Improved Sorting Algorithms for Parallel Computers

Arvin Park

K. Balasubramanian

Department Computer Science
Princeton University
Princeton, New Jersey 08544

CS-TR-048-86

August 12, 1986

Abstract

We make observations that improve processor utilization and decrease communication overhead for several parallel sorting algorithms. These lead to constant factor improvements on the best previous parallel sorting bounds for both mesh-connected and linearly connected parallel architectures [Thom77] [Baud78] (Previous bounds were within a constant factor of optimal.). These improved bounds are achieved using fewer processors with greater processor utilization.

Keywords and Phrases - *linearly connected parallel computer, mesh-connected parallel computer, multiprocessor, parallel sorting, SIMD, systolic array.*

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.

1. Introduction

The advent of Systolic Arrays has generated great interest in both linear- and mesh- connected parallel computer architectures [Kung82]. A number of these systems have been built [Fish83] [Syma82], and a great deal of work has been devoted to algorithm development for these architectures [Baud78] [Kung82] [Thom77].

Many of these algorithms have been developed assuming that the number of processors equals the number of data elements. Although these algorithms have often been generalized to accommodate more than one data element per processor [Baud78], they often suffer from the inefficiencies of their original assumptions.

In section two, we demonstrate how the assumption of one data element per processor causes inefficiencies for *Odd-Even Transposition Sort* [Knut73]. The previous algorithm sorts n elements on n processors in time $2nt_r + nt_c$ (where t_r is the time for one routing step, and t_c is the time for one comparison operation). We perform the same sort in time $nt_r + nt_c$ using $n/2$ processors .

Section three shows how these inefficiencies have propagated to *Neighborhood Sort* [Baud78] which does not assume one element per processor, and is the best known sorting algorithm for a linearly connected parallel computer. We improve this bound from $2nt_r + ((n\log n)/k + n)t_c$ (k = number of processors n = number of elements) to $nt_r + ((n\log n)/k + n)t_c$ [Park86].

In section four we apply the same principles to improve *S^2-Merge Sort* [Thom77] which is the best known algorithm for sorting on a mesh connected parallel computer. We improve this algorithm for sorting n^2 elements from the previous running time of $(6n + O(n^{2/3}\log n))t_r + (n + O(n^{2/3}\log n))t_c$ using n^2 processors, to an improved time of $(4n + O(n^{2/3}\log n))t_r + (n + O(n^{2/3}\log n))t_c$ using $n^2/2$ processors. We also show that it is possible to use $n^2/4$ processors without significant performance degradation: $(6n + O(n^{2/3}\log n))t_r + (2n + O(n^{2/3}\log n))t_c$.

A note on diagram symbols: We will use the following conventions to indicate data movement.

Diagram Symbols:	$\longleftrightarrow$	$\longrightarrow$	$\xrightarrow{C}$
	Exchange	Move	Comparison-exchange

Double arrows represent data exchanges. Single arrows represent data transfers. A single arrow with a C next to it represents a comparison-exchange operation.

2. Odd-Even Transposition Sort

2.1 Model

We assume a processor model which is very much like several existing linearly connected processors [Wilk79] [Hori86] [Farb75]. This model consists of a chain of linearly connected processing elements. We assume an SIMD machine [Flyn66], this means an instruction is broadcast to all processors but is only executed by the processors specified. Individual processors may elect not to perform an operation if the result of a local test operation is negative. (Systolic arrays are MIMD machines, but they can be restricted to operate as an SIMD machines).

2.2 Algorithm

Odd-Even Transposition Sort (described in [Knut73]) is the best known algorithm for sorting n element on an n processor array. It starts with an unsorted list of n elements $S = s_1, s_2, \ldots, s_n$ and proceeds in the following manner: (i) The even pairs (s_{2i-1}, s_{2i}) $i = 1,2, \ldots, \lfloor n/2 \rfloor$ are compared and exchanged if $s_{2i-1} > s_{2i}$. (ii) The odd pairs (s_{2i}, s_{2i+1}) $i = 1,2, \ldots, \lceil (n/2)-1 \rceil$ are compared and exchanged if $s_{2i} > s_{2i+1}$. Steps (i) and (ii) are repeated n/2 times until the entire list is sorted (A proof that this works is contained in [Knut73]).

