Efficient Implementation of Sorting Algorithms on Asynchronous Distributed-Memory Machines*

Zhou B. B., Brent R. P. and Tridgell A.†
Computer Sciences Laboratory
The Australian National University
Canberra, ACT 0200, Australia

Report TR-CS-93-06
March 1993
(revised May 1993)

Abstract

The problem of merging two sequences of elements which are stored separately in two processing elements (PEs) occurs in the implementation of many existing sorting algorithms. We describe efficient algorithms for the merging problem on asynchronous distributed-memory machines. The algorithms reduce the cost of the merge operation and of communication, as well as partly solving the problem of load balancing. Experimental results on a Fujitsu AP1000 are reported.

1 Introduction

This paper considers an important aspect of parallel sorting algorithms on asynchronous distributed-memory machines. The algorithms of interest to us first perform a local sort on each processing element (PE), then perform a sequence of merges to globally sort the data. Each merge involves pairs of PEs. The two PEs in a pair merge their two sorted sequences, and each PE keeps half of the merged sequence. It is known that this "merge-split" scheme can be applied to achieve high efficiency if a large sorting problem is to be solved on a machine with many relatively small processing elements [1, 3].

The most straightforward method for merging is first to transfer a sequence of elements from one PE to the second PE, then to merge the two sequences in the second PE, and finally to transfer the upper half (or lower half) back to the first PE. This method is inefficient, both because of its memory requirements, and because of the load imbalance – only one PE in each pair is active during the merging. The efficiency can be improved by taking advantage of the following observations.

To merge two sequences of elements, it is only necessary for each PE to transfer a portion of its sequence to the associated PE. In many sorting methods, e.g., odd-even transposition sort [3], Batcher's merge-exchange sort [2], and parallel Shell sort [6], the entire set of elements becomes more nearly sorted after each iteration, so the portion of the sequence to be transferred from a PE tends to decrease. If we have an algorithm which can efficiently find the exact number of elements to be transferred between any pair of PEs, we may reduce the cost of the merge

†E-mail addresses: {bing,rpb,tridge}@cslab.anu.edu.au rpb142tr typeset using LaTeX

operation and communication, as well as partly solving the problem of load balancing. This idea was used by Tridgell and Brent [7]. They introduced an algorithm *find_exact* that finds the exact number of elements to be transferred in $\log_2 N$ communication steps, where N is the length of the sequence stored in each PE's local memory. At each communication step one element is sent and received by each PE. The algorithm works well on distributed memory machines such as the Thinking Machines CM5 and the Fujitsu AP1000, because these machines have a small "startup" time for communication and a small message latency, so the time for running the *find_exact* procedure is small in comparison to the total communication time. On a machine with a high message latency, the algorithm would be costly. In this paper we introduce a new algorithm (Algorithm 2 below), which requires only $\log_\lambda N$ communication steps (for $\lambda \geq 2$) to find the exact number of elements to transfer, if we allow $\lambda - 1$ elements to be transferred at each step. By properly choosing λ, the running time of the algorithm can be reduced.

In Section 2 algorithms for finding the exact number of elements to be transferred are derived. Experimental results on the Fujitsu AP1000 are given in Section 3. We use the odd-even transposition sorting method as an example to show the efficiency gained by using the algorithm. The odd-even transposition sort requires only nearest-neighbour communication in a one-dimensional array, so is applicable to most special-purpose machines with restricted communication topologies. Conclusions are drawn in Section 4.

2 Algorithms

To simplify our discussion, we assume in the following that the elements are distinct, are sorted in increasing order in each PE, and that each PE has the same number (N) of elements. A processing element is referred to as PE1 (or PE2) if it stores the first half (or the second half) of the total elements after a merge. The elements in each PE are enumerated from 0 to $N-1$. The $(k+1)^{th}$ element in PE1 (or PE2) is referred to as e_k (or e'_k), for $0 \leq k < N$.

