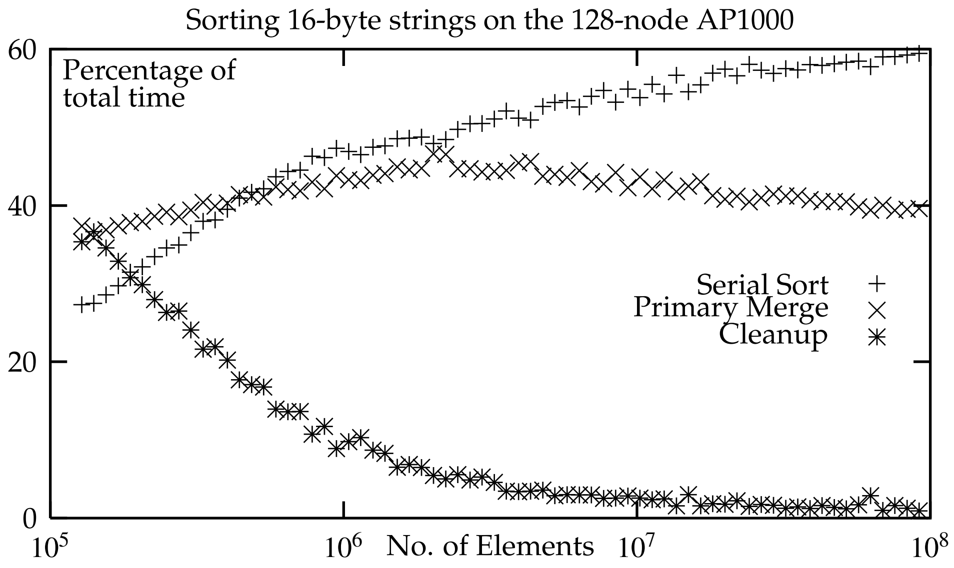

Figure 1.8: Timing breakdown by phase

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 1.9 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 idle (waiting for messages) and communicating decreases steadily as N increases. From the point of view of the parallel speedup of the algorithm these tasks are wasted time and need to be kept to a minimum.

1.3.5 CM5 vs AP1000

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

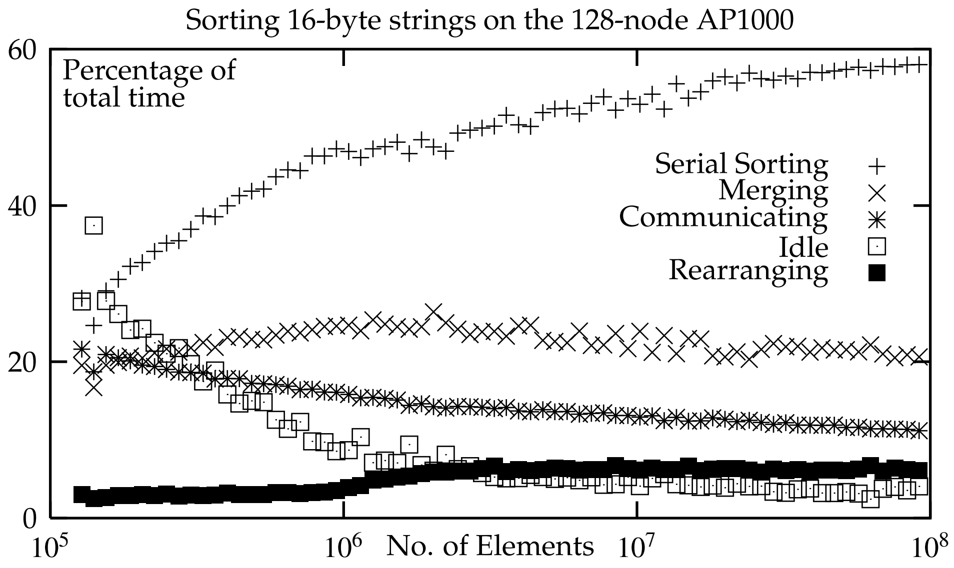

Figure 1.9: Timing breakdown by task

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 widely 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 completely representative of results that are regularly achieved in a real application.

In Table 1.1 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 both machines 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

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

Table 1.1: Sort times (seconds) for 8 million integers

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 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.

1.3.6 Primary Merge Effectiveness

The efficiency of the parallel sorting algorithm relies on the fact that after the primary merge phase most elements are in their correct final positions. This leaves only a small amount of work for the cleanup phase.

Figure 1.10 shows the percentage of elements which are in their correct final positions after the primary merge when sorting random 4 byte integers on a 64 and 128 processor machine. Also shown is $100\left(1 - P/\sqrt{N}\right)$ which provides a very good approximation to the observed results for large N.

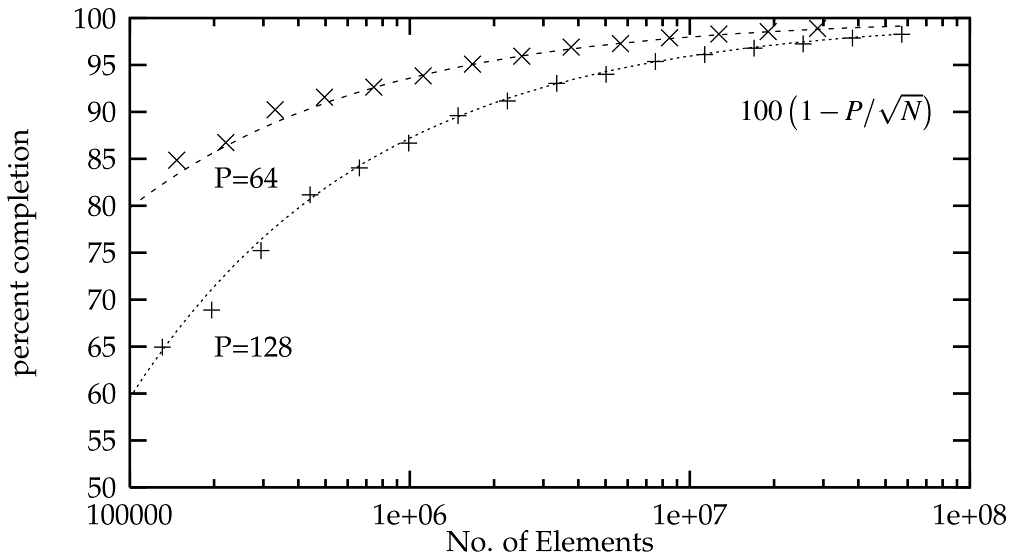

Figure 1.10: Percentage completion after the primary merge

It is probably possible to analytically derive this form for the primary merge percentage completion but unfortunately I have been unable to do so thus far. Nevertheless it is clear from the numerical result that the proportion of unsorted elements remaining after the primary merge becomes very small for large N. This is important as it implies that the parallel sorting algorithm is asymptotically optimal.

1.3.7 Optimizations

Several optimization "tricks" have been used to obtain faster performance. It was found that these optimizations played a surprisingly large role in the speed of the algorithm, producing an overall speed improvement of about 50%.

The first optimization 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 optimization was the tuning of the block size of sends performed when elements are exchanged between nodes. This optimization 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 block-wise 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 block-wise 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 optimizations to the code are possible and would lead to further improvements in performance.

1.4 Comparison with other algorithms

There has been a lot of research into parallel sorting algorithms. Despite this, when I started my work on sorting in 1992 I was unable to find any general purpose efficient parallel sorting algorithms suitable for distributed memory machines. My research attempted to fill in that gap.

This section examines a number of parallel sorting algorithms where performance results are available for machines comparable with those that my research used.

1.4.1 Thearling and Smith

In [Thearling and Smith 1992] Kurt Thearling and Stephen Smith propose a parallel integer sorting benchmark and give results for an implementation of an algorithm on the CM5. Their algorithm is based on a radix sort, and their paper concentrates on the effects of the distribution of the keys within the key space using an entropy based measure.

The use of a radix based algorithm means that the algorithm is not comparison-based and thus is not general purpose. Additionally, the sending phase of the algorithm consumes temporary memory of $O(N)$ which reduces the number of elements that can be sorted on a machine of a given size by a factor of 2.

The results given in their paper are interesting nonetheless. The time given for sorting 64 million uniformly distributed 32 bit elements on 64 CM5 processors was 17 seconds.

To provide a comparison with their results I used an in-place forward radix sort (based on the American flag sort[McIlroy et al. 1993]) for the local sorting phase of our algorithm and ran a test of sorting 64 million 32 bit elements on 64 cells of the AP1000. I then scaled the timing results using the ratio of sorting speed for the AP1000 to the CM5 as observed in earlier results[9]. This gave a comparative sorting time of 17.5 seconds. By using a more standard (and memory consuming) radix sort a time of 15.8 seconds was obtained.

The interesting part of this result is how different the two algorithms are. Their algorithm uses special prefix operations of the CM5 to compute the radix histograms in parallel then uses a single all to all communication stage to send the data directly to the destination cell. About 85% of the total sort time is spent in the communication stage. Despite the totally different algorithm the results are quite similar.

1.4.2 Helman, Bader and JaJa

In [Helman et al. 1996] David Helman, David Bader and Joseph JaJa detail a sample sort based parallel sorting algorithm and give results for a 32 node CM5, directly comparing their results with the results for our algorithm as presented in [Tridgell and Brent 1995; Tridgell and Brent 1993]. They also compare their results with a wide range of other parallel sorting algorithms and show results indicating that their algorithm is the fastest in most cases.

Like the Thearling algorithm Helman's algorithm uses a radix sort, although in this case it is applied independently to the elements in each node. This still leaves them with an $O(N)$ memory overhead and the problem that the algorithm is not general purpose because a comparison function cannot be used in a radix sort.

The table they show comparing their algorithm with ours (which they call the TB algorithm) shows results for sorting 8 million uniformly distributed integers on a 32 node CM5. They give the results as 4.57 seconds for their algorithm and 5.48 seconds for ours.

[9] The CM5 at ANU had been decommissioned by the time the Thearling paper came out, so I couldn't test directly on the same hardware.

To give a fairer comparison I again replaced the quicksort algorithm in my implementation with the American flag in-place radix sort and scaled for AP1000 to CM5 differences. This gave an equivalent sorting time of 3.9 seconds.

1.5 Conclusions

The parallel sorting algorithm presented in this chapter arises from a very simple analysis following a decision to split the algorithm into a local sorting phase followed by a parallel merge. The use of a fast but incorrect initial merge phase followed by a simple cleanup algorithm leads to a very high degree of parallel efficiency.

As far as I am aware this is the only efficient, comparison-based distributed memory parallel sorting algorithm that requires less than order N temporary storage. These features, combined with the competitive sorting performance, make the algorithm suitable for a parallel algorithms library.

Chapter 2

External Parallel Sorting

This chapter considers the problem of external parallel sorting. External sorting involves sorting more data than can fit in the combined memory of all the processors on the machine. This involves using disk as a form of secondary memory, and it presents some very interesting challenges because of the huge difference between the bandwidths and latencies of memory and disk systems.

2.1 Parallel Disk Systems

There are a large variety of different disk I/O systems available for parallel computers. In some systems each CPU has its own private disk and in others there is a central bank of disks accessible from all CPUs.

For this work I assumed the following setup:

- a distributed memory parallel computer
- a parallel filesystem where all CPUs can access all the data

The particular setup I used was a 128 cell AP1000 with 32 disks and the HiDIOS parallel filesystem[Tridgell and Walsh 1996].

The requirement of a filesystem where all CPUs can access all the data isn't as much of an impediment as it might first seem. Almost any distributed filesystem can easily be given this property using a remote access protocol coupled to the message passing library[1].

[1] The AP1000 has only local disk access in hardware. Remote disk access is provided by the operating system.

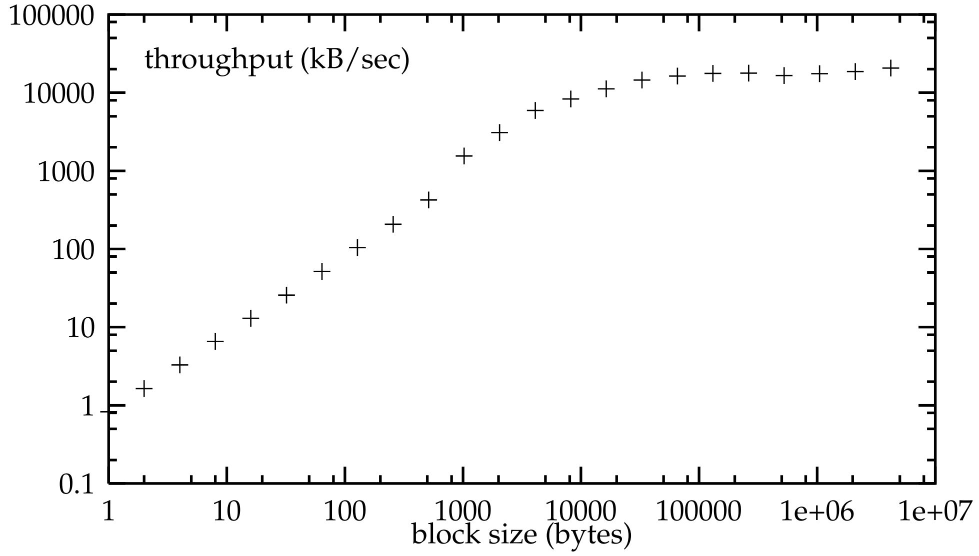

Figure 2.1: Aggregate throughput with varying block-size

2.2 Designing an algorithm

An external parallel sorting algorithm has design constraints quite different from an internal algorithm. The main differences are:

- the average number of I/O operations per element needs to be minimized.
- the I/O requests need to be gathered as far as possible into large requests
- the algorithm should, if possible, run in-place to allow data sets up to the full size of the available disk space to be sorted

The dramatic difference that the gathering of I/O requests can make is demonstrated by the write throughput results in Figure 2.1. The graph show the aggregate write throughput on a 128 cell AP1000 with 16 disks[2]. With the throughput varying by four orders of magnitude depending on the size of the individual writes it is clear that an algorithm that does a smaller number of larger I/O requests is better than one which does a large number of small requests.

[2]Some results in this chapter are for a system with 32 disks and others are for a system with 16 disks. The machine I used was reconfigured part way through my research.

2.2.1 Characterizing parallel sorting

An external sorting algorithm needs to make extensive use of secondary memory (usually disk). In this chapter I will characterize the external sorting using a parameter k where $k = \lceil \frac{N}{M} \rceil$ where N is the number of elements being sorted and M is the number of elements that can fit in the internal memory of the machine. 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.

In practice k is rarely very large as large parallel disk systems are usually attached to computers with fairly large memory subsystems. On the AP1000 available to me k is limited to 8 (a total of 2GB of RAM and 16GB of disk). To simulate the effect of a larger value for k some of the test runs use a deliberately limited amount of memory.