This algorithm has been adapted to run on n linearly connected processors [Thom77]. Each element s_i is dedicated to a processor p_i. Step (i) can be accomplished by first moving the odd elements $\{s_{2i+1}, i = 0,1,\ldots,\lceil n/2-1 \rceil\}$ to the next higher even processors, performing a comparison-exchange, then moving the smaller element back to the odd processor. Step (ii) can be accomplished by moving the even elements $\{s_{2i}, i = 1,\ldots,\lfloor n/2 \rfloor\}$ to the next higher odd processor, performing a comparison exchange, then moving the smaller element back to the even processor. Both steps (i) and (ii) require two routing steps and one comparison step. They are both repeated $n/2$ times so the total execution time is $n/2(2t_r + t_c + 2t_r + t_c) = 2nt_r + nt_c$.

Steps (i) and (ii) are illustrated in Figure 1. Note that at most half of the processors are active at any given time. There is a good reason for this. A compare operation requires two operands. Therefore only $n/2$ processors can be used at any given time to perform simultaneous comparison operations on n data items. Also note that at most half the communication channels are active at any given time.

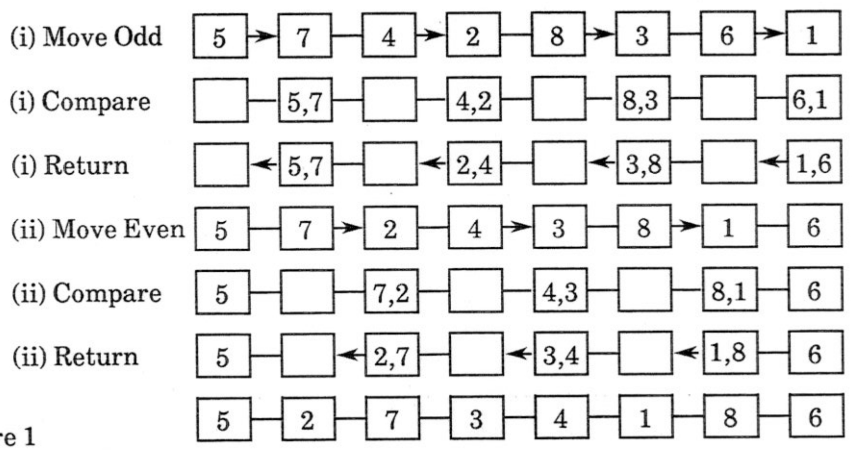


Figure 1

Odd-Even Transposition Sort can be improved by using half as many processors. Initially place two elements at each of $n/2$ processors (Note that each processor must have two registers anyway). Now half the communication steps are avoided, and all $n/2$ processors can simultaneously perform comparisons. Steps (i) and (ii) for *Improved Odd-Even Transposition Sort* are illustrated in Figure 2. Note that a boundary condition arises where either the end processor must be able to accommodate three elements, or an extra processor must be added on the boundary.

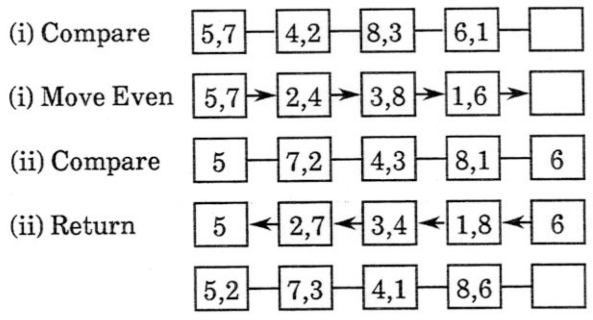


Figure 2

Steps (i) and (ii) now both require only one routing step and one comparison step each. So the total execution time becomes $n/2(t_r + t_c + t_r + t_c) = nt_r + nt_c$. The routing time has been decreased by a factor of two and the comparison time remains the same.

Using n processors for *Odd-Even Transposition Sort* introduced communication overhead in addition to wasting processors. This inefficiency has propagated to *Neighborhood Sort* (a generalization of *Odd-Even Transposition Sort*) which is the best known algorithm for sorting on a linearly connected parallel computer. We show how *Neighborhood Sort* can be improved in the next section.

3. Neighborhood Sort

For *Neighborhood Sort* we assume a computing model that is identical to the model for *Odd-Even Transposition Sort* except that each processor can store a large number ($>2n/k$) of elements in its local memory.

Neighborhood Sort [Baud78] is a generalization of *Odd-Even Transposition Sort.* Instead of performing comparison exchange operations between pairs of elements, Merge-splitting operations are performed between pairs of sorted lists of elements.