In the following Lemma we set e_{-1} and e'_{-1} to $-\infty$ and e_N and e'_N to $+\infty$ so that the inequalities still hold in two extreme cases when $K = 0$, that is, no element in PE1 is smaller than any element in PE2; and when $K = N$, that is, no element in PE1 is greater than any element in PE2.

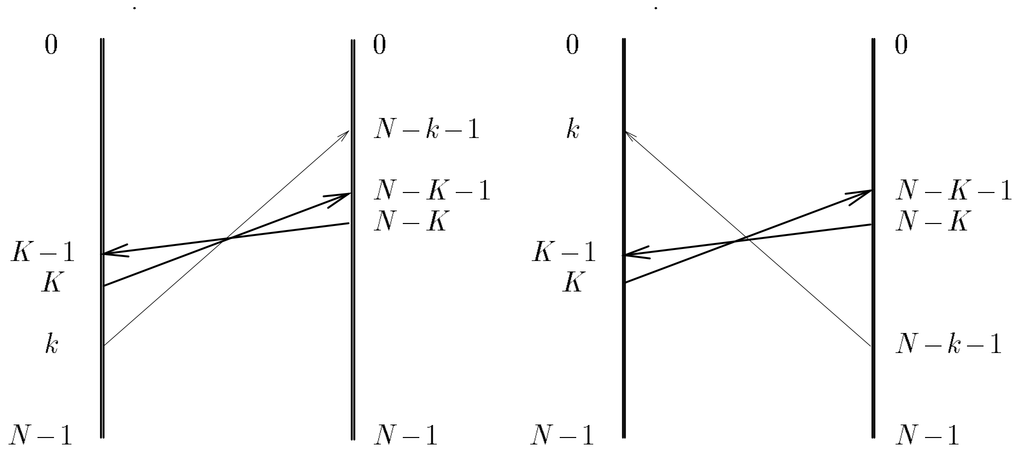
Lemma 1 *To merge two sorted sequences of N elements each stored in one PE, the exact number of elements to be transferred between the two PEs is $N - K$, for $0 \leq K < N$, if and only if the following inequalities are satisfied:*

$$\begin{cases} e_K > e'_{N-K-1}, \\ e_{K-1} < e'_{N-K}. \end{cases} \tag{1}$$

Proof. Our aim is to merge two sorted sequences and store the first half in PE1 and the second half in PE2. Suppose that K is chosen so that the two inequalities in (1) are satisfied. Since the original sequences are sorted, we have $e_K > e_{K-1}$ and $e'_{N-K} > e'_{N-K-1}$. We transfer the last $N-K$ elements from PE1 to PE2 and the first $N-K$ elements from PE2 to PE1. It is easy to see that, after the transfer, the largest element in PE1 is $\max(e_{K-1}, e'_{N-K-1})$, and the smallest element in PE2 is $\min(e_K, e'_{N-K})$. Thus, no element in PE1 is greater than any element in PE2. On the other hand, if K is chosen so that either of the two inequalities in (1) is not satisfied, there must be at least one element in PE1 that is greater than the smallest element in PE2 after the transfer.

Corollary 1 *Given an arbitrary index k, $0 \le k < N$, we have*

$$\begin{cases} e_k < e'_{N-k-1} & \text{if } 0 \le k < K, \\ e_k > e'_{N-k-1} & \text{if } K \le k < N. \end{cases} \tag{2}$$



(a) $K \le k < N$ (b) $0 \le k < K$

Figure 1: A graphical expression of Corollary 1.

Proof. An illustration of the proof is given in Fig. 1. The two vertical lines represent the two sequences, while an arrow line pointing from the left (or right) sequence to the right (or left) sequence indicates $e_x > e'_{N-x-1}$ (or $e_x < e'_{N-x-1}$). Assume that there is an index k for $k \ge K$. We have $e_k \ge e_K$ and $e'_{N-K-1} \ge e'_{N-k-1}$, since the original sequences are sorted. However, it is known from (1) that $e_K > e'_{N-K-1}$. Thus the element e_k must be greater than e'_{N-k-1}. The proof for the first inequality is similar.