2.2.2 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.

The most obvious lower limit is at least one read and one write per element. This stems from the fact that every element has to be moved at least once to change the order of the elements.

Can this limit be achieved? It is fairly easy to show that it cannot. Imagine we had an algorithm that performed only one read and one write per element. As the elements cannot be predicted in advance we cannot place any element in its correct output position with certainty until all elements have been read. That implies that all the reads must be performed before all the writes. In that case the algorithm would have to store internally (in memory) enough information to characterize the ordering of the elements. The memory required to do this for non-compressible data is at least $\log(N!)$ bits, which is approximately $N \log(N)$. With $k > 1$ this cannot be achieved[3].

The algorithm presented in this chapter demonstrates that external parallel sorting can, however, be done with an average of approximately two reads and writes per

[3] Strictly it could be done if the size of each element is greater than $k \log N$.

element, at least for values of k that are practical on the machines available to me.

2.2.3 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[4]. The columns and rows of the grid are then alternately sorted using the internal parallel sorting algorithm described above. It can be demonstrated[Schnorr and Shamir 1986] that the upper limit on the number of column and row sorts required is $\lceil \log k \rceil + 1$.

This basic algorithm is augmented with a number of optimizations 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 shearsort algorithm[Schnorr and Shamir 1986].

2.2.4 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 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 2.2 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.

[4] This algorithm was first described in [Tridgell et al. 1997].

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 2.2: Partitioning with $k = 3$ and $P = 6$

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 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[5].

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 an iterative procedure is used which first estimates k and assigns slices, then increases k until the restrictions are 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[6]. 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 processor's 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 2.2.7.

2.2.5 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.

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

[6]Slices must begin and end on element boundaries.

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.

2.2.6 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 shearsort 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 been 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 checks 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; and
- 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].

2.2.7 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 serialized 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 processors the allocation of slices to processors is made dynamically. Thus while one row is being sorted any unused processors can begin the task of sorting the next row. Even if too few unused 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, thus overlapping I/O with computation.

This dynamic 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 which slices have not completed and can separately calculate the allocation of processors to slices.

2.2.8 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 shearsort may be used which work well for much larger values of k. For example, reverse-sort[Schnorr and Shamir 1986] has a worst case of $\lceil \log \log k \rceil$ for large k. For smaller k, however, it has no advantage over shear-

[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.

sort. Values of k large enough to warrant the use of reverse-sort are not investigated in this thesis.

2.2.9 Other partitionings

It may not be obvious from the above why a 2 dimensional partitioning was chosen. Other partitionings were considered but 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 some 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 I haven't seen one yet.

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 and maximizes the size of individual I/O operations.

2.3 Performance

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 possible when the full internal memory of the machine is used. In practice a k of above

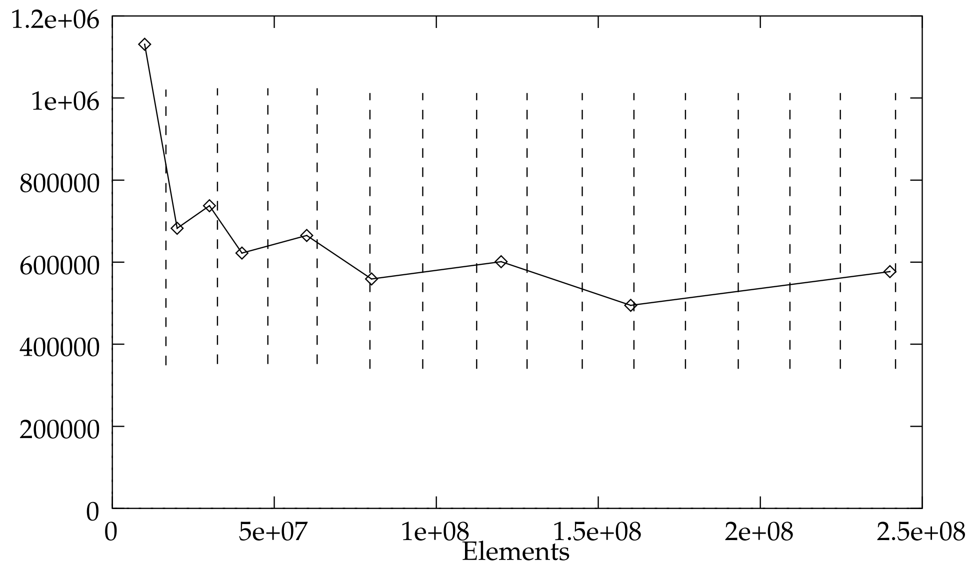

Figure 2.3: External sorting of 64 bit integers with fixed memory. The y axis is in elements per second. The dashed vertical lines show the boundaries between k values, with the left-most point at $k = 1$ and the right-most at $k = 15$.

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.

In the results that follow the local sorting phase of the internal parallel sorting algorithm was implemented using the American flag sort[McIlroy et al. 1993], an efficient form of in-place forward radix sort. A comparison-based local sort could equally well have been used.

Figure 2.3 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 granularity of k. At the point just above where k changes from 1 to 2 the number of elements being sorted

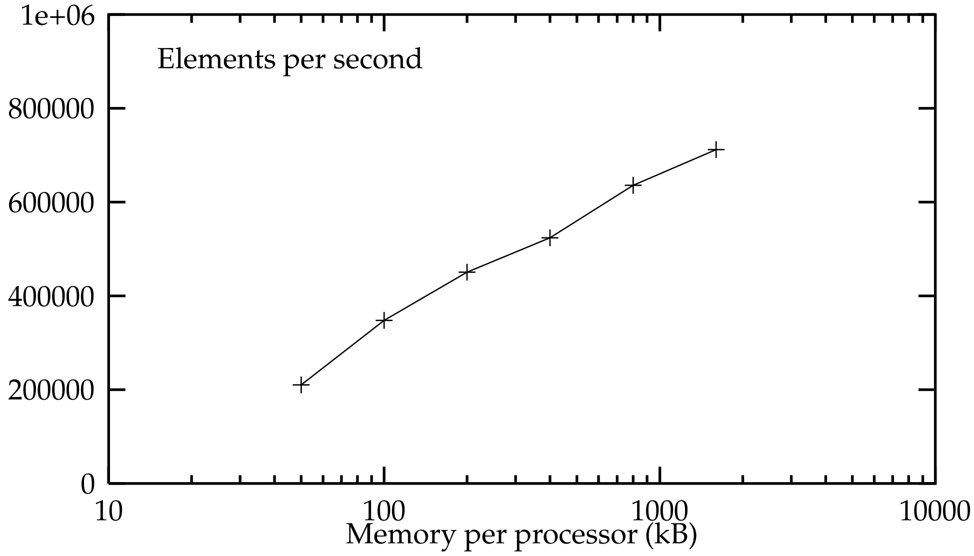

Figure 2.4: External sorting of 50 million 64 bit integers with varying amounts of available memory. The k values (from left to right) are 62, 31, 16, 8, 4 and 2

by the internal parallel sorting algorithm will be approximately half the number of elements being sorted at the upper limit of the $k = 2$ range. This results in significantly decreased efficiency for the internal algorithm (see Figure 1.5) which is reflected in the external parallel sorting times. A similar effect occurs for each value of k, leading to the observed variance.

Figure 2.4 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[8].

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 2.5 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.

[8]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!

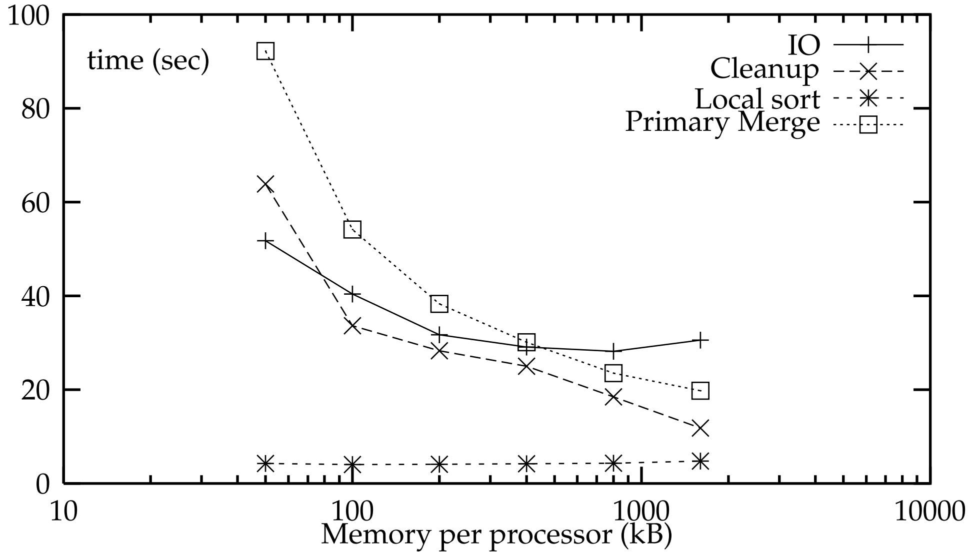

Figure 2.5: Breakdown of cost of external sorting as amount of available memory varies

2.3.1 I/O 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 2.6 shows the number of reads and writes required per element for random 64 bit integers. The graph shows that the algorithm achieves close to 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.

2.3.2 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.

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

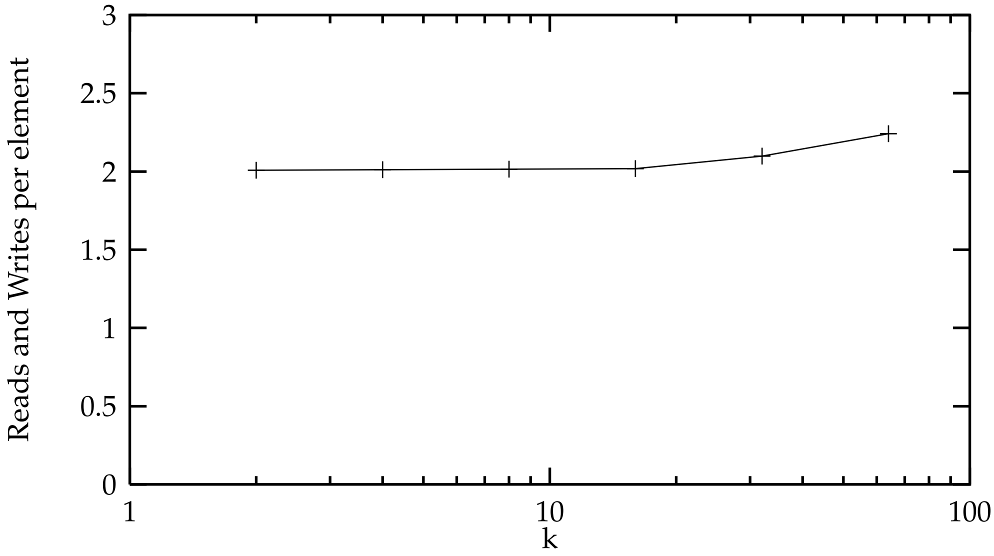

Figure 2.6: Number of reads and writes per element for a range of k values

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.

2.3.3 First pass completion

An important component of the algorithm is the fact that most slices are completed in the first pass, allowing the remaining passes to "clean up" the unfinished elements quickly. This is analogous to the primary and cleanup phases of the internal parallel sorting algorithm discussed in the previous chapter.

Figure 2.7 shows the percentage of slices that are completed after the first pass when sorting 50 million 64 bit random elements on the 128 processor AP1000 for varying values of k. Note the narrow y range on the graph. The results show that even for quite high values of k nearly 90% of slices are completed after the first pass.

2.4 Other Algorithms

The field of external parallel sorting is one for which direct comparison between algorithms can be difficult. The wide variation in the configuration of disk and memory hierarchies combined with varying requirements for data types and key sizes leaves

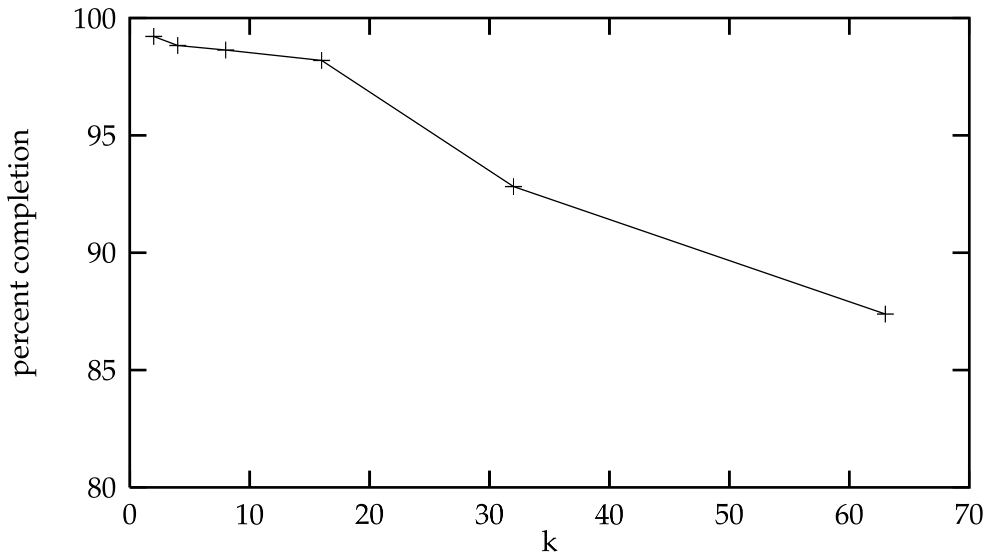

Figure 2.7: Percentage of slices completed after the first pass for varying k

little common ground between many published results. The best that can be done is to make broad comparisons.

Akl provides external parallel sorting algorithms for disks and tapes by adapting an internal parallel sorting algorithm[Akl 1985]. The result is an algorithm that performs at least $2\log P$ I/O operations per element which is excessive except for very small machines.

Aggarwal and Plaxton introduce the hypercube-based sharesort external parallel sorting algorithm[Aggarwal and Plaxton 1993] and demonstrate that it is asymptotically optimal in terms of I/O operations on general multi-level storage hierarchies. Their algorithm is aimed at larger values of k than are considered in this chapter and concentrates on providing a framework for expressing very generalised models of external parallel sorting.

Nodine and Vitter introduce an external parallel sorting algorithm called balance-sort[Nodine and Vitter 1992] which, like sharesort, is asymptotically optimal in the number of I/O operations but assumes a CRCW-PRAM model of parallel architecture which limits its practical usefulness.

I am not aware of any other external parallel sorting algorithms which take advantage of the early completion that is the key to the algorithm presented in this chapter.

The use of early completion detection results in an algorithm which uses a very small average number of I/O operations per element while maintaining a simple algorithmic structure.