3.1 Algorithm

During a merge-splitting operation of between two lists $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ... , b_n$. The two lists A and B are merged into list $C = c_1, c_2, ... , c_{2n}$. List C is then split into two halves $c_1, c_2, ... , c_n$ and $c_{n+1}, c_{n+2}, ... , c_{2n}$. The upper half is assigned to list B and the lower half to list A.

In *Neighborhood Sort*, n/k unsorted elements are initially contained on each of k linearly connected processors. The algorithm first sorts these local lists of size n/k. It then proceeds in much the same manner as *Odd-Even Transposition Sort* except that merge-splitting operations are substituted for comparison exchange steps. A proof that this works can be found in [Knut73] (Merging Network Theorem).

We have been able to improve performance of this algorithm in the same manner as we have for *Odd-Even Transposition Sort.* We place two sorted sequences at each processor node instead of one. Our algorithm proceeds as follows:

(i) 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 $n/2k$.

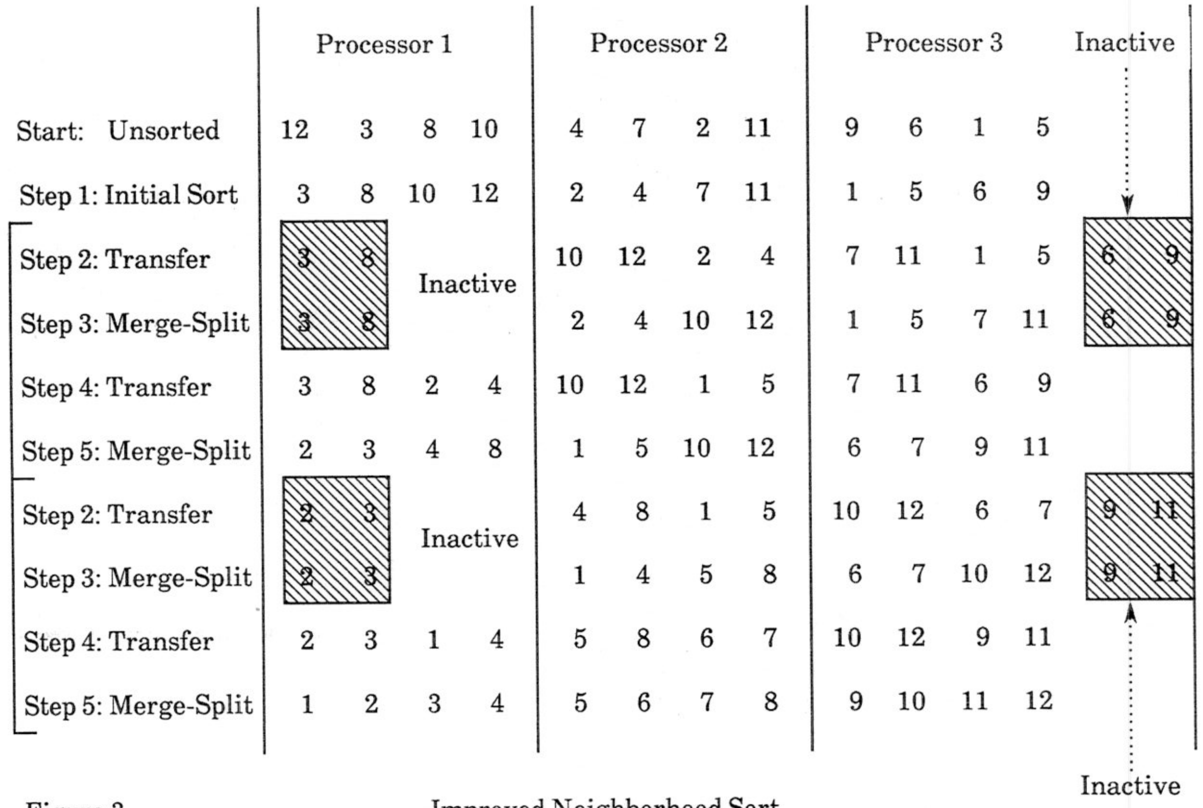
(ii) All processors P_i transmit their upper list to processor P_{i+1}.

(iii) 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 $n/2k$.

(iv) All processors P_i transmit their new lower list to processor P_{i-1}.

(v) 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 (ii) through (v) are repeated k times until the entire sequence S is sorted. An example for the case where $n = 12$ and $k = 3$ is shown in Figure 3. Note that that $k = 3$ iterations of steps 2 through 5 are not required in this example.