Corollary 2 *The index K in* (1) *is unique.*

Proof. Suppose that there is another index K' satisfying the inequalities (1), as shown in Fig. 2. If $K' > K$, we have $K' - 1 \ge K$, and thus the element $e_{K'-1}$ is greater than $e'_{N-K'}$, by Corollary 1. Similarly, $e_{K'}$ is smaller than $e'_{N-K'-1}$ if $K' < K$. In either case the result is a contradiction. Thus K' satisfies the inequalities (1) if and only if $K' = K$.

Algorithm 1

Using the above lemmas, a simple "bisection" algorithm can be derived to find the exact number of elements to be transferred between two PEs, or more specifically, to find the unique index K which satisfies the inequalities (1). We outline the algorithm.

At any stage the index K is known to lie in a certain range. The boundary indices, that is, the first and last elements of the range, are called *top* and *bottom* respectively[1]. Thus, K is known to be in $(top, bottom]$. PE1 sends the middle element of the interval, e_{mid}, to PE2.

[1]Note that with our conventions $top \le bottom$.

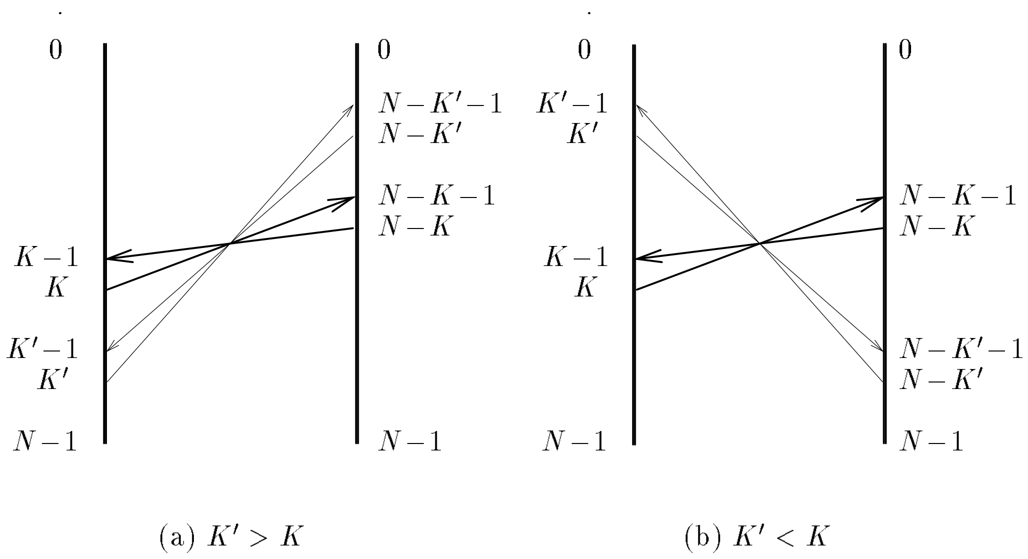


Figure 2: A graphical expression of Corollary 2.

This element is then compared with $e'_{N-mid-1}$ in PE2, and the result is sent back to PE1. If $e'_{N-mid-1}$ is greater than e_{mid}, the index K must be in the half-open interval (*mid*, *bottom*]; otherwise it must be in the half-open interval (*top*, *mid*], by Corollary 1. This is illustrated in Fig. 3. In either case the interval (*top*, *bottom*] can be updated.

The procedure is applied until there is only one element in the interval. It is easy to see that $\lceil \log_2 N \rceil$ steps[2] are required to find the index K.