2.5 Conclusions

The external parallel sorting algorithm presented in this chapter has some quite strong similarities to the internal parallel sorting algorithm from the previous chapter. Both get most of their work done with a first pass that leaves the vast majority of elements in their correct final position. In the case of the external algorithm this first pass is the natural first step in the overall algorithm whereas for the internal algorithm the first pass is logically separated from the cleanup phase.

The external algorithm also makes very efficient use of the disk subsystem by using I/O operations that are of a size equal to the internal memory size of each of the processors. This makes an enormous difference to the practical efficiency of the algorithm.

Chapter 3

The rsync algorithm

This chapter describes the rsync algorithm, an algorithm for the efficient remote update of data over a high latency, low bandwidth link. Chapter four describes enhancements and optimizations to the basic algorithm, and chapter five describes some of the more interesting alternative uses for the ideas behind rsync that have arisen since the algorithm was developed.

3.1 Inspiration

The rsync algorithm was developed out of my frustration with the long time it took to send changes in a source code tree across a dialup network link. I have spent a lot of time developing several large software packages during the course of my PhD and constantly finding myself waiting for source files to be transferred through my modem for archiving, distribution or testing on computers at the other end of the link or on the other side of the world. The time taken to transfer the changes gave me plenty of opportunity to think about better ways of transferring files.

In most cases these transfers were actually updates, where an old version of the file already existed at the other end of the link. The common methods for file transfer (such as ftp and rcp) didn't take advantage of the old file. It was possible to transfer just the changes by keeping original files around then using a differencing utility like *diff* and sending the diff files to the other end but I found that very inconvenient in practice and very error prone.

So I started looking for a way of updating files remotely without having prior knowledge of the relative states of the files at either end of the link. The aims of such an algorithm are:

- it should work on arbitrary data, not just text;
- the total data transferred should be about the size of a compressed diff file;
- it should be fast for large files and large collections of files;
- the algorithm should not assume any prior knowledge of the two files, but should take advantage of similarities if they exist;
- the number of round trips should be kept to a minimum to minimize the effect of transmission latency; and
- the algorithm should be computationally inexpensive, if possible.

3.2 Designing a remote update algorithm

Say you have two computers A and B connected by a low-bandwidth high-latency link. At the start of the transfer you have a file with bytes a_i on A and a file with bytes b_i on B. Assume for the sake of discussion that $0 \leq i < n$ so that each file is n bytes long[1]. The aim of the algorithm is for B to receive a copy of the file from A.

The basic structure of a remote update algorithm will be:

1. B sends some data S based on b_i to A
2. A matches this against a_i and sends some data D to B
3. B constructs the new file using b_i, S and D

The questions then is what form S will take, how A uses S to match on a_i and how B reconstructs a_i.

Even with this simple outline we can already see that the algorithm requires a probabilistic basis to be useful. The data S that B sends to A will need to be much smaller than the complete file in order for there to be any significant speedup[2]. This

[1] It turns out that the two files being the same length is unimportant, but it does simplify the nomenclature a little.

[2] Unless the link is asymmetric. If the link was very fast from B to A but slow from A to B then it wouldn't matter how big S is.

means that S cannot uniquely identify all possible files b_i which means that the algorithm must have a finite probability of failure. We will look at this more closely after the algorithm has been fleshed out some more[3].

3.2.1 First attempt

An example of a very simple form for the algorithm is

1. B divides b_i into N equally sized blocks b'_j and computes a signature S_j on each block[4]. These signatures are sent to A.

2. A divides a_i into N blocks a'_k and computes S'_k on each block.

3. A searches for S_j matching S'_k for all k

4. for each k, A sends to B either the block number j in S_j that matched S'_k or a literal block a'_k

5. B constructs a_i using blocks from b_i or literal blocks from a_i.

This algorithm is very simple and meets some of the aims of our remote update algorithm, but it is useless in practice[5]. The problem with it is that A can only find matches that are on block boundaries. If the file on A is the same as B except that one byte has been inserted at the start of the file then no block matches will be found and the algorithm will transfer the whole file.

3.2.2 A second try

We can solve this problem by getting A to generate signatures not just at block boundaries, but at all byte boundaries. When A compares the signature at each byte boundary with each of the signatures S_j on block boundaries of b_i it will be able to find matches at non-block offsets. This allows for arbitrary length insertions and deletions between a_i and b_i to be handled.

---

[3]While there has been some earlier information theoretic work on the remote update problem[Orlitsky 1993], that research did not lead to a practical algorithm.

[4]Think of the blocks as being a few hundred bytes long, we will deal with the optimal size later.

[5]The above algorithm is the one most commonly suggested to me when I first describe the remote update problem to colleagues.

This would work, but it is not practical because of the computational cost of computing a reasonable signature on every possible block. It could be made computationally feasible by making the signature algorithm very cheap to compute but this is hard to do without making the signature too weak. A weak signature would make the algorithm unusable.

For example, the signature could be just the first 4 bytes of each block. This would be very easy to compute but the algorithm would fail to produce the right result when two different blocks had their first 4 bytes in common.

3.2.3 Two signatures

The solution (and the key to the rsync algorithm) is to use not one signature per block, but two. The first signature needs to be very cheap to compute for all byte offsets and the second signature needs to have a very low probability of collision. The second, expensive signature then only needs to be computed by A at byte offsets where the cheap signature matches one of the cheap signatures from B.

If we call the two signatures R and H then the algorithm becomes[6]:

1. B divides b_i into N equally sized blocks b'_j and computes signatures R_j and H_j on each block. These signatures are sent to A.

2. For each byte offset i in a_i A computes R'_i on the block starting at i.

3. A compares R'_i to each R_j received from B.

4. For each j where R'_i matches R_j A computes H'_i and compares it to H_j.

5. If H'_i matches H_j then A sends a token to B indicating a block match and which block matches. Otherwise A sends a literal byte to B.

6. B receives literal bytes and tokens from A and uses these to construct a_i.

For this algorithm to be effective and efficient we need the following conditions:

- the signature R needs to be cheap to compute at every byte offset in a file;

[6] I call them R and H for *rolling checksum* and *hash* respectively. Hopefully those names will become clear shortly.

- the signature H needs to have a very low probability of random collision; and
- A needs to perform the matches on all block signatures received from B very efficiently, as this needs to be done at all byte offsets.

Most of the rest of this chapter deals with the selection of the two signature algorithms, and the related problem of implementing the matching function efficiently.

3.2.4 Selecting the strong signature

The strong signature algorithm is the easier of the two. It doesn't need to be particularly fast as it is only computed on block boundaries by B and at byte boundaries on A only when the fast signature matches.

The main property that the algorithm must have is that if two blocks are different they should have a very low probability of having the same signature. There are many well known algorithms that have this property, perhaps the best known being the *message digest* algorithms commonly used in cryptographic applications. These algorithms are believed to have the following properties (where b is the number of bits in the signature)[Schneier 1996]:

- The probability that a randomly generated block has the same signature as a given block is $O(2^{-b})$.
- The computational difficulty of finding a second block that has the same signature as a given block is $\Omega(2^b)$.
- The individual bits in the signature are uncorrelated and have a uniform distribution.

These properties make a message digest algorithm ideal for rsync. The particular algorithm that is used for most of the results in this chapter is the 128 bit MD4 message digest[Rivest 1990]. This algorithm was chosen because of the ready availability of source code implementations and the high throughput compared to many other algorithms. Although MD4 is thought to be not as cryptographically strong as some later algorithms (such as MD5 or IDEA) the difference is unimportant for rsync.

In reality, MD4 is “overkill” as the strong signature algorithm for rsync. It would be quite possible to use a cryptographically weaker but computationally less expensive algorithm such as a simple polynomial based algorithm. This wasn’t done because testing showed that the MD4 computation does not provide a significant bottleneck on modern CPUs. For example, on a 200 MHz Pentium processor the MD4 implementation achieved 6 MB/sec throughput, which is far in excess of most local area networks. As rsync is aimed at low bandwidth networks the computational cost of MD4 is insignificant[7].

3.2.5 Selecting the fast signature

The role of the fast signature in rsync is to act as a filter, preventing excessive use of the strong signature algorithm. The most important features is that it needs to be able to be computed very cheaply at every byte offset in a file.

The first fast signature algorithm that I investigated was just a concatenation of the first 4 and last 4 bytes of each block. Although this worked, it had the serious flaw that for certain types of structured data (such as tar files) sampling a subset of the bytes in a block provided a very poor signature algorithm. It was not uncommon to find a large proportion of byte offsets in a file that had the same fast signature, which resulted in an excessive use of the strong signature algorithm and very high computational cost.

This led to the investigation of fast signature algorithms which depended on all the bytes in a block, while requiring very little computation to find the signature values for every byte offset in a file. Perhaps the simplest such algorithm is

$$R(a) = \sum a_i$$

This would be very fast because it can be computed incrementally with one addition and one subtraction to “slide” the signature from one byte offset to the next[8]. The problem with this signature is that it is independent of the order of the bytes in the buffer. This means that, when sliding the buffer, if we add the same byte on the end of

[7] The use of MD4 also seemed to be an important factor in convincing skeptical users of the safety of the file transfer algorithm. The fact that a failed transfer is equivalent to “cracking” MD4 allayed suspicions significantly and led to the faster adoption of the algorithm.

[8] I will sometimes refer to this property as the *rolling* property.

the buffer as we remove from the start then we end up with the same signature value.

The solution is to make the signature order dependent by introducing factors dependent on i into the signature. The algorithm that was chosen is defined by[9]

$$
\begin{aligned}
r_1(k,L) &= (\sum_{i=0}^{L-1} a_{i+k}) \bmod M \\
r_2(k,L) &= (\sum_{i=0}^{L-1} (L-i)a_{i+k}) \bmod M \\
r(k,L) &= r_1(k,L) + M r_2(k,L)
\end{aligned}
$$

where $r(k,L)$ is the signature at offset k for a block of length L. M is an arbitrary modulus, and was chosen to be 2^{16} for simplicity and speed. Note that this results in a 32 bit signature.

This signature can be computed incrementally as follows

$$
\begin{aligned}
r_1(k+1,L) &= (r_1(k,L) - a_k + a_{k+L}) \bmod M \\
r_2(k+1,L) &= (r_2(k,L) - La_k + r_1(k+1,L)) \bmod M \\
r(k+1,L) &= r_1(k+1,L) + M r_2(k+1,L)
\end{aligned}
$$

This allows the computation of successive values of the fast signature with 3 additions, 2 subtractions, 1 multiplication and a shift, assuming that M is a power of two[10].

3.2.6 The signature search algorithm

Once A has received the list of signatures from B, it must search a_i for any blocks at any offset that match the signatures of some block from B. The basic strategy is to compute the fast signature for each block starting at each byte of a_i in turn, and for each signature, search the list for a match.

The search algorithm is very important to the efficiency of the rsync algorithm and also affects the scalability of the algorithm for large file sizes. The basic search strategy used by my implementation is shown in Figure 3.1.

The first step in the algorithm is to sort the received signatures by a 16 bit hash of

[9]This form of the fast signature for rsync was first suggested by Paul Mackerras. It is based on ideas from the Adler checksum as used in zlib[Gailly and Adler 1998]. It also bears some similarity to the hash used in the Karp-Rabin string matching algorithm[Karp and Rabin 1987].

[10]Some alternative fast signature algorithms are discussed in the next chapter.

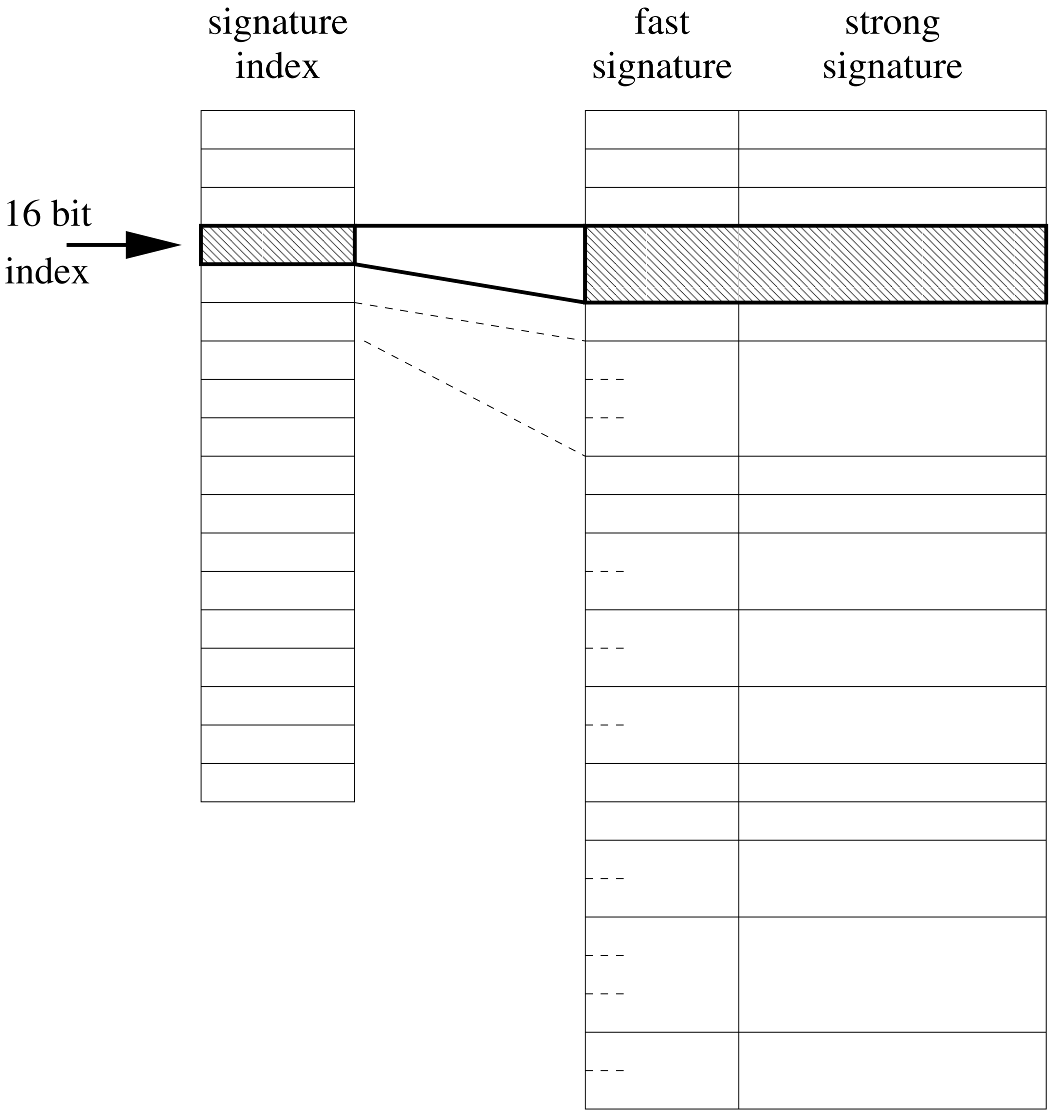

Figure 3.1: The signature search algorithm

the fast signature[11]. A 16 bit index table is then formed which takes a 16 bit hash value and gives an index into the sorted signature table which points to the first entry in the table which has a matching hash.