	Processor 1				Processor 2				Processor 3				Inactive	
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	Inactive		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	Inactive		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 3 Improved Neighborhood Sort

3.2 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:

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

The merge splitting operation involves steps (ii) through (v). Steps (ii) and (iv) both involve transfering lists of size $n/2k$ which takes time $(n/2k)t_r$. Steps (iii) and (v) require merging two lists of size $n/2k$ which takes time $(n/2k + n/2k - 1)t_c = (n/k-1)t_c$. Steps (ii) through (v) are iterated k times so the total merge-splitting time is:

$$(2n - 2k)t_c + nt_r$$

The total execution time is then given by the expression:

$$((n\log n)/k - (n\log k)/k - n/k + 2n - 2k + 1)t_c + nt_r$$

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

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 $k = 20$, $n = 2^{20}$). 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.

Note that each processor is fully utilized in the initial sorting phase. So equivalent performance in the initial sort can not be achieved using fewer processors.

4. Sorting on a Mesh-Connected Parallel Computer

We have been able to improve the *S^2-Merge Sorting Algorithm* (due to Thompson and Kung [Thom77]) which is the best known sorting algorithm for a mesh-connected parallel computer. This algorithm sorts n^2 elements on an $n \times n = n^2$ mesh connected parallel computer in time $(6n + O(n^{2/3}\log n))t_r + (n + O(n^{2/3}\log n))t_c$. The improved algorithm performs the same sort in time $(4n + O(n^{2/3}\log n))t_r + (n + O(n^{2/3}\log n))t_c$ using half as many ($n/2 \times n = n^2/2$) processors.

The original $n \times n$ array is compressed row-wise so that pairs of adjacent row elements are stored at each processor. The resulting mesh consists of n rows of $n/2$ processors. This arrangement increases processor utilization and improves performance on Thompson and Kung's algorithm.

4.1 Model

We assume a processor model which is very much like the Illiac IV, and several existing systolic computers [Barn68] [Kung83] [Syma82]. Our model consists of an two dimensional mesh sized $n/2 \times n$ of processing elements. Each processor is connected to its four vertically and horizontally adjacent neighbors (of course boundary processors will have fewer connections) (Figure 4a). The control structure is SIMD

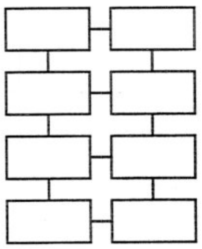


Figure 4a: Mesh Connected Processor

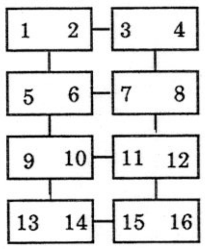


Figure 4b: Row Major Order

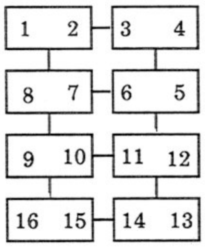


Figure 4c: Snake-Like Row Major Order

We assume that each processor initially contains two unsorted elements in its local memory (The original algorithm assumed one element per processor). At the end of the computation, the elements are sorted in snake-like row major ordering [Thom77]. Figures 4b and 4c provide examples of row major and snake-like row major orderings. Snake-like row major ordering can be obtained from row major ordering by reversing the even rows. Note that it is possible to perform this transformation using n routing operations.

4.2 2-Way Odd Even Merge

This parallel merging algorithm is based on *Batcher's Odd-Even Merge* of two sorted sequences $A = a_1, a_2, \dots, a_n$ and $B = b_1, b_2, \dots, b_n$. First the odd members of sequences A and B ($a_1, a_3, \dots, a_{2i+1}$ and $b_1, b_3, \dots, b_{2i+1}$) are merged (recursively). Next the even sequences ($a_2, a_4, \dots, a_{2i}$ and $b_2, b_4, \dots, b_{2i}$) are merged (recursively). The resulting sequences are interleaved and a comparison exchange operation produces a single sorted sequence.

Figure 5 illustrate how *Odd-Even Merge* can be performed on a set of linearly connected processors. Steps (i) and (iii) involve unshuffle and shuffle operations. Which can be performed in time $(k/2)t_r$ for k elements stored on $k/2$ processors. This is a factor of two improvement from the k-1 time required to perform the same operations for k elements on k processors [Thom77]. Figure 6 illustrates the shuffle operation for the case of 8 elements on 4 processors (unshuffle is merely the inverse operation).

