Parallel Computing 14 (1990) 89–97
North-Holland

Optimal parallel merging and sorting algorithms using $\sqrt{N}$ processors without memory contention

Jau-Hsiung HUANG

Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.

Leonard KLEINROCK

Computer Science Department, University of California, Los Angeles, Los Angeles, CA, USA

Received August 1989
Revised November 1989

Abstract. A *multi-way parallel merging algorithm* is described to merge two sorted lists each with size N on a shared-memory parallel system. The structure of this algorithm is very regular and highly parallel. It is shown that using P processors, the time complexity of this algorithm is $O(N/P)$ when $N \geqslant P^2$, which is known to be optimal. This approach for parallel merging leads to a *multi-way parallel sorting algorithm* with time complexity $O(N \log N)/P)$ when $N \geqslant P^2$. Clearly this is also optimal. In addition, these two algorithms do not require reading from or writing into the same memory location simultaneously, hence they can be applied on a EREW machine. In cases when $N < P^2$, we recursively apply this merging algorithm to show that for $P = N^{(2^k-1)/2^k}$, the complexities of the merging algorithm and the sorting algorithm are $O(3^k N/P)$ and $O(3^k(N \log N)/P)$ respectively.

Keywords. Merging, sorting, shared-memory multiprocessor, complexity analysis, multi-way merging and sorting.

1. Introduction

The performance of a parallel algorithm is normally measured in terms of the number of processors, P, used and the time complexity, $T(N)$, required. It is well known that a parallel merging algorithm is optimal if $O(P \cdot T(N)) = O(N)$. Similarly, a parallel sorting algorithm is optimal if $O(P \cdot T(N)) = O(N \log N)$. In this paper we denote log as the logarithm based on 2. There are a variety of algorithms in which parallel merging and sorting are designed [1,4,7,9,10,12–15]. Taxonomies of parallel sorting algorithms can be found in [2,3,11].

In a shared-memory parallel system, we assume that there are P processors sharing a global memory space. Each processor can read from or write into any memory location. Depending on whether concurrent read from or concurrent write into a memory is allowed, shared-memory parallel systems are categorized into the following four groups:

(a) CRCW (Concurrent Read Concurrent Write) machines: both concurrent read from and concurrent write into a memory location by more than one processor is allowed.

(b) CREW (Concurrent Read Exclusive Write) machines: concurrent read from but not concurrent write into a memory location by more than one processor is allowed.

(c) ERCW (Exclusive Read Concurrent Write) machines: concurrent write into but not concurrent read from a memory location by more than one processor is allowed.

(d) EREW (Exclusive Read Exclusive Write) machines: neither concurrent read from nor concurrent write into a memory location by more than one processor is allowed.

It has been shown that every CRCW algorithm that requires time $T(N)$ (where N is the size of the input or output data) using P processors can be transformed into an EREW algorithm which requires time $O(T(N)\log N)$ still using P processors [8].

An early work in parallel merging and sorting was the sorting circuit due to Batcher [5] with $O(P \cdot T(N)) = O(N \log^2 N)$. This construction is an implementation of odd-even mergesort. Later, [7,10,14,15] presented optimal merging algorithms for CRCW and CREW machines. Recently, [1] gave a sorting algorithm that used N processors in $O(\log N)$ time by constructing an $O(\log N)$-level sorting circuit. More recently, [4] described an optimal merging and sorting algorithm for $P \leqslant N/\log^2 N$ running on an EREW machine. To compare the sorting algorithm in [4] with our algorithm, both sorting algorithms are mergesort algorithms and the concurrency occurs during the merging phase. However, the merging algorithm in [4] involves a *selection* algorithm to find the appropriate sublists for each processor to merge; while in our merging algorithm, merging two lists is simply done by three merging runs and each merging run employs all processors concurrently. Our algorithm is more straightforward in idea and implementation while the algorithm in [4] uses more processors to achieve optimal complexity.

Batcher's merging exchange algorithm [5] first separated each of the two lists into odd and even sublists. It then merges both of the odd sublists and both of the even sublists concurrently. This is why it is an implementation of the odd-even merging algorithm. Hence, this algorithm can be regarded as a two-way merging algorithm since it only parallelly merges the odd sequences and the even sequences simultaneously. Batcher's algorithm does not achieve the optimal complexity. We extend this two-way merging approach into a *multi-way* merging algorithm which divides each of the two lists into P sublists and uses P processors to merge these $2P$ sublists from both lists concurrently to obtain an optimal time complexity when $N = P^2$. In this paper, we first describe a *multiway parallel merging algorithm* which achieves an optimal time complexity when $N \geqslant P^2$. The same approach is used to develop an optimal *multi-way parallel sorting algorithm* when $N \geqslant P^2$. The contribution of this paper is the regularity and the simplicity of these algorithms.