Once the sorted signature table and the index table have been formed the signature search process can begin. For each byte offset in a_i the fast signature is computed, along with the 16 bit hash of the fast signature. The 16 bit hash is then used to lookup the signature index, giving the index in the signature table of the first fast signature with that hash.

A linear search is then performed through the signature table, stopping when an entry is found with a 16 bit hash which doesn't match. For each entry the current 32 bit fast signature is compared to the entry in the signature table, and if that matches then the full 128 bit strong signature is computed at the current byte offset and compared to the strong signature in the signature table.

If the strong signature is found to match then A emits a token telling B that a match was found and which block in b_i was matched[12]. The search then continues at the byte after the matching block.

If no matching signature is found in the signature table then a single byte literal is emitted and the search continues at the next byte[13].

At first glance this search algorithm appears to be $O(n^2)$ in the file size, because for a fixed block size the number of blocks with matching 16 bit hashes will rise linearly with the size of the file. This turns out not to be a problem because the optimal block size also rises with n. This is discussed further in Section 3.3[14], for now it is sufficient to know that the average number of blocks per 16 bit hash is around 1 for file sizes up to around 2^{32}.

For file sizes much beyond that an alternative signature search strategy would be needed to prevent excessive cost in the signature matching. For example the signatures could be arranged in a tree giving $\log n$ average lookup cost at the expense of

[11]The hash algorithm used is the 16 bit sum of the two 16 bit halves of the fast signature.

[12]These tokens are very amenable to simple run length encoding techniques. This is discussed further in the section dealing with stream compression of the rsync output.

[13]In fact, literal bytes are buffered and only emitted when a threshold is reached or a match is found. The literal bytes may also be compressed. This is discussed more in the next chapter.

[14]It rises as $\sqrt{n}$ or n depending on the definition of "optimal".

some extra memory and bookkeeping.

3.2.7 Reconstructing the file

One of the simplest parts of the rsync algorithm is reconstructing the file on B. After sending the signatures to A, B receives a stream of literal bytes from A interspersed with tokens representing block matches. To reconstruct the file B just needs to sequentially write the literal bytes to the file and write blocks obtained from the old file when a token is received.

Note this that process assumes that

- B has fast random access to the old file; and
- the reconstruction is not performed in place.

The fast random access requirement is not a problem for most applications of rsync, as that is the natural access method for files. In practice the performance loss due to random access isn't a problem as in most cases sequential matching blocks will be adjacent in the old file unless the file has undergone a considerable rearrangement. In either case, the speedups due to the reduced network traffic from rsync will generally far outweigh the cost of random file access.

The requirement that the reconstruction is not done in place is usually not a problem except when disk space is very tight. Even without this restriction it would, in many cases, be wise for the reconstruction to happen to a temporary file followed by an atomic rename as otherwise the transfer would be susceptible to network interruptions leaving a corrupted, half updated file[15].

3.3 Choosing the block size

The primary tuning factor in rsync is the block size, L. The choice is governed by the following factors:

[15] Many people use rsync to synchronize application, binary and database files where a partly updated file would have disastrous consequences for the integrity and stability of the system.

- The block size must be larger than the combined sizes of the fast and strong signatures or rsync will provide no benefit in reducing the total bytes transmitted over the wire[16].
- A larger block size will reduce the size of the signature information sent from B to A.
- A smaller block-size is likely to allow A to match more bytes thus reducing the number of bytes transmitted from A to B.

To work out the optimal block size we must make some assumptions about the number and distribution of differences between the old and new files.

If, for example, we assume that the two files are the same except for Q sequences of bytes, with each sequence smaller than the block size and separated by more than the block size from the next sequence then the total number of bytes transmitted will be approximately

$$t(L) = (s_f + s_s)n/L + QL + s_t(n/L - Q) + O(1)$$

where s_f is the byte size of the fast signature, s_s is the byte size of the strong signature and s_t is the byte size of the block match token. If we assume that s_f is 4, s_s is 16 and s_t is 4 then the non-constant parts become

$$t(L) = 24n/L + Q(L-4)$$

in which case the optimal value for L is $\sqrt{24n/Q}$. This means that for common file sizes of a few kilobytes to a few megabytes and with about a dozen differences in the two files the optimal block size is between a hundred and a few thousand bytes.

3.3.1 Worst case overhead

Another way of choosing the block size is to consider the worst case overhead that rsync can impose as compared to the traditional "send the whole file" methods. This

[16]Although it will change the proportion of bytes sent in either direction, which may be beneficial in some circumstances.

is an important consideration when deciding whether rsync should be used for a particular application.

Clearly the worst case for rsync is where there are no common blocks of data between the old and new files. In that case the overhead will be the cost of sending the signature information from B to A. The information sent from A to B will be a single block of literal bytes and will be the same size (apart from a few control bytes) as what would be sent by traditional file transmission algorithms.

If the acceptable overhead is expressed as a percentage p of the total file size then the minimum block size that could be used is

$$L(p) = 100(s_f + s_s)/p$$

so for an acceptable overhead of 1% and with $s_f = 4$ and $s_s = 16$ we would need to use a block size of at least 2000 bytes[17].

3.4 The probability of failure

The use of a signature smaller than the block size means that the rsync algorithm must have a finite probability of failure, where failure is defined as B constructing an incorrect copy of the new file. It is interesting to investigate what this probability is.

3.4.1 The worst case

The first probability of interest is the worst case probability of failure assuming that the fast signature is completely ineffective (assume that it is of zero effective bit strength). In that case A will need to compare approximately n strong signatures (one for each byte offset) with each of the n/L strong signatures from B. The algorithm fails if any of these comparisons give a false positive. If we assume that L is $O(\sqrt{n})$ then the number of pairwise strong signature comparisons per file is $O(n^{3/2})$.

To convert this to a probability of failure we need to make an important assumption about the nature of the rsync algorithm – that rsync is no better at finding collisions in the strong signature algorithm than a brute force search. This is not an unrea-

[17]Optimizations considered in the next chapter reduce the effective size of the signatures, allowing for a considerably smaller block-size while maintaining the same maximum overhead target.

sonable assumption as otherwise rsync could be employed as a code breaking engine, which would indicate a serious flaw in the cryptographic strength of the strong signature algorithm. Although cryptanalysts do regularly find flaws in message digest algorithms it seems implausible that rsync happens to have exposed such a flaw.

Given that assumption we can calculate the probability of failure under specific conditions. With a 128 bit strong signature (such as MD4) we would need about 2^{127} signature comparisons to have an even chance of finding a false positive. If we have one million computers each transferring a one gigabyte file each second then we would expect it to take about 10^{11} years[18] for a transfer failure to occur.

3.4.2 Adding a file signature

Even so, it would be desirable for the probability of failure of the rsync algorithm to scale not as $n^{3/2}$ but instead to be independent of the file size. Luckily this is easy to achieve.

The change that is needed is the addition of another strong signature which applies to the whole file, rather than just to the individual blocks of the file. The algorithm needs to change in the following way:

- When working through the file to find the matches, A incrementally computes the strong signature of the whole file. This is then sent to B after the matching is complete.
- When reconstructing the file, B incrementally computes the strong signature on the whole file.
- If the file signature sent from A doesn't match the reconstructed file signature then the whole algorithm is repeated, but with a new signature s_s' for each block.

The algorithm loops until the file signatures match[19].

The use of a different strong signature for the block signatures each time through the loop is necessary to prevent the same false positive signature match happening

[18]The universe is thought to be about 10^{10} years old.

[19]When the transfer is repeated B uses the result of the incorrect transfer as the starting point for the new transfer. This is an advantage as it is highly unlikely that more than one of the blocks was transferred incorrectly, so the second transfer will be very fast.

again. In my implementation I use the simple solution of using the same signature code but seeding it with a different value each time through the loop. The seed is applied by computing the signature algorithm on an extended block obtained by concatenating the original block with a 4 byte timestamp.

With this file signature in place the probability of failure changes to $O(2^{-128})$ for each file[20] which means that for the previous example of one million computers transferring one file per second it will take about 10^{25} years on average for a failed transfer to occur. That is quite a comfortable margin[21].

3.5 Practical performance

The real test of any algorithm is its performance on real world data. In the case of rsync that means the amount of network traffic required to update a set of modified files.

Choosing appropriate data sets for testing purposes is a difficult task. I wanted data sets which were readily available for download on the Internet to facilitate the testing of alternative algorithms by other researchers. I also wanted data sets for which rsync might reasonably be used. The data sets finally chosen were:

- The Linux kernel source code as distributed by Linus Torvalds. An enormous number of people download this source code at each release either by downloading the whole source tree or patches from previous versions. I chose versions 2.0.9 and 2.0.10. These versions were chosen because they have the least number of changes of any of the 2.0 versions, as measured by the compressed diff files used to distribute kernel source changes.

- The Samba source code as distributed at *http://ftp.samba.org/*. This was chosen largely because it provided the original motivation for developing rsync. I chose versions 1.9.18p10 and 2.0.0beta1 as these are two adjacent releases which are

[20] Note that the "birthday paradox" does not apply to rsync because when transferring N files each file is compared only to one other file, rather than to the $N-1$ other files.

[21] It is interesting to note that the probability of failure is now independent of the strength of the signature used on each block and depends only on the strength of the file signature. In the next chapter this is used to safely reduce the size of the signature bytes to increase the network efficiency of the algorithm.

of particular interest from the point of view of rsync because the source tree was substantially rearranged between the two releases. This means there are a large number of changes, but many of them are data movements rather than insertions or deletions[22].

- The binary distribution of Netscape Communicator professional version 4.06 to version 4.07, as distributed by Netscape in the original compressed .exe format.

3.5.1 Choosing the format

A problem with the first two files was choosing the appropriate format to process the files in. These files are normally distributed as compressed tar files but both the compression and the tar format lead to problems in making reasonable comparisons between algorithms.

The compression algorithm (usually gzip) has the property that a small change in the uncompressed data may cause a change in all of the compressed file beyond that point. This can be dealt with using the techniques discussed in the next chapter but it unnecessarily complicates the discussion of the initial rsync algorithm and would render comparison with text-based differencing algorithms impossible. I decided to use these archives in uncompressed form.

The second problem is the tar format. The tar files as distributed for Linux (and to a lesser extent for Samba) have varying dates for a large number of files that have the same content in both versions. In the Linux case these header changes dominate the differences between the two versions. The obvious (and most commonly used) solution is to run the tools with a *recursion into directories* option so that only files containing differences are actually processed. While both my rsync implementation and the GNU diff tools have this option, it does present some significant problems in analyzing the results. In particular the appropriate block size for rsync is quite different for each of the files in the archive, which makes a simple analysis of the results difficult.

To overcome this problem I decided to transform the files into a format that is more amenable to analysis. This was done by concatenating all files in the Linux and Samba

[22] The commonly used "diff" algorithm cannot encode data movements, whereas rsync can.

name	**size A**	**size B**	**diff**
Linux	22577611	22578079	4566
Samba	9018170	5849994	15532295
Netscape	17742163	17738110	n/a

Table 3.1: Sizes of the test data sets

archives into single files. This produces a file containing all of the data changes of the original but without any of the complications of the tar format[23].

Table 3.1 shows the size of the two versions of each of the three sets of data. The column marked "diff" is the size of a GNU diff file taken between the two versions. As diff cannot operate correctly on binary files, diff sizes are not shown for the Netscape data sets.

One interesting thing to note is the enormous size of the diff on the Samba data set, larger than the sum of the two versions of the file. The size reflects the fact that diff encodes only insertions and deletions, so the source code rearrangement made between these two versions cannot be efficiently coded by diff.

3.5.2 Speedup

The *speedup* is the ratio of the size of the source file to the total number of bytes transmitted. The bytes transmitted are made up of the signatures, the literal data, the match tokens and a small number of protocol overhead bytes. The speedup peaks at the best tradeoff point between the size of the signatures and the size of the literal data[24].

Table 3.2 shows the performance of rsync for four different block sizes on the Linux data set. The block sizes were chosen to span the observed optimal range for this data set. The amount of literal data sent increases roughly in proportion to the block size. This is expected for non-overlapping changes in a data set. The matched data decreases by the same amount that the literal data increases, as the sum of literal and matched data equals the size of the source file.

For the Linux data set there are a total of 20 changes between the two files[25]. When

[23]rsync actually works very well on tar files, but GNU diff doesn't.

[24]Note that the speedup figure does not take into account time spent in computation. In practice the computational cost of the algorithm is only a tiny fraction of the total time cost unless an unusually slow CPU or fast network is used.

[25]As reported by diff.

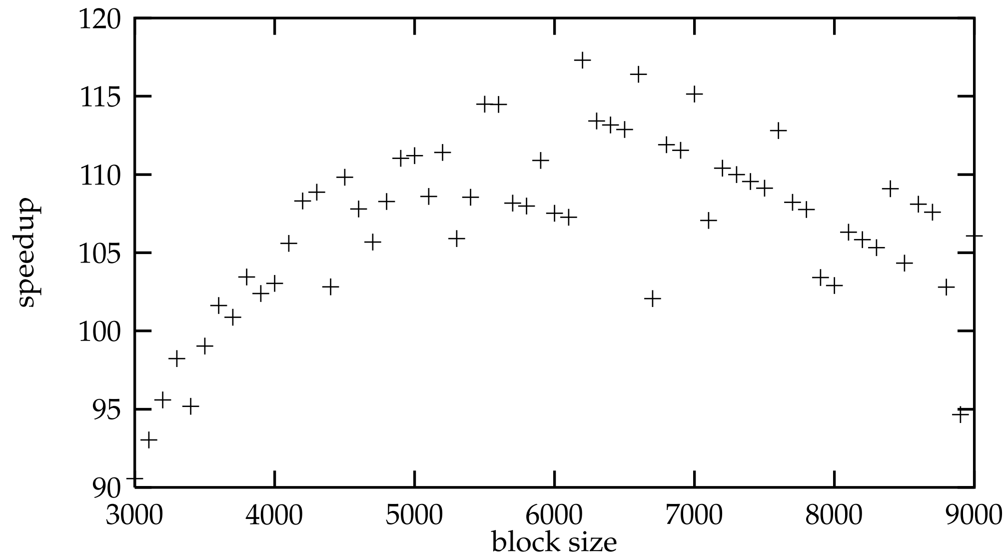

Figure 3.2: Speedup on the Linux data set with varying block size

this number is used in the approximate formula given in Section 3.3 we find that the predicted optimal block size is 5205, which is in rough agreement with the observed best size.