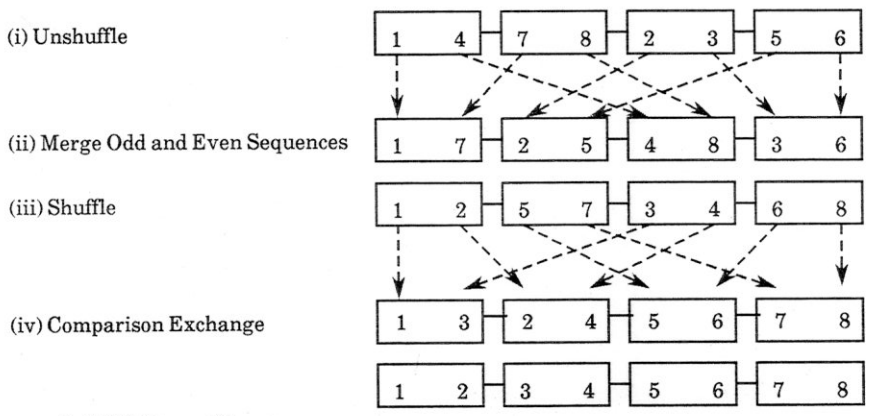


Figure 5: Odd-Even Merge

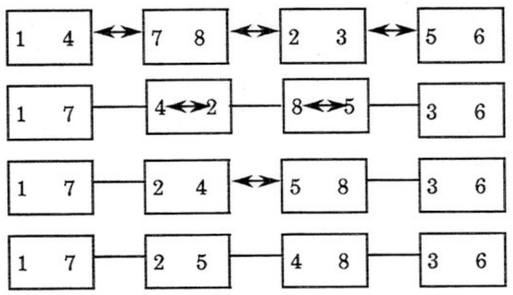


Figure 6: Unshuffle

Odd-Even Merge can be extended to a rectangular mesh of processors. Let M(j,k) be an algorithm that takes as input two sorted k/2 x j subarrays and outputs a sorted k x j array. Let us first examine the case of M(j,2). This algorithm proceeds as follows and is illustrated for the case where j = 4 in Figure 7:

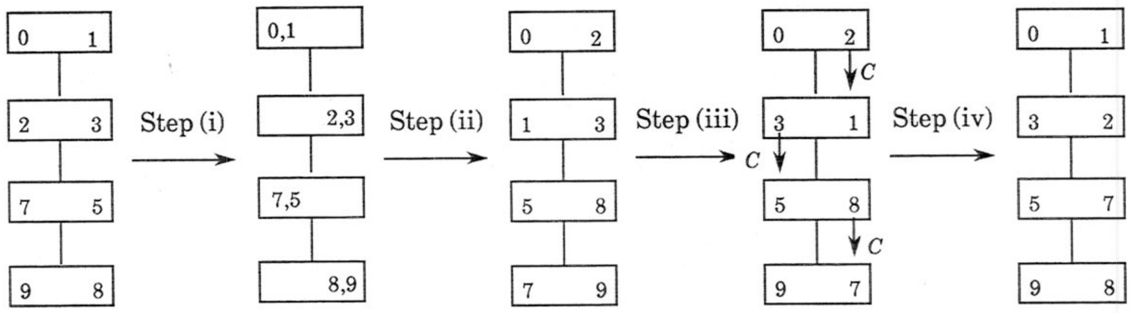


Figure 7: Odd Even Merge M(j,2)

(i) Move all odd elements to the left and evens to the right.

(ii) Sort each column using Odd-Even Transposition Sort.

(iii) Interchange on even rows.

(iv) Comparison-exchange each even with the next odd.

Note that If we place two adjacent row element on each processor then steps (i) and (iii) require no time since the both elements are contained on the same processor. The comparison exchange operation simply involves one compare (time $= t_c$). Sorting two array columns on one column of processors is the tricky part. We use *Odd-Even Transposition Sort* to sort each of the two array columns. If the sort on the left column is delayed in phase by one comparison exchange step, then it is possible to sort the two columns of j elements on j processors in almost the same amount of time as it takes to sort a single column of j elements on j processors. This sorting technique illustrated in Figure 8. The running time for this sort is $(2j + 2)t_r + (j + 1)t_c$ so the total running time for $T(j,2) = (2j + 2)t_r + (j + 2)t_c$.