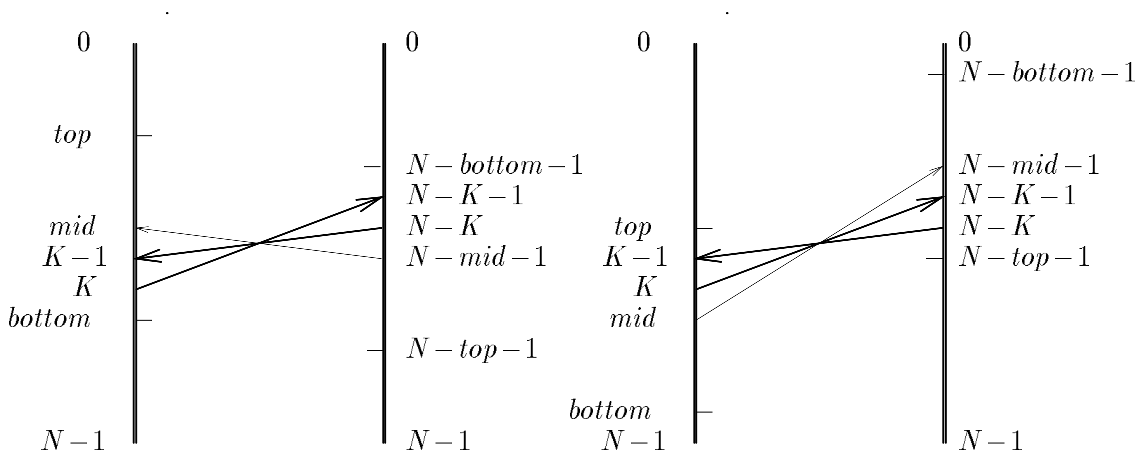
This algorithm was derived by Tridgell and Brent [7], and has been implemented on both the CM5 and the Fujitsu AP1000. The results are good because both machines have a small message latency, so the time for finding K is small in comparison with the total communication time. On a machine with a high message latency, the communication costs due to multiple small messages would be considerable.

A modification of Algorithm 1 can reduce the search interval by more than a factor of two at each step, and thus reduce communication startup costs. The modification follows from Lemma 2. In the lemma, if $e_k > e'_{N-top-1}$, we define $k' = top$, and if $e_k < e'_{N-bottom-1}$, we define $k'' = bottom$.

Lemma 2 *Suppose that K is within the interval $(top, bottom]$ and that k is an index in the interval. If e_k is greater than e'_{N-k-1} and $e'_{N-k'-1}$ is the first element in PE2 which is greater than e_k for k' in the interval and $k' < k$, then index K must be in the half-open interval $(k', k]$. If e_k is smaller than e'_{N-k-1} and $e'_{N-k''-1}$ is the first element in PE2 which is smaller than e_k for k'' in the interval and $k < k''$, then index K must be in the half-open interval $(k, k'']$.*

Proof. We only prove the case $e_k < e'_{N-k-1}$. The proof for $e_k > e'_{N-k-1}$ is similar. Since $e_k < e'_{N-k-1}$, we have $k < K$ by Corollary 1, and so $e_{K-1} \geq e_k$. We know from (1) that $e'_{N-K} > e_{K-1}$, so $e'_{N-K} > e_k$. Since $e_k > e'_{N-k''-1}$, we also have $e'_{N-K} > e'_{N-k''-1}$. Thus $e_{k''} > e'_{N-k''-1}$ and $K \leq k''$ by Corollary 1. This gives $k < K \leq k''$ (see Fig. 4).

[2] A single step requires communication in both directions between PE1 and PE2. By alternating the roles of PE1 and PE2, s steps could be performed with $s+1$ communications.



(a) $e_{mid} < e'_{N-mid-1}$ (b) $e_{mid} > e'_{N-mid-1}$

Figure 3: Deciding the new search interval for the next step.

Using Lemma 2, a search procedure is required at each step in order to find the index k' or k''. Since the search interval may be reduced by more than half, the number of steps may be less than the number required by Algorithm 1. Note that the number of steps may still be close to $\log_2 N$ in the worse case, which may occur when K is close to 0 or $N-1$. In next paragraph we describe a new algorithm (Algorithm 2), in which $\lambda - 1$ elements ($\lambda \geq 2$) are allowed to be transferred from PE1 to PE2 at each step, and the total number of steps is about $\log_\lambda N$. By properly choosing λ, the time to find K can be reduced significantly.