A more accurate picture of the best block size for the Linux data set is given in Figure 3.2 which shows the speedup for block sizes ranging from 3000 to 9000 in increments of 100. The ragged nature of the graph reflects the fact that some of the differences lie within the same block for some block sizes, leading to a higher speedup than is predicted by the very simple formula given earlier[26].

It is interesting to note that the amount of literal data transferred in the case which gives the best speedup is more than an order of magnitude larger than the difference file produced by GNU diff for the same data. The reason for this is that the total data transferred is dominated by the signature information, which is 20 bytes per block. Even with a block size as large as 5000 this signature information amounts to 20 times the size of the diff file. In the next chapter some methods of reducing the size of this signature information are investigated.

[26] Note the compact scale on the y-axis of the graph. The speedup is in fact not very sensitive to block size, which is fortunate as when used in real applicatons a user would have no way of knowing in advance the precise shape of the graph for a given data set, and thus no way of choosing the optimal blocksize beyond the rough formula that has been given.

block size	**literal data**	**matched data**	**speedup**
3000	68532	22509079	90.57
4000	83532	22494079	103.04
5000	94532	22483079	111.20
6000	119532	22458079	107.52

Table 3.2: rsync performance with Linux data set

block size	**literal data**	**matched data**	**speedup**
100	4758070	4260100	1.48
200	5093570	3924600	1.56
300	5281070	3737100	1.57
400	5444570	3573600	1.56

Table 3.3: rsync performance with Samba data set

Table 3.3 shows the speedup for the Samba data set. The block size range is again chosen to center on the best observed block sizes.

The speedups for the Samba data set are much lower than those observed for the Linux data set, primarily because the new file contains so much data that is not present in the old file as is shown by the fact that the new file is around 3.1 MB larger than the old file.

In this case the number of changes between the two versions as reported by diff is about 29000, which according to the simple formula derived earlier would give an optimal block size of approximately 86. The actual optimal block size is much larger, reflecting the fact that many of the changes are in fact rearrangements rather than insertions or deletions. The large number of changes also means that the assumption in the derivation of that formula of non-overlapping changes smaller than the block size is completely incorrect.

The speedup shown for the Samba data set is smaller than would be achieved by using conventional compression techniques. In the next chapter optimizations and the addition of compression techniques in rsync lead to speedups far above those achievable with conventional compression.

Table 3.4 shows the speedup of rsync for the Netscape data set. The interesting thing about this data set is that it consists of already-compressed data. The large

block size	**literal data**	**matched data**	**speedup**
500	7422663	10319500	2.16
1000	7501163	10241000	2.25
1500	7562110	10176000	2.27
2000	7610163	10132000	2.27

Table 3.4: rsync performance with Netscape data set

data set	**block size**	**tag hits**	**false alarms**	**matches**
Linux	5000	10835	0	4497
Samba	300	1358701	179	12457
Netscape	1500	1256800	159	6784

Table 3.5: Fast signature performance

amount of common data found by rsync in the two versions indicates that the two files share large portions of unchanged data, despite the compressed format.

3.5.3 Signature performance

Another important measure of the performance of rsync is the number of times that the strong signature needs to be computed. This gives an indication of whether the fast signature is doing its job of reducing the computational cost of the algorithm.

Table 3.5 shows the number of tag hits, false alarms and block matches for the entries from the previous tables with the best speedup values.

A tag hit is defined as an offset in the new file where the 16 bit hash of the fast signature matches any of the 16 bit hashes of the fast signatures from the old file. When a tag hit occurs A needs to check all matching 32 bit fast signatures from the old file for a match. The number of tag hits is an indication of the effective bit strength of the 16 bit hash of the fast signature.

A false alarm occurs when a 32 bit match is made between a fast signature in the new and old files but the strong signature indicates that the indicated blocks do not in fact match. The ratio of false alarms to actual matches indicates the effectiveness of the fast signature algorithm at reducing the computational cost of the strong signature.

We can perform some simple calculations on the number of tag hits and false alarms to find an estimate for the effective bit strength of the 32 bit fast signature

data set	**16 bit strength**	**32 bit strength**
Linux	16	32
Samba	16	30
Netscape	16	29

Table 3.6: Fast signature effective bit strength

and the 16 bit fast signature hash. The fast signature is calculated once for each byte of literal data plus once per block and each fast signature is compared to the fast signatures of each block from B so the number of pairwise comparisons of fast signatures is $(d_l + n/L)n/L$ where d_l is the number of bytes of literal data transferred. If a signature algorithm has an effective bit strength of b then we would expect it to incorrectly match approximately every 2^b times, which means the effective bit strength is

$$b = \log\left(\frac{(d_l + n/L)n}{Lm}\right)$$

where m is the observed number of incorrect matches. Table 3.6 shows the effective bit strengths for the three sets of results given in the previous table[27].

In all three test sets the 16 bit effective strength is the full 16 bits, which demonstrates that the 16 bit hash is doing the job it was designed for. The performance of the 32 bit fast signature is not quite as good, but still does quite well considering the simplicity of the algorithm.

3.6 Related work

While I haven't found any papers describing algorithms like the one described in this chapter, it has been pointed out to me that US patent number 5446888 describes a somewhat similar algorithm. The algorithms are not identical, but do seem to provide similar functionality[Pyne 1995].

[27] With zero false alarms for the Linux data set a "perfect" effective 32 bit strength is indicated.

3.7 Summary

The rsync algorithm provides an effective way to solve the remote update problem using a randomized method with a very low probability of failure. The dual signature method used by rsync allows the algorithm to efficiently find matches at any byte boundary in the source file while still using a strong signature for block matching. The next chapter will consider various enhancements and optimisations to the basic rsync algorithm.

Chapter 4

rsync enhancements and optimizations

The previous chapter introduced the basic rsync algorithm which allows for efficient remote update of data over a high latency, low bandwidth link.

This chapter will consider optimizations and extensions to the algorithm, practical issues to do with mirroring large filesystems and transformations that make rsync applicable to a wider range of data.

4.1 Smaller signatures

In Section 3.4.2 an overall file signature was added to the algorithm in order to make the probability of failure independent of the file size. This adds very little network overhead to the algorithm[1] but it does add a lot of redundancy. The algorithm is no longer dependent on the bit strength of the strong block signatures in order to produce the correct result.

A consequence of this change is that we can afford to use a much weaker (and thus smaller) block signature algorithm. Previously we used a 16 byte MD4 block signature but as the aim now is to make the probability of a second pass small rather than preventing incorrect transfers a much smaller block signature is warranted. The question is how small.

To work this out we need to find the probability of a second pass being required, given the file size and effective bit strengths of the fast and strong signatures. This probability will be approximately the number of pairwise signature comparisons per-

[1] It adds 16 bytes per file assuming that a signature mismatch isn't detected.

bit strength	file sizes
8	0 to 2MB
16	2MB to 64MB
24	64MB to 2GB
32	2GB to 64GB
40	64GB to 4TB

Table 4.1: Required strong signature strengths with p=0.01

formed multiplied by the probability of a single pairwise comparison randomly giving an incorrect match

$$p = 2^{-(\beta_s+\beta_f)}\frac{n^2}{L}$$

where β_s and β_f are the effective bit strengths of the strong and fast signatures respectively[2].

This form for p clearly implies that $\beta_s + \beta_f$ should be scaled in proportion to $\log n$ (for large n) in order to maintain a constant probability of a second pass being needed.

In particular the required bit strength of the combined fast and strong signatures needs to be

$$b = \log\left(\frac{n^2}{Lp}\right)$$

In practice the strong signature needs to be a multiple of eight to avoid messy bit shifting in the implementation. If we use $L = \sqrt{n}$ and $p = 0.01$[3] with an effective bit strength value for the fast signature of 29 bits[4] then the approximate sizes needed for the strong signature algorithm are given in Table 4.1.

4.2 Better fast signature algorithms

The 32 bit fast signature algorithm described in the previous chapter works quite well, but it is not the best possible fast signature algorithm. Any algorithm with the *rolling* property – the property that it can be quickly updated when the buffer is shifted by

[2]This is actually an overestimate of the probability except in the case where the two files really do share no common blocks because when a block match is made the algorithm skips the next L bytes. A more accurate estimate would use the number of literal bytes that will be transferred, but that is unavailable before the start of the transfer.

[3]The choice of p is arbitrary.

[4]See Table 3.6.

one byte – can be used. The choice of fast signature algorithm is a tradeoff between speed and effective bit strength.

In [Taylor et al. 1997] Richard Taylor examines the fast signature algorithm used in rsync and compares it to a number of alternative algorithms with the rolling property with the aim of finding a better fast signature algorithm. The algorithms he proposes are:

$$
\begin{aligned}
C_1(k) &= \sum_{j=0}^{L-1} 2^{L-(j+1)} a_{j+k} \bmod [2^{16}-1] \\
C_2(k) &= \sum_{j=0}^{L-1} 8^{L-(j+1)} a_{j+k} \bmod [2^{16}-1] \\
C_3(k) &= \sum_{j=0}^{L-1} 32^{L-(j+1)} a_{j+k} \bmod [2^{16}-1] \\
C_4(k) &= \sum_{j=0}^{L-1} 128^{L-(j+1)} a_{j+k} \bmod [2^{16}-1] \\
D_1(k) &= \sum_{j=0}^{L-1} 3^{L-(j+1)} a_{j+k} \bmod [2^{16}-1] \\
D_2(k) &= \sum_{j=0}^{L-1} 5^{L-(j+1)} a_{j+k} \bmod [2^{16}-3] \\
D_3(k) &= \sum_{j=0}^{L-1} 7^{L-(j+1)} a_{j+k} \bmod [2^{16}-5] \\
D_4(k) &= \sum_{j=0}^{L-1} 17^{L-(j+1)} a_{j+k} \bmod [2^{16}-7]
\end{aligned}
$$

These signatures all have the *rolling* property with the following recurrence relations[5]:

$$
\begin{aligned}
C_1(k+1) &= (C_1(k) << 1) + a_{k+L} - a_k \bmod [2^{16}-1] \\
C_2(k+1) &= (C_2(k) << 3) + a_{k+L} - a_k \bmod [2^{16}-1] \\
C_3(k+1) &= (C_3(k) << 5) + a_{k+L} - a_k \bmod [2^{16}-1] \\
C_4(k+1) &= (C_4(k) << 7) + a_{k+L} - a_k \bmod [2^{16}-1] \\
D_1(k+1) &= 3D_1(k) + a_{k+L} - 3^L a_k \bmod [2^{16}-1] \\
D_2(k+1) &= 5D_2(k) + a_{k+L} - 5^L a_k \bmod [2^{16}-3]
\end{aligned}
$$

[5] The << notation is used for left shift, as in C.

Algorithm	strength
T—U	27.1
C1—C2	30.4
C1—C3	28.0
C1—C4	30.4
C2—C3	30.4
C2—C4	28.1
C3—C4	30.4
D1—D2	32.0
D1—D3	32.1
D1—D4	31.9
D2—D3	32.0
D2—D4	31.8
D3—D4	32.1

Table 4.2: Effective bit strengths for Taylor's checksum algorithms

$$D_3(k+1) = 7D_3(k) + a_{k+L} - 7^L a_k \bmod [2^{16} - 5]$$
$$D_4(k+1) = 17D_4(k) + a_{k+L} - 17^L a_k \bmod [2^{16} - 7]$$

which allows the signature to be calculated with one multiplication and a small number of elementary operations (shift, add, subtract).

The effective bit strengths as calculated in the paper are shown in Table 4.2. The algorithm marked as T—U is the same as that used in the previous chapter. Taylor uses a somewhat different method of calculating effective bit strengths based on random data, which results in a more pessimistic view of the T—U algorithm than was shown in Table 3.6. These results clearly show that a fast signature algorithm can have an effective bit strength close to the optimal theoretical value.

4.2.1 Run-length encoding of the block match tokens

Until now I have assumed that the block match tokens are sent individually with each token match containing information about an individual block in B that has been matched to a portion of A[6].

When A and B are quite similar these block match tokens consume a considerable

[6] The offset in A does not need to be sent because it is implied given the current position in the file reconstruction process.

data set	block size	without RLE	with RLE
Linux	5000	161	185
Samba	300	1.65	1.67
Netscape	1500	2.31	2.32

Table 4.3: Speedup with and without run-length encoding of match tokens

proportion of the total number of bytes sent over the link so it is worthwhile reducing this cost if possible. This can be done quite easily by observing that block match tokens will come in runs of adjacent blocks when the matching ranges of the two files are larger than twice the block size. In practice this is very common.

To encode these tokens A sends the difference between adjacent tokens to B rather than the tokens themselves and run-length encodes these token differences. In many cases the difference is one which allows for efficient run-length encoding.

Table 4.3 shows the improvement in the speedup factor with the addition of run-length encoding of the block match tokens. As expected, the improvement is most obvious for the Linux data set where literal data makes only a small part of the total network cost.

4.3 Stream compression

Thus far I have ignored the compressibility of the literal data and the token matching stream in rsync. While it is fairly clear that employing some sort of stream compression on the literal data sent from A to B is possible, it is more interesting to look at ways to do better than just employing a traditional stream compression algorithm.

The problem with using a normal stream compression algorithm on the literal data is that when only a small number of widely separated literal blocks are sent the compressor will usually achieve a lower compression ratio than would be achieved in compressing the entire file as it will be deprived of the surrounding data for each block. Stream compression algorithms generally rely on finding localized redundancy and building adaptive compression tables, which is why they generally perform worse on small chunks of data.

To improve upon stream compression of the literal data we need to take advantage

data set	uncompressed	**stream compression**	**enhanced compression**
Linux	94532	32406	30589
Samba	5281070	1160349	1152019

Table 4.4: Literal data sizes with stream and enhanced compression

of some shared knowledge between A and B. That knowledge takes the form of the blocks which the rsync algorithm has already determined to be present at both ends of the link. This knowledge can be used to increase the compression ratio achieved for the literal data by giving the compression algorithm a greater amount of history data to work with, without sending that history data over the link.

To achieve this, the compression algorithm used needs to be modified to include a "process but don't emit" operation which updates the internal tables used in the compressor as though the data had been compressed but doesn't emit the resulting compressed data for that range of bytes. The token sending operation from A then uses this operation to update the compression tables for data blocks that are found to be in common in the old and new files. The result is that the literal data is compressed with approximately the same compression ratio that would be achieved with the same compression algorithm used on the whole file.

To test this idea the deflate[Gailly and Adler 1998] compression algorithm as used by gzip was used, both with and without the "process but don't emit" code[7]. Table 4.4 shows the results for the Linux and Samba data sets[8] with the same block sizes as used in Section 3.5.3.