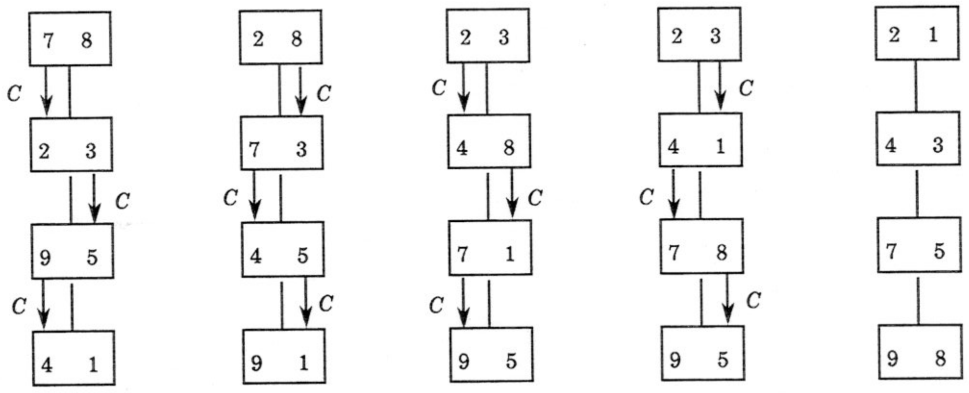


Figure 8: Sorting two array columns on a single column of processors

The recursive algorithm to sort M(j,k) works in the following way (refer to Figure 9):

(i) Single interchange elements so that columns contain either all even or all odd elements.

(ii) Unshuffle all rows in parallel

(iii) Recursively merge by calling M(j,k/2) on each half of the array.

(iv) Shuffle all rows in parallel

(v) Single interchange on even rows

(vi) Expensive comparison-exchange every even with the next odd.