Algorithm 2

Divide the search interval into λ smaller intervals with each of these intervals containing c_i elements. We may obtain $\lambda - 1$ elements in PE1. The original index of the first of these elements is $top + c_1$, and the original index for the l^{th} element is $\sum_{i=0}^{l} c_i$, where $c_0 = top$. The $\lambda - 1$ elements are sent from PE1 to PE2. Once the elements are received by PE2, a similar procedure to that for finding the exact number described previously is applied to find, from the $\lambda - 1$ elements, the index of the k^{th} element which satisfies the two inequalities

$$\begin{cases} e_L > e'_{N-L-1} \\ e_{L-c_k} < e'_{N-(L-c_k)-1} \end{cases} \tag{3}$$

where e_L is the k^{th} element and L is its original index. Likewise e_{L-c_k} is the $(k-1)^{th}$ element and $L - c_k$ is its original index. The only difference between this procedure and Algorithm 1 is that all computations are performed locally and no extra communication is required in the procedure since the $\lambda - 1$ elements have already been sent from PE1 to PE2.

Lemma 3 *If the above procedure is applied, the index K must be in the half-open interval $(L - c_k, L]$.*

Proof. We prove that the two inequalities (3) cannot both be satisfied if index K is not in the interval. If $L < K - 1$, we have $e_L < e'_{N-L-1}$ by Corollary 1. We also have $e_{L-c_k} > e'_{N-(L-c_k)-1}$

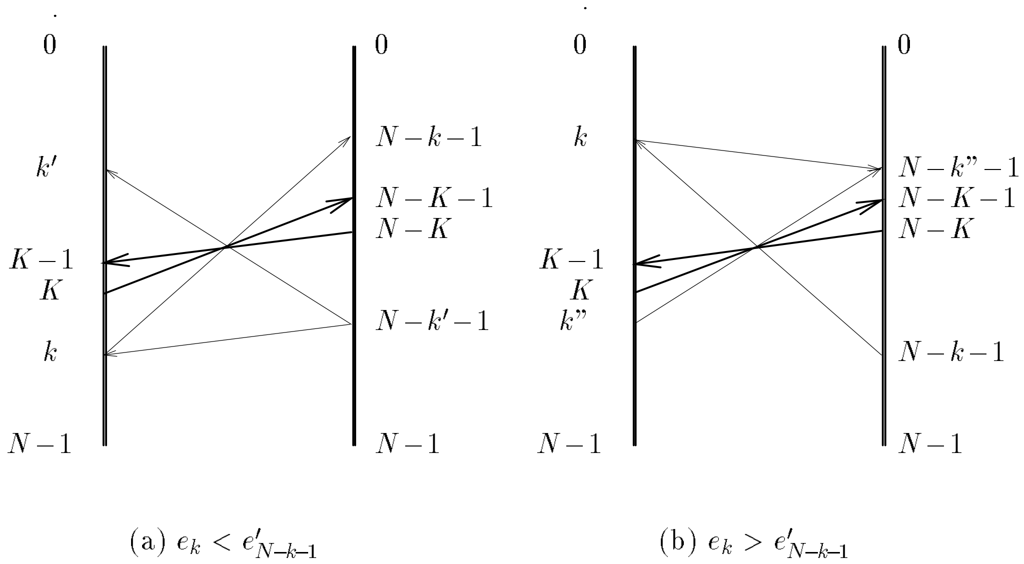


Figure 4: A graphical expression of Lemma 2

if $L - c_k > K$. It is easy to see that in either case the inequalities in (3) cannot both be satisfied. Therefore, index K must be in the interval $L - c_k < K \leq L$.