Although the enhanced compression algorithm does give a noticeable improvement over the use of normal stream compression the effect is not large, just under 1% for the Samba data set and less than 6% for the Linux data set.

[7]It was fortuitous that the PPP deflate driver written by Paul Mackerras has a "process but don't emit" mode for reasons quite different from those required by rsync. This code was reused in rsync.

[8]The Netscape data set was omitted as the file is already compressed, making additional compression of the literal data pointless.

4.4 Data transformations

One of the weaknesses of the rsync algorithm is that there are some cases where seemingly small modifications do not result in common blocks between the old and new files which can be matched by the algorithm. The two situations where this is particularly common are compressed files and files where systematic global editing has been done. In both cases the rsync algorithm can be modified to include a data transformation stage to allow these files to be updated efficiently.

4.4.1 Compressed files

Imagine a situation where a server stores text files (such as source code or documents) in a compressed form, using a common compression algorithm such as gzip, bzip or zip. A property of these compression algorithms is that any change in one part of the uncompressed file results in changes throughout the rest of the compressed file. This means that if a change is made towards the start of the uncompressed file and the new file is stored in a compressed format then an rsync update of the compressed file to a machine containing the old compressed file will end up sending almost all of the file.

A solution to this problem is to uncompress the files at both ends before the rsync algorithm is applied, and then re-compress the files after the transfer is complete. This simple data transformation allows the efficient update of compressed files.

This solution isn't ideal, however. The problem is that many compression formats are not symmetric, so that when a file is uncompressed then re-compressed you do not necessarily end up with identical files. This is particularly the case when different versions of the compression utility are used. Even if the data formats are compatible there is no guarantee that the compressed data will be identical.

In some cases this isn't a problem as it is the uncompressed content of the compressed file that is important, rather than the compressed file itself, but there are important situations where the exact bytes in the compressed file must be preserved. One such case is where a site is distributing software[9] that has been digitally signed by the authors to ensure that no tampering is possible. It is common practice to dis-

[9] One of the most common uses for rsync is as an anonymous rsync server for distributing and mirroring software archive sites on the Internet.

tribute PGP or MD5 fingerprints of the compressed archives. Unless the exact contents of the compressed archives are preserved these fingerprints become useless.

Although it would be possible to distribute fingerprints of the uncompressed files rather than the compressed archive this may be inconvenient for users who wish to confirm the contents of downloaded archives before they are unpacked. This is especially true if a self-extracting compression format is used as any malicious content in the file could be executed during the uncompression process.

4.4.2 Compression resync

A different solution to using rsync with compressed files which overcomes these problems is to use file compression algorithms which do not propagate changes throughout the rest of the compressed file when changes are made. Unfortunately most commonly used compression algorithms use adaptive coding techniques which result in a complete change in the compressed file after the first change in the uncompressed file.

It is, however, quite easy to modify almost any existing compression algorithm to limit the distance that changes propagate without greatly reducing the compression ratio. If the designers of compression algorithms were to include such a modification in future algorithms then this would significantly enhance the utility of the rsync algorithm.

The modification is quite simple[10]:

1. A fast rolling signature is computed for a small window around the current point in the uncompressed file;

2. stream compression progresses as usual;

3. when the rolling signature equals a pre-determined value the compression tables are reset and a token is emitted indicating the start of a new compression region.

[10] Although I have found no reference to this compression modification idea in the literature I would be surprised if something similar has not been proposed before.

This works because the compression will be "synchronized" as soon as the rolling signature matches the pre-determined value. Any changes in the uncompressed file will propagate an average of $2^b/c$ bytes through the compressed file, where b is the effective bit strength of the fast rolling signature algorithm and c is the compression ratio.

The value for b can be chosen as a tradeoff between the propagation distance of changes in the compressed file and the cost in terms of reduced compression of resetting the internal tables in the compression algorithm. For compression algorithms designed to be used for the distribution of files which are many megabytes in size a propagation distance of a few kilobytes would be appropriate[11]. In that case the first few bits of the fast signature algorithm used in rsync could be used to provide a weak fast signature algorithm.

4.4.3 Text transformation

The second common case where rsync will not work efficiently is where simple global editing has been applied to a file so that no common blocks remain between the old and new files. Examples of this sort of editing include conversion between text formats used on different operating systems or reformatting of text to reflect adjusted margins or layout rules.

The case of text format conversion is particularly common. Operating systems such as MSDOS store text files using a pair of characters (a carriage return and a linefeed) to indicate the end of a line. Other operating systems such as Unix store files with a single end-of-line character (a linefeed). Once a file has been converted between these two text formats the rsync algorithm would not be able to find any common blocks between the old and new files unless a very small block size were used, and would thus not result in a speedup over transferring the whole file.

The solution to this problem is to use a reversible data transformation which conditions the modifications (in this case the change of end of line marker) so that they are more closely clustered in the file. One such transformation is the BWT transform[Burrows and Wheeler 1994] which is more commonly used as part of block sorting

[11] This would also give a degree of fault tolerance in the compressed archives.

transformation	speedup
none	0.99
BWT	27.1

Table 4.5: The effect of a BWT transformation on *Middlemarch*

compression algorithms.

The BWT transformation sorts each character in a file as indexed by all the other characters in a file viewed as a circular buffer. This means that groups of characters that share a common context in the file will be grouped together in the transformed file. The transformation is reversible[12] and only increases the size of the file by a fixed small amount, typically four bytes. In a file stored in MSDOS text format the carriage return characters will end up all grouped together as they all share a common context – they are immediately next to a linefeed character.

To take advantage of this transformation all that needs to be done is to apply the BWT transform to the files on both A and B then to apply the normal rsync algorithm to transfer the transformed file. After the transfer is complete the BWT transform can be reversed on B to obtain the original file from A. As the BWT transform is completely reversible this system does not suffer from the problems that are inherent in uncompressing/recompressing compressed files.

To demonstrate the effectiveness of this transform on text files an electronic copy of the book *Middlemarch* by George Eliot as distributed by Project Gutenberg[GUTENBERG 1998] was tested with and without a BWT transformation when the book was changed from the distributed MSDOS text format to Unix text format. The results are shown in Table 4.5 for a block size of 700.

The dramatic speedup afforded by the transformation is an indication of the usefulness of reversible transformations in rsync. It also seems likely that other types of transformations might be found which are appropriate for a variety of regular editing operations.

[12]The reversibility of the BWT transform is truly amazing, it is one of the most elegant algorithms I have ever encountered.

4.5 Multiple files and pipelining

The speedup in terms of the number of bytes sent over the network link is only one aspect of an efficient file transfer algorithm. The other really important consideration is the latency costs involved, particularly when the transfer (or update) consists of many files rather than a single file.

Commonly used file transfer protocols such as ftp, rcp and rdist send each file as a separate operation, waiting for the receiver to acknowledge the completion of the transfer of the current file before starting to transfer the next file. This means that when transferring N files the minimum time will be $N\lambda$ where λ is the round-trip latency of the network link.

When used for a large number of small files (such as mirroring a typical web site) over an international Internet link this latency cost may dominate the time taken for the transfer. To overcome the latency cost the file transfer algorithm needs to use one logical round-trip for multiple files, rather than one per file[13]. This is commonly known as pipelining.

The pipelining method used by rsync is shown in Figure 4.1. The process is broken into three pieces – the generator, the sender and the receiver. The generator runs on B and generates the signatures for all files that are to be transferred, sending the signatures to the sender. The sender (running on A) matches the signatures from B against the new files and sends a stream of block tokens and literal bytes to the receiver. The receiver (running on B) reconstructs the files[14].

The link from the receiver to the generator (shown as a dashed line in the diagram) is for a small message to indicate if a file has not been successfully received (which is detected when the strong signature of the reconstructed file is not correct). In that case the generator starts again, but with a larger per-block strong signature[15].

To demonstrate the effectiveness of this scheme at reducing latency costs I trans-

[13] I use the term *logical round trip* to emphasize the fact that I am ignoring round-trips due to the transport layer, such as TCP acknowledgement packets. Although TCP guarantees that a return acknowledgement packet will be sent at least every second packet these usually do not contribute to latency due to the way the TCP windowing works.

[14] Note that I have ignored the transfer of the file list, which must be done prior to starting the 3-way process. This adds a total of one more latency to the algorithm.

[15] It will also start with the incorrectly transferred file, as it is highly likely that most of the blocks are in fact correct.

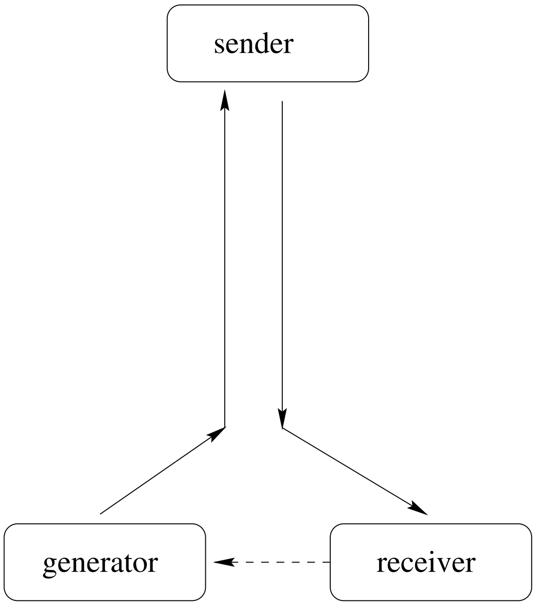

Figure 4.1: Pipelining in rsync

protocol	time (seconds)
rsync	6.9
rcp	351
ftp	853

Table 4.6: Time to transfer 1000 small files over a dialup link

ferred 1000 files of size one byte each using rsync, rcp and ftp[16]. The files were transferred over a PPP link which has a latency of approximately 120 milliseconds and a maximum bandwidth of approximately 3.6 kilobytes/sec. The results are shown in Table 4.6.

The dramatic effect of the per-file latency in rcp and ftp is clearly shown in the results. In real application (where files larger than one byte are usually used!) the observed differences will be the additional overhead when transferring 1000 files. For very large files this will be unimportant but it can have a very significant impact for moderately large files or when using a high latency network link.

[16]For ftp the *mget* operation was used with prompting disabled.

4.6 Transferring the file list

Before starting a multiple file transfer with the rsync algorithm it is useful to first transfer the list of files from the sending machine to the receiving machine. This allows the two computers to reference files using an index into the file list which simplifies the implementation considerably.

Although it is quite easy to pack the file list so that it consumes a minimum of space while being transferred[17] there are some quite common situations where the network costs associated with transferring the file list dominate the total network costs of the file transfer. This most commonly occurs when rsync is being used to *mirror* a directory tree consisting of many thousands of files where the number of files that have changed between mirror runs is small[18].

To reduce this cost it is possible to employ the rsync algorithm to more efficiently transfer the file list itself. This is done by generating the file list at both the senders and receivers end of the connection and storing it in a canonical (sorted) form. The receiver can then obtain a copy of the senders file list by running the rsync algorithm on the file list data itself.

By doing this the network cost of sending the file list is reduced dramatically when the sender and receiver have very similar directory trees. If their trees are not similar then the network cost of applying this technique is limited to the cost of sending the signatures for the data in the receivers file list.

4.7 Summary

This chapter has examined a number of optimizations and enhancements to the basic rsync algorithm described in the previous chapter. The use of smaller signatures, stream compression and data transformations improved the efficiency considerably for some data sets while the pipelining technique allows the algorithm to minimise latency costs normally associated with the transfer of large number of files. The next

[17]My rsync implementation uses approximately ten bytes per file in the file list, including meta data such as file permissions, file type and ownership information.

[18]By default my rsync implementation skips the transfer phase completely for files that are the same size and have the same timestamp on both ends of the link, thereby avoiding the network cost of sending the signatures but not avoiding the transfer of a file list entry.

chapter will consider alternative uses for the ideas behind rsync.

Chapter 5

Further applications for rsync

This chapter describes some alternative uses of the ideas behind rsync. Since starting work on rsync (and distributing a free implementation) the number of uses for the algorithm has grown enormously, well beyond the initial idea of efficient remote file update. From new types of compression systems to differencing algorithms to incremental backup systems the basic rsync algorithm has proved to be quite versatile.

This chapter describes some of the more interesting uses for the rsync algorithm.

5.1 The xdelta algorithm

The xdelta algorithm is a binary differencing algorithm developed by Josh MacDonald[MacDonald 1998]. It is based on the rsync algorithm but has been extensively modified to optimize for high speed and small emitted differences, taking advantage of the fact that both files being differenced are available locally, which allows the algorithm to ignore the cost of sending the signatures.

Binary differencing algorithms are extremely useful for transmitting differences over non-interactive links or broadcasting differences to many sites. A company which wishes to provide a more up-to-date binary for their large software product, for example, might distribute a binary patch which can be applied by users without requiring them to download the whole product a second time.

Using rsync as a binary differencing algorithm is quite simple. The fast and slow signatures are calculated as usual and the hash tables are formed in the same way that B does in the rsync algorithm, although the block size is set much smaller than would normally be set for rsync. The file being differenced is then searched for block matches in the same way that A searches for matches in rsync, with the literal data and token

name	diff size	diff time (seconds)	xdelta size	xdelta time (seconds)
Linux	4566	21.8	1148	10.5
Samba	15532295	130	883824	10.6
Netscape	n/a	n/a	7066917	25.4

Table 5.1: xdelta performance compared to GNU diff

matches stored in the "difference" file.

To reconstruct the new file using the difference file and the old file is the same as reconstructing the new file on *B* in the rsync algorithm, although it is wise to add additional checks in the form of file signatures to ensure that the file that the difference is being applied to is the same as the original file. Unlike the text based "diff" algorithm, xdelta cannot apply changes to files other than the original file.

As with rsync it is possible to further improve the algorithm by using stream compression on the generated difference file, possibly using the enhanced compression algorithm described in Section 4.3. The block match tokens in the difference file can also be run-length encoded in the same way that was described in Section 4.2.1.

Table 5.1 shows the performance of xdelta on the three test data sets from Chapter 3[1]. The results show a vast improvement in both the execution time and the resulting difference size between GNU diff and xdelta. The small size of the differences from xdelta is partly a result of delta being able to efficiently deal with the differences in the binary headers in the Linux and Samba tar files.

It is also interesting to note that xdelta produced a considerably smaller difference file than the number of bytes of literal data sent by the rsync algorithm in the results from Table 3.2, despite the fact that both xdelta and rsync use what is essentially the same algorithm. The difference is primarily a result of the very small block size used by xdelta, allowing for much finer grained matches. Using such a small block size when applying rsync over a network link would result in an excessive amount of signature data being sent from *B* to *A*.