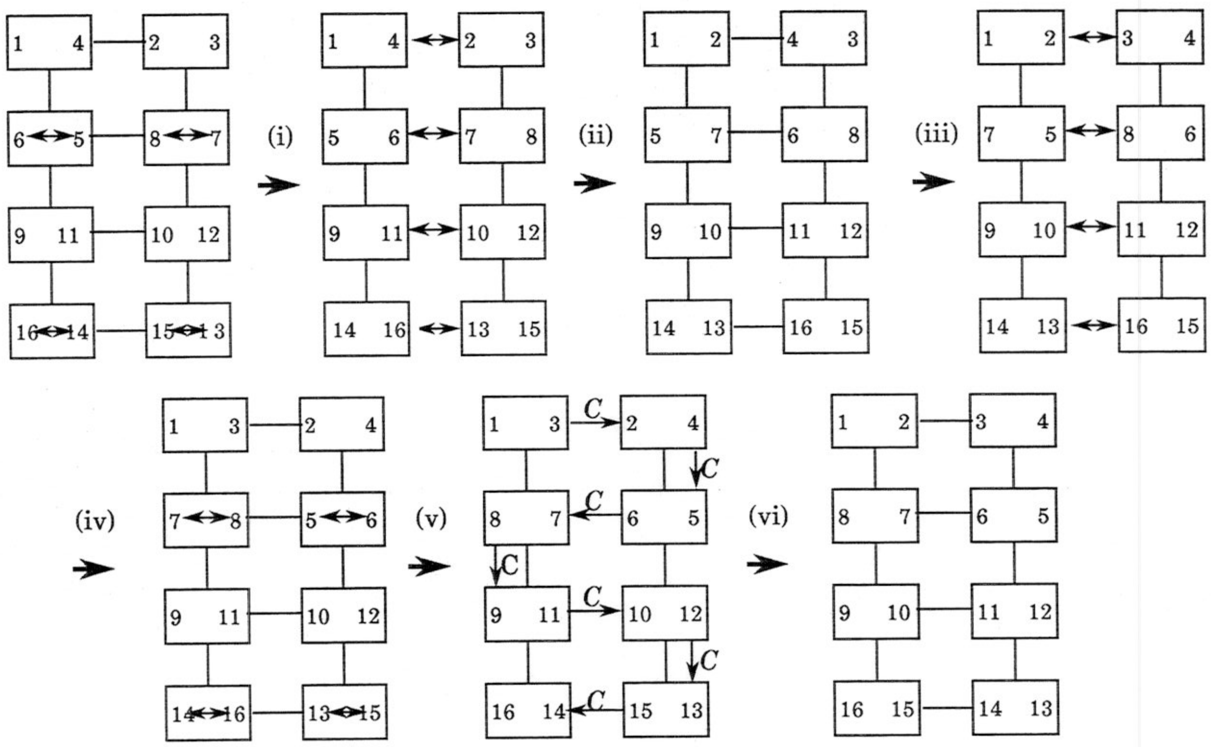


Figure 9: 2 Way Odd-Even Merge

Steps (i) and (v) require no time since both elements are contained on the same processor. Steps (ii) and (iv) require time $(k/2)t_r$ each by the method described earlier in the paper. Step (vi) requires time $4t_r + t_c$. So the total execution time for M(j,k) is given by:

$$\begin{aligned} T(j,k) &= (k + 4)\, t_r + t_c + M(j, k/2) \\ &= (2k + 2j + 4\log k + 4)t_r + (j + \log k + 3)t_c \end{aligned}$$

4.3 2s-Way Merge

This algorithm is a variant of *2-way Odd-Even Merge*. It takes as input $2s$ arrays of size j/s x $k/2$ which fit into a j x $k/2$ region of processors and are sorted in snake-like row major order. It produces a sorted array of size j x k. The algorithm is a simple modification of the 2-way Odd-Even Merge Sort of the last section. Simply replace steps (i) and (vi) by:

(i) If j > s, Single interchange on even rows so that each array column contains all even or all odd entries. If j = s, do nothing since they are already divided.

(vi) Perform the first 2s-1 comparison exchange operations of *Odd-Even Transposition Sort.*

A proof that this algorithm works is based on the 0-1 principle [Knut73] [Thom77]. This states that if a network sorts all sequences of 0's and 1's, then it will sort any arbitrary sequence from a linearly ordered set. Note that if the input sorted arrays were composed entirely of 0's and 1's, then in the worst case s 0's would be out of place after steps (iii) (iv) and (v). 2s-1 comparison exchange operations are required to move these remaining 0's into place.

Step (vi) requires $s(4t_r + t_c) + (s\text{-}1)t_c = 4st_r + (2s\text{-}1)t_c$ time. So the total running time of this algorithm will be:

$$\begin{aligned} T'(j,k,s) &= (k+4s)t_r + (2s\text{-}1)t_c + T'(j,k/2,s) \\ &= (2j + 2k + 4s(\log k) + O(s+\log k))t_r + (j + 2s(\log k) + O(s+\log k))t_c \end{aligned}$$

If $s = 2$ then a merge sort can be produces that has running time:

$$\begin{aligned} S'(n,n) &= S'(n/2,n/2) + T'(n,n,2) \\ &= (8n + O(\log^2 n))t_r + (2n + O(\log^2 n))t_c \end{aligned}$$

This sorts in linear time, and it is an improvements over Thompson and Kung's 2s-Way Merge Sort which runs in time $(12n + O(\log 2n))t_r + (2n + O(\log 2n))t_c$, but this running time can be halved by using an more complicated algorithm due to Thompson and Kung.

4.4 S^2-Way Merge Sort

This algorithm is also a variant of *2-way Odd-Even Merge*. It takes as input s^2 arrays of size j/s x k/s which fit into a j x $k/2$ region of processors and are sorted in snake-like row major order. It then produces a sorted array of size j x k. Simply replace steps (i) and (vi) in *2-Way Odd-Even Merge* by:

(i) If j > s, Single interchange on even rows so that each array column contains all even or all odd entries. If j = s, do nothing since they are already divided.

(vi) Perform the first s^2-1 comparison exchange operations of *Odd-Even Transposition Sort.*

The s^2-1 comparison exchange steps are are required to move a possible $s^2/2$ 0's from the right half to the left half of the array after shuffling.

The case of $M(j,s,s)$ can be performed by:

(i) $\log s/2$ 2-Way Odd-Even Merges: $M(j/s,2), M(j/s, 4), \dots, M(j/s,k/2)$
(ii) One 2s-Way Merge: $M'(j,s,s)$

So the running time for $T''(j,s,s) = (2j + O(s + j/s)\log s)t_r + (j + O(s + j/s)\log s)t_c$. This leads to a total merge time of :

$$\begin{aligned} T''(j,k,s) &= (k + 2s^2 + O(1))t_r + (s^2 + O(1))t_c + T''(j,k/2,s) \\ &= (2k + 2j + 2s^2\log(k/s) + O((s + j/s)\log s) + \log k)t_r \\ &\quad + (j + s^2\log(k/s) + O((s + j/s)\log s) + \log k)t_c \end{aligned}$$