Since $\lambda - 1$ elements are sent to PE2 at each step, a much smaller search interval can be decided for the next step, and the total number of steps required to find the exact number is decreased. Supposing that all intervals have equal size at each step, the total number of steps is only $\log_\lambda N$. If λ is not very large, the "startup" time for communication will be dominant in the running time of Algorithm 2. Therefore, Algorithm 2 will be more efficient than Algorithm 1, by a factor of about $\log_2 \lambda$. Our experimental results on the Fujistu AP1000 confirm this prediction.

It is worth noting that the two algorithms are exactly the same if λ is set to one and the two intervals at each step are equally divided in Algorithm 2. Therefore, Algorithm 1 is just a special case of Algorithm 2.

3 Experimental Results

We use the odd-even transposition sort as an example to show that the efficiency can be gained by adopting the algorithms described in Section 2. Our experimental results were obtained on the Fujitsu AP1000 located at the Australian National University. The Fujitsu AP1000 is a distributed memory MIMD machine with up to 1024 independent 25 MHz SPARC processors for processing elements. (Our machine has 128 PEs.) Each PE has 16 MByte of dynamic RAM and 128 KByte cache. The topology of the machine is a torus, with hardware support for wormhole routing. The communication network (T-net) provides a theoretical bandwidth of 25 Mbyte/sec between PEs, and in practice about 6 MByte/sec is achievable. For details of the AP1000 architecture and software environment see [4, 5].

The odd-even transposition sort is not optimal for the AP1000 (better methods are given in [7]), but we use it because it only requires nearest-neighbour communication in a one-dimensional array. The methods of [7] take advantage of the wormhole routing and use more general communication patterns (for example, communication along the edges of a hypercube).

Table 1: Experimental results.

Problem size (millions)		1.280	5.120	10.24	12.80	25.60	38.40	51.20	64.00
Program 1	time (sec.)	12.58	58.57	118.8	148.9	298.0	447.4	596.7	746.0
Program 2	time (sec.)	6.688	29.96	61.96	77.94	156.1	235.2	313.5	394.8

In our examples the 128 PEs are configured as a one-dimensional array.

The original odd-even transposition sorting algorithm described in [3], and the modified algorithms that utilise Algorithms 1 and 2, have been implemented to sort large sets of 32-bit integers on 128 PEs. The algorithms described in Section 2 work almost equally well on the AP1000 for moderate values of λ. This is because the AP1000 has a small message latency (less than 100 μsec). Thus, the cost of finding K is only a small portion of the total communication cost, and the overall efficiency gained by using $\lambda > 2$ is small.

Some experimental results are given in Table 1. In this table "Program 1" is the program implemented for the original odd-even transposition sorting algorithm, and "Program 2" is a modified version which incorporates Algorithm 2 (as discussed in Section 2) with $\lambda = 10$. It is clear that the use of Algorithm 2 approximately doubles the efficiency of the sort.

4 Conclusions

We have described several algorithms for finding the exact number of elements to be transferred between two PEs when merging two sequences of elements stored separately in the PEs. Although the algorithms require a number of communication steps to send/receive small messages between the PEs, a large gain in merging efficiency can be obtained. This is because the merge operations are performed by two PEs instead of just one PE, the computational load is better balanced, and the cost of transferring large messages between the PEs may be reduced.

When the odd-even sorting method is implemented on the AP1000, Algorithm 1 and Algorithm 2 (with $\lambda > 2$ but not too large) are almost equally effective in reducing the merge time. This is because the AP1000 has low message latency. The difference between the Algorithms 1 and 2 would be more significant on a machine with a high message latency.

References

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

[2] K. E. Batcher, "Sorting networks and their applications", *Proc. AFIPS 1968 Spring Joint Comput. Conf.*, 1968, 307–314.

[3] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers", *IEEE Trans. on Computers*, C–27, 1978, 84–87.

[4] R. P. Brent (editor), *Proceedings of the CAP Workshop '91*, Australian National University, Canberra, Australia, November 1991.

[5] R. P. Brent and M. Ishii (editors), *Proceedings of the First CAP Workshop*, Fujitsu Research Laboratories, Kawasaki, Japan, November 1990.

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

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