Optimal Parallel Sorting on a Linear Processor Array

Arvin Park

K. Balasubramanian

Department Computer Science
Princeton University
Princeton, New Jersey 08544

CS-TR-046-86
July 29, 1986

Abstract

The problem of sorting n elements on k linearly connected processors is examined. We reduce the communication complexity (measured in units of single data transfers between adjacent processors) by a factor of two from the previous best bound [Baud78] while maintaining the same (asymptotically optimal) number of comparisons. The result holds even if only unidirectional communication between processors is allowed (a unidirectional ring architecture). This result is significant because communication time dominates computation time in most realistic parallel sorting problems.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0446 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Introduction

Linearly connected multiprocessor architectures have become increasingly prevalent. Many linear Systolic Arrays [Kung82], general purpose linearly connected multiprocessors [Hori86], and ring connected machines [Wilk79] [Farb75] have been designed and built in recent years. This paper examines sorting algorithms on these types of machines.

We present a modification of the *Neighborhood Sort* [Baud78] which is a generalization of *Odd-Even Transposition Sort* (described in [Knut73]). This modified algorithm matches the asymptotically optimal $n\log n/k$ (k = number of processors, n = number of elements) number of comparisons for Neighborhood sort while reducing the number of communication steps from $2n$ to n.

At first glance it would seem that $n\log n/k$ comparisons would dominate $O(n)$ communication steps. This is in general not the case for realistic parallel sorting problems. Communication operations involve expensive processor to processor data transfer operations which can take orders of magnitude more time than fast register to register computation steps [Wilk79]. It is also likely that k is close to $\log n$ (e.g $k = 20$, $n = 1$ million). In which case the number of comparison operations is $O(n)$. In the extreme cases where n is very large and k is very small then the $n\log n/k$ comparison steps will clearly dominate. However, if k is very small then little parallelism can be exploited.

Section 1 of this paper presents a model for parallel sorting on a linear array of processors. In Section 2 our parallel sorting algorithm is presented. This algorithm is extended to a simpler unidirectional ring communication model in Section 3. A detailed analysis of communication and comparison complexities is provided in Section 4. In Section 5 we discuss the dominance of communication complexity in realistic parallel sorting problems.

1. Model and Problem

We assume our computing system is composed of k processors connected in a linear chain (Figure 1). Processors can communicate only with adjacent processors, and each unit distance communication operation takes time t_c. Concurrent data movement is allowed as long as it is all in the same direction. This is how many systolic architectures operate [Fish83]. In each processing step t_p each of the k processors can perform one comparison operation. Complexities will be evaluated in terms of t_c an t_p time units.

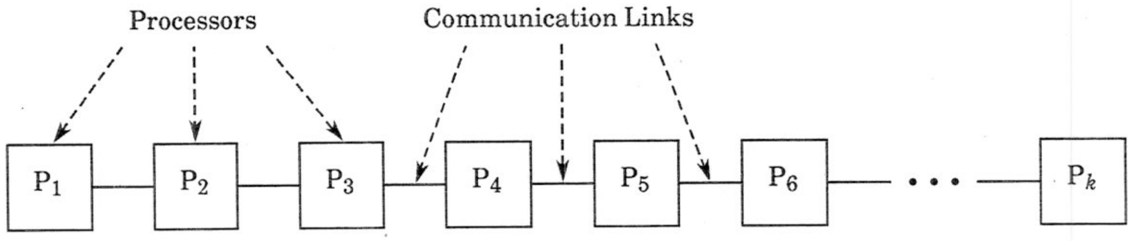

Figure 1 Linear Processor Array

We assume the machine is SIMD (Single Instruction stream Multiple Data stream [Flyn66] although the results still apply to tightly coupled MIMD (Multiple Instruction stream Multiple Data Stream), and loosely coupled distributed MIMD systems [Wilk79].

Initially the n unsorted elements are distributed evenly between the k processors so that each processor contains n/k elements in its local memory. The goal is to produce the final sorted list of n elements distributed evenly along k contiguous processors.

2. Algorithm

The algorithm is a modification of *Neighborhood Sort* [Baud78], which is a generalization of *Odd-Even Transposition Sort* (described in [Knut73]). *Odd-Even Transposition Sort* is an algorithm that sorts n elements with n processing steps on n processors. It works in the following manner.

Assume $S = s_1, s_2, s_3, s_4, \ldots, s_n$ is the sequence to be sorted where each s_i is an individual element. First the even pairs (s_j,s_{j-1}) (where $j = 2i$, $1 \leq i \leq \lfloor n/2 \rfloor$) are compared, and exchanged if $s_j > s_{j+1}$. Then the odd pairs (s_j,s_j+1) ($j = 2i$, $1 \leq i \leq \lfloor n\text{-}1/2 \rfloor$) are compared and exchanged. This process is repeated n times until all of the elements are sorted. A proof that this works is contained in [Knut73].