[1] Version 1.0 of xdelta as distributed by Josh MacDonald was used for these tests.

5.2 The rzip compression algorithm

One of the more interesting adaptions of the rsync algorithm is as part of a compression algorithm. This section describes the rzip algorithm, a text compression algorithm inspired by rsync which works particularly well on large, highly redundant files. Although I developed the rzip algorithm myself, some similar work on rsync-inspired compression algorithms was done independently by Richard Taylor[Taylor 1998] and although we ended up with quite different compression algorithms we both rather confusingly called our algorithms "rzip".

The rsync algorithm has a lot in common with history based compression algorithms[2]. Both use a known set of data to efficiently encode another set of data using references into the known data set interspersed with literal data. In the case of rsync the known data is the old file (the one on B). In the case of history based compression algorithms it is the data in the file up to the point that is currently being compressed.

One of the problems with these compression algorithms is that they typically do not take good advantage of very widely spaced redundancies in the files being compressed. For example, if you take a large file (several megabytes) and produce a new file by concatenating the large file together with itself then commonly used history based compression algorithms won't produce a significantly better compression ratio on the new file than would be achieved with the original file. The reason is that compression algorithms typically have a limited *window* over which they operate, so they can only see redundancies within the window. They may carry over some information (such as Huffman encoding tables) when the window moves, but they can't just reference the second half of the concatenated file as being a duplicate of the first half.

It may seem that this is a very contrived example, but it is actually quite representative of data sets that are commonly compressed. If you think about the contents of a typical home directory on a workstation, or the contents of several home directories among a group of people who are working together then it is quite likely that the directories will contain many files with large amounts of common data. Typically the

[2] I shall refer to the family of compression algorithms that encodes the file in terms of offset/length pairs as *history based compression algorithms*. The details of these algorithms vary considerably but they all have the same basic form. One of the most commonly used examples of this type of algorithm is deflate[Gailly and Adler 1998] which is an implementation of a Lempel-Ziv method[Bell et al. 1990].

files will be temporary backups or earlier versions of current data files or may contain common elements such as images or copyright notices. When these directories (which may amount to several gigabytes) are archived using conventional compression algorithms the compression ratios achieved won't be much better than that achieved if each file were compressed separately.

The reason that these algorithms aren't able to encode such long distance redundancies is twofold. The first reason is they use a fixed bit-size in the offset encoded in the compressed file, fifteen bits in the case of deflate. As the units used for the offset are bytes this means they can't encode large offsets. The second reason is computational cost. The core of history based compression algorithms is a routine that tries to find the longest match between the current point in the file and the data in the history window. If the history window is made very large then the computational cost of searching for the longest match becomes excessive.

5.2.1 Adding an exponent

If we view rsync as a history based compression algorithm where the "history" is the old file, then we see that it solves these problems by changing the granularity of the offset portion of the offset/length pair. The new file is encoded in terms of block references into the old file plus some literal data, but these block references are just equivalent to offsets with a coarse grain. Using this coarse grain solves the problem of the short maximum offset while at the same time making the task of searching very large windows much easier, because the windows are only searched at discrete offsets. The use of the fast signature and hash tables from rsync also makes the search quite efficient.

So if we want a compression algorithm that works well with very widely spaced redundancies then we could modify the algorithm to work with offsets larger than one byte. While this would solve the problem of efficiently handling very widely spaced repeated blocks, it will reduce the compression ratio for closely spaced blocks. Ideally we want an algorithm that retains a fine grain for closely spaced repeated blocks while using a coarse grain for larger offsets.

The solution I chose for rzip was to break the offset component of Lempel-Ziv up into an offset/exponent pair. The exponent encodes the power-of-two size of the block size used for the match and the offset encodes the number of these block sized units between the current point and the start of the matching block in the history data. So when the exponent is zero it refers to matches on any byte boundary in the history data, when the exponent is five it refers to matches only on 32 byte boundaries in the history and so on. The result is that rzip is able to encode matches at very long distances but only starting at coarse byte boundaries, or it can encode matches at short distances on any byte boundary.

5.2.2 Short range redundancies

This solves the problem of long-distance matches but leaves us with an algorithm that will not perform any better than Lempel-Ziv based algorithms for short-distance redundancies. Lempel-Ziv based compressors are not anywhere near state of the art for common text compression tasks so what we really need is a combination of good long distance matching while obtaining close to state of the art compression for fine-grained matches. After some experimentation I found that a simple way of achieving this was to divide the compression into two stages. The first stage compresses with the offset/exponent based algorithm proposed above but without attempting to encode any matches that are less than β bytes long, so that shorter matches are encoded as literal strings. The second stage then compresses the resulting literal strings using a block sorting compression algorithm, which gives efficient and close to state of the art compression[Fenwick 1996].

Although this gives the basic form for rzip, the details are a bit more complex. Compression algorithms have been extensively studied and tuned over the years making it quite hard to produce an algorithm that performs well for datasets which do not have much long-distance redundancy while taking advantage of long-distance redundancy if it is present. The details of rzip are as follows:

- After some experimentation β was chosen to be 64.
- The exponent values are 4 bits and the offset values are 16 bits. The exponents range from 0 to 11, with the values 12-15 reserved for special codes. One of them is assigned to mean "literal data" and immediately precedes a literal data block

(a block of data for which no match was found).

- The lengths of matches are encoded in 8 bits and include a implicit offset of 64, so a value of one means a match length of 65 bytes[3].
- While compressing rzip keeps a 16 bit fast rolling signature (similar to that used in rsync) of a 64 byte window at the current point in the file. This signature is used to index a hash table containing a list of offsets where that signature value occurs in the history data.
- At each byte in the file the hash table is updated with an entry for the current signature. While this is done any offsets in the list for that entry in the hash table which have *expired* are removed. An expired entry is an offset which cannot be encoded using an offset/exponent pair. For example, any offset less than 2^{16} can be encoded, but only offsets on even byte boundaries can be encoded from 2^{16} to 2^{17}.
- A match is found by looking up the current fast signature in the hash table and comparing the current point in the file with the data at the offsets indicated in the list of offsets in that table entry. All list entries are compared and the longest match found.
- The rzip output is broken up into a number of *streams*, each of which is separately Huffman encoded[4] and output in an interleaved form in the compressed file. One stream contains the exponents, another the offsets and a third contains the match lengths. A final stream contains the literal data and is not Huffman encoded.
- The streams are written to the output file in large interleaved chunks.
- The output file produced by rzip is further compressed using a block sorting compression algorithm. This compresses the literal data far better than either

[3]Actually, 4 bits are used to encode the match length for matches of less than 79 bytes, with a simple encoding trick used to include the match length with the exponent bits. Bit packing in compression algorithms is a fiddly business!

[4]Any entropy coder could be used. An arithmetic coder would give better compression for higher computational cost.

using small matches or using entropy encoding. I used bzip2[Seward 1998] for this.

From the above it should be clear that rzip is intended to be used to compress files, not streams of bytes. The matching algorithm requires that fast random access is possible in the file being compressed (in order to find the block matches) and the multiple stream encoding used in the output file requires random access when uncompressing.

The algorithm also has memory requirements which rise as the log of the file size in order to store the hash table entries. This logarithmic rise stops at a file size of 2^{16+11} after which memory usage is constant at $2^{16} + 11 \times 2^{15}$ hash entries.

The use of a block sorting compression algorithm as the final stage of rzip greatly simplified the design of rzip, and allowed rzip to perform well on a much wider range of files. The Lempel-Ziv compression method is not a state of the art compression method although it is ideally suited to adaption for coarse-grained block matching using ideas derived from rsync. Block sorting compression algorithms are much closer to state of the art[Fenwick 1996], while being quite efficient. By using a block sorting compression algorithm as the final stage in rzip the algorithm can obtain good fine-grained compression while also taking advantage of any long range block repetitions.

5.2.3 Testing rzip

A compression algorithm is useless unless it performs well on real data that people might want to compress. To test rzip I first looked at existing text compression data sets, such as the Calgary Text Corpus[Witten and Bell 1990] but I found that the existing test data sets contained files that are far too small to show the advantages of long-distance block matches. To address this I put together a new Large Text Corpus[Tridgell 1998] consisting of five large files taken from real world applications. The files are

1. A tar file of the source code to GNU Emacs version 20.2.
2. A tar file of the source code to the Linux kernel, version 2.1.100.
3. A tar file snapshot of my Samba work area on my workstation.

4. A snapshot of a two month archive of the linux-kernel mailing list mirror which I run on my machine.

5. A snapshot of the unsolicited commercial email caught by the "spam filter" on my machine over a two week period.

The Samba snapshot has a high degree of internal redundancy because it contains multiple copies of different versions of the source code of Samba in various states of testing. Although each version differs slightly from the other versions, they contain a large amount of common text. The "spamfile" also contains a lot of widely spaced redundancy, due to the nature of unsolicited commercial email. Such email tends to be very repetitive and the people who send such email usually send it to many addresses on the one machine[5].

These files were chosen to represent a range of commonly compressed data while being freely distributable so that other researchers can test different algorithms on the same data.

Table 5.2 shows the result of applying gzip[6], bzip2[7] and rzip[8] to the five test files. The results show that rzip was able to find some long-distance redundancy in all the files, giving improved compression ratios for each file. The small improvements for the Emacs and Linux source code are probably due to repeated copyright headers and patterns of include files, while the improvement for the mailing list archive is probably a result of quoting and repeated mail headers and email signatures.

The improvements for the Samba snapshot and the spamfile are much larger. The Samba snapshot demonstrates the effectiveness of rzip at finding the common components in a repetitive source code archive which I believe would be typical of a programmer's home directory. The largest improvement is for the highly redundant spamfile, which has a compression ratio almost three times larger than the Lempel-Ziv based gzip and almost twice that for the bzip2 block sorting compressor.

[5] Several people have commented that the best way to compress spam is to delete it, and while I tend to agree it does make for a good example of data with a very high degree of internal redundancy.

[6] gzip version 1.2.4 was used with maximum compression selected.

[7] bzip2 version 0.1pl2 was used with maximum compression selected.

[8] The rzip snapshot from July 2nd 1998 was used with the default options.

name	file size	gzip	bzip2	rzip
Emacs	47462400	3.66	4.62	5.00
Linux	47595520	4.24	5.22	5.54
Samba	41584640	3.50	4.78	8.93
archive	27069647	3.64	4.97	5.48
spamfile	84217482	8.43	14.23	26.11

Table 5.2: Compression ratios for gzip, bzip2 and rzip

5.3 Incremental backup systems

One of the first applications for rsync that sprang to mind after it was initially developed was as part of an incremental backup system. Although this has not been implemented yet, it is interesting to examine how such a system would work.

Imagine that you need to backup a number of large files to tape, and that these files change regularly but each change only affects a small part of each file. An example of an application that might generate such files is a large database[9]. A typical backup system would either backup all files or backup any files that had changed in any way since the last backup. If the time taken to do backups is important (as it often is) then this is an inefficient way of doing backups.

The solution is to backup only the changes in the files rather than the whole file every time. The problem with doing this is that normal differencing algorithms need access to the old file in order to compute the differences. This is where rsync comes in.

To backup incrementally with rsync the backup application needs to store the fast and strong signatures used in the rsync algorithm along with the file. When the time comes to perform an incremental backup these signatures are then read and the rsync matching algorithm is applied to the new file, generating a stream of literal blocks and block match tokens. This delta stream is then stored to tape as the incremental backup.

Note that the backup system does not need to read the old file from tape in order

[9] I should note that rsync seems to work particularly well with large database files. It is used by several large companies (including Nokia and Ford) to synchronize multi-gigabyte engineering databases over slow network links and I use it myself for a number of smaller databases. I didn't include a database file in the example data sets because I couldn't find one that didn't contain any proprietary data. A proprietary data set would have prevented other researchers from reproducing the results.

to compute the differences. This is important because it allows the backup to proceed very quickly, particularly if the speed of the tape device is the limiting factor in the backup process. As the extra storage space required by the signatures is only a tiny fraction of the total space consumed by the file on tape, the storage overhead is small even if the rsync algorithm does not find a significant number of matches between the new and old files.

This sort of incremental backup system would be particularly effective when large hierarchical storage systems are used as the backup medium. These devices typically have a large front-end set of disks that are used to stage data to a high latency tape robot. If the file signatures were kept on the front-end disks then no tape reading would be required before the incremental backup started.

5.4 rsync in HTTP

The Hyper-Text Transport Protocol (HTTP) is the most widely used protocol for presenting formatted information on the Internet, and as such the network efficiency of the protocol is very important. Increased efficiency results in better response times for users and decreased network costs.

Although cache systems such as those employed by popular web browsers or implemented in proxy caching web servers are very effective at reducing this traffic for static documents, they do not help at all for dynamic documents. For documents that change regularly the currently deployed browsers and caching servers must fetch the whole page every time, even if only a small part of the page changes. With the trend towards dynamically generated web sites such dynamic documents are becoming more common.

To address this issue a proposal has been put to the W3C, the standards body that coordinates HTTP, to incorporate support for differencing in the HTTP protocol. The proposal[Hoff and Payne 1997] introduces a differencing format called *gdiff* for sending changes in HTTP served documents and suggests the rsync algorithm as the means of generating the differences.

The way this would work is that a client requesting a page would send an augmented HTTP GET request to the server which contained the fast and strong signatures for the page currently held in the local cache. The server would then apply the rsync algorithm to the new page and send back the differences in gdiff format.

The notable features of this system are:

- the server does not need to keep a record of the pages that it has previously served in order to compute differences;
- the algorithm does not rely on any defined structure in the document; and
- the worst case overhead (in the case that there are no common blocks between the old and new page) is well defined and can be kept small.

The major disadvantage of this system is the additional computational costs imposed on the server. The practicality of the system will depend on the relative changes in CPU power and effective network bandwidths in common use on the Internet.

5.5 rsync in a network filesystem

Another possible use for rsync is as part of a network filesystem. The advent of proposed wide area filesystem protocols such as CIFS[Leach and Naik 1998], WebNFS[Callaghan 1998] and CODA[Satyanarayanan 1998] where clients and servers may be separated by high latency, low bandwidth Internet links provides the ideal environment for an efficient remote data update algorithm like rsync.

The particular feature of these filesystem protocols which makes them good candidates for using the rsync algorithm is the notion of file leases, particularly write leases. A file write lease is an exclusive contract between the server and client giving the client sole access to a file for an extended period of time. These leases allow the client to safely cache file changes locally which can have a dramatic effect on the effective throughput of the network filesystem.

The use of the rsync algorithm comes when the file lease is terminated. At that time the client must send any changes that have been made to the file to the server so that the server can make these changes available to other clients. If the changes are small, the rsync algorithm could be utilized to update the server's copy of the file much more efficiently than could be done by sending the whole file.