If $s = n^{1/3}$, then we can produce a sorting algorithm from this merge with the following running time:

$$\begin{aligned} S''(n,n) &= S''(n^{2/3},n^{2/3}) + T''(n,n,n^{1/3}) \\ &= (4n + O(n^{2/3}\log n))t_r + (n + O(n^{2/3}\log n))t_c \end{aligned}$$

This is a constant factor improvement over the $(6n + O(n^{2/3}\log n))t_r + (n + O(n^{2/3}\log n))t_c$ running time of Thompson and Kung's algorithm and this is accomplished using half as many processors ($n^2/2$).

4.5 $n^2/4$ Processors

Note that $n^2/2$ processors can be fully utilized in some portions of *s^2-Way Merge Sort*. So we would expect some performance penalty in using even fewer processors. We can in fact reduce the number of processors to $n^2/4$ and still maintain a running time of $(6n + O(n^{2/3}\log n))t_r + (2n + O(n^{2/3}\log n))t_c$. The processor model will have to be slightly modified so that each processor can store at least four elements.

We map our original n x n problem array onto a $n/2$ x $n/2$ processor mesh by a placing 2 x 2 sub-arrays at each processor node. This arrangement affects the running time of several critical portions of the *s^2-Way Merge Sort* algorithm. The row-wise unshuffle and shuffle operations (steps (ii) and (iv)) require twice the number of communication steps since the number of communication channels in decreased by a factor of two. These now each take time kt_r. Comparison exchange operations can take twice as long, but this does not affect the asymptotic running time.

Besides the shuffle and unshuffle operations, the only other asymptotically significant portion of the algorithm is the two-column sort. A sort of two columns of size k can be performed on a single column of $k/2$ processors. Sort one column and then the other using our *Improved Odd-Even Transposition Sort* (which takes time $kt_r + kt_c$. This will require twice the sorting time of a single column of k elements on on a column of $k/2$ processors. The total two column sorting time then becomes $2kt_r + 2kt_c$. This has the effect of doubling the number of comparison operations in the 2s-Way Merge algorithm.

The new running time M'''(j,s,s) now becomes:

$T'''(j,s,s) = (2j + O(s + j/s)\log s)t_r + (2j + O(s + j/s)\log s)t_c$. This leads to a total merge time of:

$$\begin{aligned}
T'''(j,k,s) &= (2k + 2s^2 + O(1))t_r + (s^2 + O(1))t_c + T'''(j,k/2,s) \\
&= (4k + 2j + 2s^2\log(k/s) + O((s + j/s)\log s) + \log k)t_r \\
&\quad + (2j + s^2\log(k/s) + O((s + j/s)\log s) + \log k)t_c
\end{aligned}$$

If $s = n^{1/3}$, we can produce a sorting algorithm from this merge with the following running time:

$$S'''(n,n) = S'''(n^{2/3},n^{2/3}) + T'''(n,n,n^{1/3})$$

$$= (6n + O(n^{2/3}\log n))t_r + (2n + O(n^{2/3}\log n))t_c$$

It is not surprising the number of comparison steps were doubled since the all $n^2/2$ processors were utilized during comparison portions of the $n^2/2$ processor algorithm.

Conclusion

The assumption that the number processors equals the number of elements to be sorted may have facilitated the development of parallel sorting algorithms. However, this assumption has lead to added communication overhead and poor processor utilization. Using these observations, we have been able to improve several sorting algorithms by a constant factor in running time using fewer processors with greater processor utilization.

Future research will extend these results to other classes of parallel algorithms.

Acknowledgments

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

References

[Barn68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, R. A. Stokes, "The Illiac IV Computer", *IEEE Transactions on Computers*, Volume C-17, Number 8, August 1968, pp. 746-757.

[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.

[Park86] A. Park, K. Balasubramanian, "Optimal Parallel Sorting on a Linear Processor Array", Technical Report #CS-TR-048-86, Department of Computer Science, Princeton University, Princeton, New Jersey, 08544, July 1986.

[Syma81] J. J. Symanski, "Systolic Array Processor Implementation", *Proceedings of the SPIE*, Volume 341, Real-Time Signal Processing V, Society of Photo-Optical Instrumentation Engineers, 1982. pp. 2-7.

[Thom77] 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.