This algorithm can be generalized to merging n/k sorted lists instead of performing individual comparison-exchanges [Baud78]. Let $S = s_1,s_2, \ldots, s_{n/k}$ be a sequence of sorted lists where each s_i is a sorted sequence of size n/k. The algorithm proceeds in the same fashion except that the comparison-exchange steps are replaced by merge-splitting steps. During a merge splitting operation, two lists s_i and s_{i+1} are merged into a larger list which is then split into two halves. The lower half is assigned to s_i and the upper half is assigned to s_{i+1}. The proof that this results in a sorted list after k merge splittings can be found in [Knut73].

We have made the optimization of including two merged sequences at each processor node, thereby decreasing the communication complexity by factor of two. Our algorithm proceeds in the following manner:

Step 1: Each processor sorts the n/k unsorted elements which are initially contained in its local memory. The sorted list is then divided into two sorted lists (upper half and lower half) of size $n/2k$.

Step 2: All processors P_i transmit their upper list to processor P_{i+1}.

Step 3: The upper list form P_{i-1} is merged with the lower list from P_i. The resulting sorted sequence is divided into an upper and a lower sorted lists of size $n/2k$.

Step 4: All processors P_i transmit their new lower list to processor P_{i-1}.

Step 5: The lower list from P_{i+1} is merged with the upper list from P_i. The resulting sorted sequence is divided into an upper and a lower sorted list of size $n/2k$.

Steps 2 through 5 are repeated k times until the entire sequence S is sorted. Note that the boundary processors may be periodically inactive. An example for the case where $n = 12$ and $k = 3$ is shown in Figure 2. Note that that $k = 3$ iterations of steps 2 through 5 are not required in this example.

	Processor 1				Processor 2				Processor 3					
Start: Unsorted	12	3	8	10	4	7	2	11	9	6	1	5		
Step 1: Initial Sort	3	8	10	12	2	4	7	11	1	5	6	9		
Step 2: Transfer	3	8			10	12	2	4	7	11	1	5	6	9
Step 3: Merge-Split	3	8			2	4	10	12	1	5	7	11	6	9
Step 4: Transfer	3	8	2	4	10	12	1	5	7	11	6	9		
Step 5: Merge-Split	2	3	4	8	1	5	10	12	6	7	9	11		
Step 2: Transfer	2	3			4	8	1	5	10	12	6	7	9	11
Step 3: Merge-Split	2	3			1	4	5	8	6	7	10	12	9	11
Step 4: Transfer	2	3	1	4	5	8	6	7	10	12	9	11		
Step 5: Merge-Split	1	2	3	4	5	6	7	8	9	10	11	12		

Figure 2 Our Parallel Sorting Algorithm

3. Unidirectional Extension

This algorithm works even when communication is constrained to be unidirectional (unidirectional ring). This restriction on the algorithm simplify architectural design. A number of these unidirectional distributed multiprocessor [Wilk79] and systolic multiprocessor systems have been built [Fish83].

The data motion is easy to visualize. Simply modify Step 4 of our algorithm. Instead of transmitting the lower sequence back to the processor P_{i-1}, the same effect can be achieved by transmitting the upper sequence to processor $P_{(i+1) \bmod k}$. In both cases the lower sequence from processor P_i is united with the upper sequence from P_{i-1} for the next merge-splitting operation. The beginning of the sorted sequence will be cyclically shifted during the computation. A marker must be kept at the beginning of the sequence to insure that it is not merged with the end of the sequence. Figure 3 illustrates the unidirectional algorithm for the same initial unsorted sequence of Figure 2.

4. Analysis

For the purpose of analysis the algorithm can be divided into the initial sort, and the merge splitting operations. The initial sort can be performed using *Merge Sort* which has a running time of $(n/k)\log(n/k) - (n/k) + 1$ comparisons for a sort of n/k elements [Knut73]. So the execution time for the initial sort is:

$$((n/k)\log(n/k) - n/k + 1)t_p = ((n\log n)/k - (n\log k)/k - n/k + 1)\, t_p$$

The merge splitting operation involves steps 2 through 5. Steps 2 and 4 involve transfering lists of size $n/2k$. so each of these steps takes time $(n/2k)t_c$. Steps 3 and 5 involve merging two lists of size $n/2k$. Each of these takes time $(n/2k + n/2k - 1)t_p = (n/k-1)t_p$. Steps 2 through 5 are iterated k times so the total merge-splitting time is:

$$(2n - 2k)t_p + nt_c$$

The total execution time is then given by the expression:

$$((n\log n)/k - (n\log k)/k - n/k + 2n - 2k + 1)t_p + nt_c$$

Sequence Beginning Marker	Processor 1				Processor 2				Processor 3			
Start: Unsorted	$\boxed{12}$	3	8	10	4	7	2	11	9	6	1	5
Step 1: Initial Sort	$\boxed{3}$	8	10	12	2	4	7	11	1	5	6	9
Step 2: Transfer	6	9	$\boxed{3}$	8	10	12	2	4	7	11	1	5
Step 3: Merge-Split	6	9	$\boxed{3}$	8	2	4	10	12	1	5	7	11
Step 4: Transfer	7	11	6	9	$\boxed{3}$	8	2	4	10	12	1	5
Step 5: Merge-Split	6	7	9	11	$\boxed{2}$	3	4	8	1	5	10	12
Step 2: Transfer	10	12	6	7	9	11	$\boxed{2}$	3	4	8	1	5
Step 3: Merge-Split	7	6	10	12	9	11	$\boxed{2}$	3	1	4	5	8
Step 4: Transfer	5	8	7	6	10	12	9	11	$\boxed{2}$	3	1	4
Step 5: Merge-Split	5	6	7	8	9	10	11	12	$\boxed{1}$	2	3	4
Step 2: Transfer	3	4	5	6	7	8	9	10	11	12	$\boxed{1}$	2
Step 3: Merge-Split	3	4	5	6	7	8	9	10	11	12	$\boxed{1}$	2
Step 4: Transfer	$\boxed{1}$	2	3	4	5	6	7	8	9	10	11	12
Step 5: Merge-Split	$\boxed{1}$	2	3	4	5	6	7	8	9	10	11	12

Figure 3 Unidirectional Parallel Sorting Algorithm

This differs from *Neighborhood Sort* which has the same (asymptotically optimal) comparison complexity, but a communication complexity of $2nt_c$.

In the case where $k = \log n$, our algorithm takes time $3nt_p + nt_c$. while *Neighborhood Sort* takes time $3nt_p + 2nt_c$. Note that in general communication time t_c can be orders of magnitude longer than processing time t_p. So our algorithm is in most cases twice as fast.

In the case where $k = n/2$. Our algorithm takes time $nt_p + nt_c$ while *Neighborhood Sort* takes time $nt_p + 2nt_c$. This outperforms *Odd-Even Transposition Sort* which was the best known parallel algorithm for sorting n elements on a linear array of n processors and has a running time of $nt_p + 2nt_c$. Our algorithm requires the same computation time and half the communication time as *Odd-Even Transposition Sort.* And it does this using half as many processors! Note that storing two elements at each node does not change the computing model, since each processor must have two registers to perform comparison operations anyway.

Using more than $n/2$ processor to sort n elements merely introduces communication delays without improving computational performance. At most $n/2$ processors can concurrently perform comparison operations on n data items. This is because a processor requires at least two elements to perform a comparison.

Our algorithm can be adjusted to outperform *Odd-Even Transposition Sort* even if it is required that n processors be used. This means the n unsorted data items must start out uniformly distributed among the n processors, and the final sorted list must be uniformly distributed between n processors. First compress the data elements so that they fit onto $n/2$ processor each with two elements (Figure 4). Perform the sort operation. Then distribute the sorted elements back out to the n processors. The

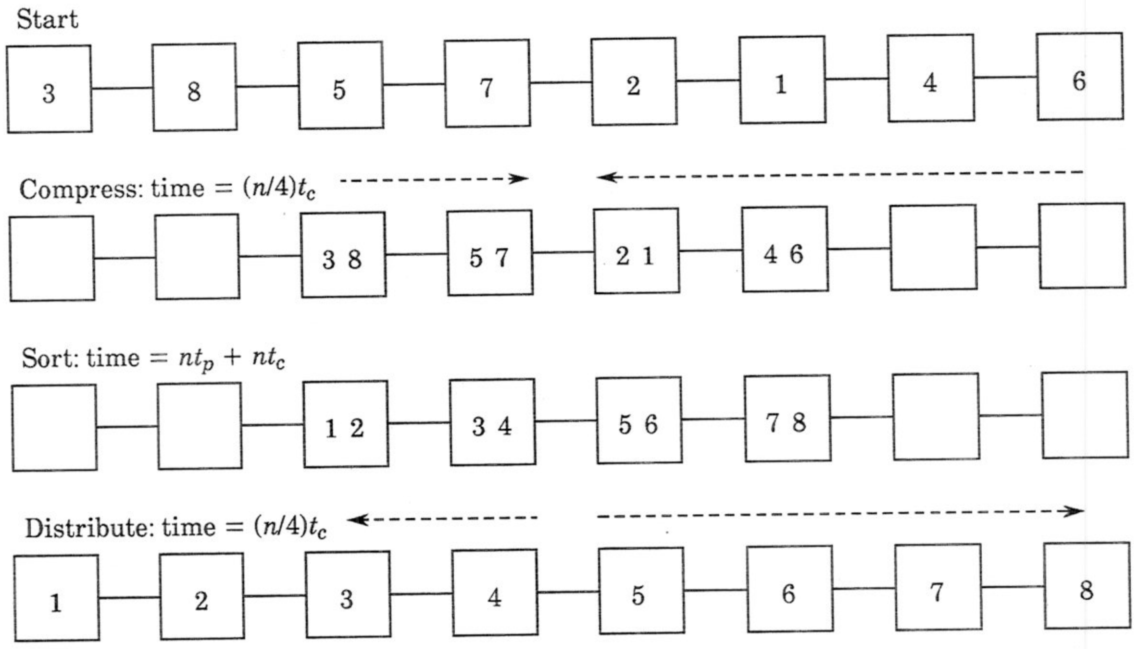
Start
3
8
5
7
2
1
4
6
Compress: time = $(n/4)t_c$
3 8
5 7
2 1
4 6
Sort: time = $nt_p + nt_c$
1 2
3 4
5 6
7 8
Distribute: time = $(n/4)t_c$
1
2
3
4
5
6
7
8