To extend an existing filesystem protocol to support efficient remote rsync updates requires the following additions to the standard suite of filesystem operations[10]:

- The client must be able to request a list of fast and strong signatures from the server for a specified file. Alternatively these could be sent as part of the write lease break request sent from the server.
- The client must be able to request a file to file block copy operation. This would operate like a combined read/write request which takes two file handles, two file offsets and a length. The client would use this to remotely reconstruct the new file on the server.

As with the proposed HTTP extensions the practicality of these filesystem extensions will depend on the ratio of CPU power to network bandwidth available in real deployments of these network protocols.

5.6 Conclusion

This chapter has explored a number of alternative uses for the ideas behind the rsync algorithm. Some of the uses have been implemented and demonstrated to work well on real-world data while others remain to be implemented.

It is also likely that futher uses for rsync will be discovered once the algorithm becomes better known. I think that the rsync algorithm has the potential to become a commonly used tool in the toolboxes of software engineers, particularly given the explosive growth of wide area networks and online communities that has been seen in recent years.

[10] I have discussed these additions with an engineer working on the specification of the next version of the NFS protocol. I am hopeful that they will be included.

Chapter 6

Conclusion

This thesis has covered a range of algorithms for parallel sorting and remote data synchronization. The first chapter described a comparison based internal parallel sorting algorithm with low memory overhead and high parallel efficiency. The design of the algorithm took advantage of a hypercube primary sorting algorithm that works very efficiently but leaves a small proportion of elements unsorted. This is followed by a less efficient cleanup algorithm that completes the sort.

The second chapter looked at external parallel sorting using the internal parallel sorting algorithm of the first chapter as a building block. The external algorithm, based around a two dimensional algorithm called shearsort, is able to sort large disk based data sets in place while using little more than two read/write I/O pairs per element. This is achieved by taking advantage of the fact that the first pass of the algorithm leaves most elements in their correct final positions, in a manner similar to the primary sorting phase of the internal algorithm.

In the third chapter the thesis starts to look at an algorithm for remote data synchronization. The rsync algorithm, which arose out of work associated with the development of the external parallel sorting algorithm from the second chapter, provides a low latency remote update mechanism appropriate for a wide range of data sets. The algorithm uses a dual signature method to match blocks of data at arbitrary byte boundaries while remaining computationally feasible for real-world applications.

Chapter four looked at optimizations to the basic rsync algorithm, reducing the communications overhead of the signatures and taking advantage of stream compression techniques. The issue of latency when transferring multiple files over high latency links was also addressed and solved using a simple pipelining technique.

The final chapter looked at alternative uses for the ideas behind the rsync algorithm, including text compression, differencing, incremental backup and enhancements for commonly used network protocols. Some of the proposed algorithms remain to be tested while others have been shown to work well on real-world data.

Some of the algorithms presented in this thesis offer considerable opportunities for further research. The sorting algorithms need, if possible, a more complete analytical analysis to put strict bounds on the observed efficiency. It would also be useful to consider extensions to allow the sorting of elements with non-uniform sizes.

The rsync algorithm also offers extensive possibilities for further research, especially with regard to the ideas presented in the last chapter. The wide applicability of the ideas behind rsync combined with the explosion in the application of network technologies that has happened in recent years bodes well for the future of algorithms like rsync.

Appendix A

Source code and data

A large part of the work behind this thesis involved writing the source code for the various algorithms described. In total about 18000 lines of source code were written, although only a part of that was for the core parts of the algorithms, the rest was driver and support code.

Rather than including the code in this thesis I decided to make it available for download on the Internet. The various parts of the code and their state of development are described in this appendix. This appendix also contains some information on the availability of the data sets used in the testing of rsync and rzip.

If you have any questions about the source code or data sets then please contact me via email at tridge@samba.org.

A.1 Internal parallel sorting

The internal parallel sorting code is available from
ftp://samba.org/pub/tridge/sorting/internal/.

The main algorithm is in par_sort.c and should be reasonably easily portable to a wide range of distributed memory parallel machines. To date it has been ported in various forms to six different platforms; the AP1000, AP+/Linux, CM5, Cenju, VP-P300 and NCube. The code as distributed only supplies build instructions for the AP1000 and CM5, if you are interested in other platforms then please contact me.

The driver routines use a custom portable message passing layer contained in mimd.h which can use a variety of underlying message passing libraries. The code is suitable for academic and research use.

A.2 External parallel sorting

The external parallel sorting code is available from
ftp://samba.org/pub/tridge/sorting/external/.

The main algorithm is again in par_sort.c and is derived from the internal parallel sorting code, it also includes a simplified version of the internal algorithm as a subroutine.

The driver routines assume MPI but could be easily ported to other message passing systems. The code is less well developed than the internal parallel sorting code but would be suitable for researchers familiar with parallel programming.

A.3 rsync

The rsync source code is available from http://rsync.samba.org/. Binaries are also available for a wide range of UNIX-like platforms. The source code uses GNU autoconf to facilitate portability.

The rsync code is in production use at a large number of sites and is suitable for general purpose efficient file transfer and mirroring. A library version of the algorithm suitable for embedding in other applications is not yet available but is planned for sometime in 1999.

To reproduce some of the results in this thesis requires a small amount of code modification to disable some features of the code (such as the reduced signature sizes). I can provide details on request.

A.4 rzip

The rzip source code is in a state of early development and should only be used by researchers very familiar with C programming. It is available via anonymous CVS at :pserver:cvs.samba.org:/cvsroot as module *rzip*[1]. The code can also be viewed by a web browser at http://cvs.samba.org/cgi-bin/cvsweb/rzip/.

The current implementation of rzip is very slow. It could be made much faster and the speed problem will be addressed as the algorithm is further developed. No

[1] Details on accessing the anonymous CVS repository are available from http://samba.org/cvs.html

documentation is provided.

A.5 rsync data sets

The data sets used in the testing of the rsync algorithm are available from ftp://samba.org/pub/tridge/rsync_data/. The Linux and Samba data sets are in gzip compressed format.

I am actively seeking new public data sets for the testing of rsync and similar algorithms. Please contact me if you have some data that is suitable. I would particularly like to have a broader variety of data sets.

A.6 rzip data sets

The data sets used in the testing of rzip are available from ftp://samba.org/pub/tridge/large-corpus/. These data files are stored in bzip2 compressed format.

Appendix B

Hardware

This appendix gives a brief description of the hardware used to generate the results given in this thesis.

B.1 AP1000

The Fujitsu AP1000 [Ishihata et al. 1991] at ANU 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 AP1000 supports a general message passing model using a proprietary library or the MPI standard. The CPUs have a 25 MHz clock speed and a 128KB cache.

B.2 CM5

The Thinking Machines CM5 [TMC 1991] that I used contained 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 this thesis no use is made of the vector processors other than as memory controllers. The CM5 supports a general message passing model using the proprietry CMMD message passing library. The CPUs have a 32 MHz clock speed and a 64KB cache.

B.3 RS6000

The results in Chapter 5 were produced on a IBM RS6000 with a 233 MHz 604 PowerPC CPU. The machine has 128 MB of RAM and runs AIX 4.2.

Bibliography

AGGARWAL, A. AND PLAXTON, C. 1993. Optimal parallel sorting in multi-level storage. Technical Report CS-TR-93-22, University of Texas at Austin. (p. 47)

AJTAI, M., KOLMOS, J., AND SZERMEREDI, E. 1983. Sorting in $c \log n$ parallel steps. *Combinatorica 3*, 1 – 19. (p. 9)

AKL, S. G. 1985. *Parallel Sorting Algorithms*. Academic Press, Toronto. (p. 47)

BATCHER, K. E. 1968. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, Volume 32 (1968), pp. 307 – 314. (pp. 7, 14)

BELL, T., CLEARY, J., AND WITTEN, I. 1990. *Text Compression*. Prentice Hall. (p. 86)

BLELLOCH, G. E., LEISERSON, C. E., MAGGS, B. M., PLAXTON, C. G., SMITH, S. J., AND ZAGHA, M. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proc. Symposium on Parallel Algorithms and Architectures* (Hilton Head, SC, July 1991). (p. 9)

BURROWS, M. AND WHEELER, D. 1994. A block-sorting lossless data compression algorithm. Technical Report SRC Research Report 124 (May), Digital Systems Research Center. (p. 78)

CALLAGHAN, B. 1998. Network file system version 4. http://www.ietf.org/html.charters/nfsv4-charter.html. (p. 94)

ELLIS, J. AND MARKOV, M. 1998. A fast, in-place, stable merge algorithm. ftp://csr.uvic.ca/pub/jellis/. (p. 6)

FENWICK, P. 1996. Block sorting text compression. In *Proc. 19th Australasian Computer Science Conference, Melbourne, Australia* (January 1996). (pp. 88, 90)

FOX, G. C., JOHNSON, M. A., LYZENGA, G. A., OTTO, S. W., SALMON, J. K., AND WALKER, D. W. 1988. *Solving Problems on Concurrent Processors, Volume 1*. Prentice-Hall. (pp. 10, 15, 21)

GAILLY, J. AND ADLER, M. 1998. zlib. http://www.cdrom.com/pub/infozip/zlib/. (pp. 55, 75, 86)

GNU. 1998. The GNU project. http://www.gnu.org. (p. 14)

GUTENBERG. 1998. Project gutenberg. http://www.gutenberg.net. (p. 79)

HELMAN, D., BADER, D., AND JÀJÀ, J. 1996. A randomized parallel sorting algorithm with an experimental study. Technical Report CS-TR-3669, Institute for Advanced Computer Studies, University of Maryland. (p. 32)

HOFF, A. AND PAYNE, J. 1997. Generic diff format specification. http://www.w3.org/TR/NOTE-gdiff-19970825.html. (p. 93)

HUANG, B. AND LANGSTON, M. A. 1988. Practical in-place merging. *Communications of the ACM 31*, 348 – 352.

ISHIHATA, H., HORIE, T., INANO, S., SHIMIZU, T., KATO, S., AND IKESAKA, M. 1991. Third generation message passing computer AP1000. In *International Symposium on Supercomputing* (November 1991), pp. 45 – 55. (pp. 3, 101)

ISHIHATA, H., HORIE, T., AND SHIMIZU, T. 1993. Architecture for the AP1000 highly parallel computer. *Fujitsu Sci. Tech. J. 29*, 6 – 14. (p. 14)

KARP, R. AND RABIN, M. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development 31*, 249 – 260. (p. 55)

KNUTH, D. E. 1981. *The Art of Computer Programming*, Volume 3: Sorting and Searching. Addison-Wesley. (pp. 4, 7, 10, 21)

KRONROD, M. A. 1969. An optimal ordering algorithm without a field of operation (in Russian). *Dokl. Akad. Nauk SSSR 186*, 1256 – 1258.

LEACH, P. AND NAIK, D. 1998. Common internet filesystem protocol. ftp://ftp.ietf.org/internet-drafts/draft-leach-cifs-v1-spec-01.txt. (p. 94)

MACDONALD, J. 1998. Versioned file archiving, compression and distribution. ftp://ftp.xcf.berkeley.edu/pub/xdelta/. (p. 84)

MCILROY, D., MCILROY, P., AND BOSTI, K. 1993. Engineering radix sort. *Computing systems 6*, 1, 5 – 27. (pp. 32, 43)

NATVIG, L. 1990. Logarithmic time cost optimal parallel sorting is not yet fast in practice! In *Proc. Supercomputing '90* (1990), pp. 486 – 494. IEEE Press. (p. 26)

NODINE, M. AND VITTER, J. 1992. Optimal deterministic sorting in parallel memory hierarchies. Technical Report CS-92-38 (August), Dept. of Computer Science, Brown University. (p. 47)

ORLITSKY, A. 1993. Interactive communication of balanced distributions and of correlated files. *SIAM J. Disc. Math. 6*, 4, 548 – 564. (p. 51)

PYNE, C. 1995. Remote file transfer method and apparatus. US patent number 5446888. (p. 68)

RIVEST, R. 1990. The MD4 message digest algorithm. RFC1186. (p. 53)

SATYANARAYANAN, M. 1998. Coda file system. http://www.coda.cs.cmu.edu/. (p. 94)

SCHNEIER, B. 1996. *Applied Cryptography*. Wiley. (p. 53)

SCHNORR, C. AND SHAMIR, A. 1986. An optimal sorting algorithm for mesh connected computers. In *STOC'86* (1986), pp. 255 – 263. (pp. 37, 41)

SEWARD, J. 1998. bzip2. http://www.muraroa.demon.co.uk/. (p. 90)

TAYLOR, R. 1998. rzip. Private communication. (p. 86)

TAYLOR, R., JANA, R., AND GRIGG, M. 1997. Checksum testing of remote synchronisation tool. Technical Report 0627 (November), Defence Science and Technology Organisation, Canberra, Australia. (p. 72)

THEARLING, K. AND SMITH, S. 1992. An improved supercomputing sorting benchmark. In *Supercomputing 92* (1992), pp. 14 – 19. IEEE Press. (pp. 9, 31)

TMC. 1991. CM-5 technical summary. (p. 101)

TRIDGELL, A. 1996. The PIOUS project. http://acsys.anu.edu.au/research/completed/pious.html. (p. 1)

TRIDGELL, A. 1998. Large corpus. ftp://samba.anu.edu.au/pub/tridge/large-corpus. (p. 90)

TRIDGELL, A. AND BRENT, R. P. 1993. An implementation of a general-purpose parallel sorting algorithm. Technical Report TR-CS-93-01 (February), Computer Sciences Lab, Australian National University. (p. 8)

TRIDGELL, A. AND BRENT, R. P. 1995. A general-purpose parallel sorting algorithm. *Int. J. High Speed Computing 7*, 285 – 301. (p. 32)

TRIDGELL, A., BRENT, R. P., AND MCKAY, B. 1997. Parallel integer sorting. Technical Report TR-CS-97-10 (May), Department of Computer Science, Australian National University. (p. 37)

TRIDGELL, A. AND HAWKING, D. 1996. Bigram BMG text search algorithm. in preparation. (p. 1)

TRIDGELL, A., MILLAR, B., AND AHN-DO, K. 1992. Alternative pre-processing techniques for discrete hidden Markov model phoneme recognition. In *ICSLP'92* (1992), pp. 631 – 634. (p. 1)

TRIDGELL, A. AND WALSH, D. 1996. The HiDIOS filesystem. In *PCW'95* (September 1996), pp. 53 – 63. (pp. 1, 34)

WITTEN, I. AND BELL, T. 1990. Calgary compression corpus. ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/. (p. 90)

ZHOU, B. B., BRENT, R. P., AND TRIDGELL, A. 1993. Efficient implementation of sorting algorithms on asynchronous distributed-memory machines. Technical Report TR-CS-93-06 (March), Computer Sciences Lab, Australian National University. (p. 17)