Figure 4 Parallel Sorting when $k = n$

compression and distribution steps each take time $(n/4)t_c$ so the total execution time is $nt_p + (3/2)nt_c$.

5. Communication Dominance

We have noted that communication time can be the dominant factor in parallel sorting time. When is this the case? Communication time will dominate when $nt_c > (n\log n/k)t_p$. Solving for k we get:

$$k > \log n(t_p/t_c)$$

This is a very small number. Suppose that one communication step takes ten times as long as one comparison step (e.g. one microsecond versus 100 nanoseconds). In some computing systems, communication time can take several orders of magnitude more time than computational steps [Wilk79]. Also suppose $n = 10^9$. If $k > 3$, communication time will dominate. If k is three or less, very little parallelism is being employed to solve the problem anyway. This example illustrates why communication complexity, which is often less emphasized than computational complexity, should be weighed with equal if not greater importance than computational complexity in parallel algorithm analysis.

Conclusions and Future Research

In this paper we have described a parallel sorting algorithm for a linear processor array which reduces the communication complexity of the best known algorithm by a factor of two while maintaining the same asymptotically optimal number of computational steps. This result is important because communication time dominates computation time in realistic sorting problems on this common parallel architecture. The algorithm works even if only unidirectional communication between processors is allowed (unidirectional ring).

In future research we will extend these results to two, three, and higher dimensional processor arrays and other regular interconnection schemes [Thom78] [Nass79].

Acknowledgments

We would like to thank Robert Sedgewick and Toshio Nakatani for their valuable comments and suggestions.

References

[Baud78] G. Baudet, D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers", *IEEE Transactions on Computers*, Volume C-27, Number 1, January 1978, pp. 84-87.

[Farb75] D. J. Farber, "A Ring Network". *Datamation*, Volume 21, Number 2, February 1975, pp. 44-46.

[Fish83] A. L. Fisher, H. T. Kung, L. M. Monier, "Architecture of the PSC: A Programmable Systolic Chip", *Proceedings of the Tenth Annual Symposium on Computer Architecture*, 1983, pp. 48-53.

[Flyn66] M. J. Flynn, "Very High Speed Computing Systems", *Proceedings of the IEEE*, Volume 54, Number 12, December 1966, pp. 1901-1909.

[Hori86] S. Horiguchi, Y. Shigei, "A Parallel Sorting Algorithm for a Linearly Connected Multiprocessor System", *Proceedings of the International Conference on Distributed Computing Systems*, May 1986, pp. 111-118.

[Knut73] D. E. Knuth, " The Art of Computer Programming - Sorting and Searching", Volume 3, Addison-Wesley, Reading Ma, 1973.

[Kung82] H. T. Kung, "Why Systolic Architectures?", *Computer Magazine*, Volume 15, Number 1, January 1982, pp. 37-46.

[Nass79] D. Nassimi, S. Sahni, "Bitonic Sort on a Mesh Connected Parallel Computer", *IEEE Transactions on Computers*, Volume C-28, Number 1, January 1979, pp. 2-7.

[Thom78] C. D. Thompson, H. T. Kung, "Sorting on a Mesh Connected Parallel Computer", *Communications of the ACM*, Volume 20, Number 4, April 1977, pp. 263-271.

[Wilk79] M. V. Wilkes, "The Cambridge Digital Communication Ring", *Proceedings of the LACN Symposium*, May 1979, pp. 47-61.