If $N < P^2$, it is shown that by recursively applying the multi-way merging algorithm, we achieve a merging algorithm with time complexity $O(3^k N/P)$ using $P = N^{(2^k-1)/2^k}$ processors for $k \geqslant 1$. Similarly, the sorting algorithm has a time complexity of $O(3^k(N \log N)/P)$ using $P = N^{(2^k-1)/2^k}$ processors for $k \geqslant 1$.

Section 2 describes the multi-way parallel merging algorithm and Section 3 describes the multi-way parallel sorting algorithm. A conclusion is given in the last section.

2. Multi-way parallel merging algorithm

The input of the multi-way parallel merging algorithm is two sorted lists with equal size and are denoted as L_1 and L_2. For ease of explanation, we assume these $2N$ elements are all distinct. However, the same algorithm still applies when these $2N$ elements are allowed to have the same value. Further, we assume the memory addresses occupied by L_1 is from 1 to N and the memory addresses occupied by L_2 is from $N+1$ to $2N$. We would like to merge L_1 and L_2 into one sorted list using P processors. We first describe the algorithm when $N = P^2$. We then extend this approach to cases when $N > P^2$ and $N < P^2$. To begin with, we number all the processors and denote P_i as the ith processor ($1 \leqslant i \leqslant P$).

2.1. Cases when $N = P^2$

This algorithm is separated into four steps. Follows we explain the algorithm in each step. In Step 1, we divide L_1 into P sorted sublists in such a way that each sublists contains elements which are P positions apart and each sublist contains P elements. The ith sublist ($1 \leqslant i \leqslant P$) contains elements at locations i, $i+P$, $i+2P,\ldots, i+(N/P-1)P$. Similarly, L_2 is divided into P sublists in the same way. Therefore, the ith sublist in L_2 contains elements at locations $N+i$, $N+i+P$, $N+i+2P,\ldots, N+i+(N/P-1)P$.

In Step 2, we assign P_i to merge the ith sublist from L_1 and the ith sublist from L_2 ($1 \leqslant i \leqslant P$). All processors work simultaneously. Note that each processor puts the merged result back to the corresponding memory locations which were originally occupied by the two sublists it merges. That is, P_i puts its merged result into memory locations i, $i+P$, $i+2P,\ldots, i+(N/P-1)P$, and $N+i$, $N+i+P$, $N+i+2P,\ldots, N+i+(N/P-1)P$. Obviously, all processors concurrently merge two sorted sublists each with size N/P, hence, all processors will finish approximately at the same time in $2N/P$ time units. Note that every processor works in the memory locations assigned to the two sublists it merges and which do not overlap with the memory locations in which other processors work. Therefore, no concurrent read from or write into a memory location is required. As we will later prove in Lemma 2.1, after Step 2, every element is at most a distance P from its final position! The final position of an element is defined as the position of this element in the final sorted list.

After Step 2, L_1 and L_2 are mixed together to become a large list occupying locations from 1 to $2N$. In Step 3, we first group this entire list into $2P$ non-overlapping *groups* with P consecutive elements in each group. We number these groups from 1 to $2P$; hence, the ith group occupies the memory locations from $(i-1)P+1$ to iP. Note that each group is a sorted sublist as will be proved in Lemma 2.2. We then assign P_i to merge groups $2i-1$ and $2i$ ($i=1, 2,\ldots, P$). As in Step 2, all P processors concurrently merge two sorted sublists each with size P and there is no overlapping in memory locations between processors. Note also that each processor also puts the merged result back to the corresponding memory locations originally occupied by those two groups it merges.

In Step 4, we again group the entire list after Step 3 into $2P$ non-overlapping groups with P consecutive elements in each group as in Step 3. We then assign P_i to merge groups $2i$ and $2i+1$ for i from 1 to $P-1$. Note that only $P-1$ processors are used and the first and the last groups are not processed in this step. All $P-1$ processors concurrently merge two sublists each with size P and no overlapping in memory locations between processors. As before, the merged result is put back to the corresponding memory locations. As will be proved in Theorem 2.4, the resulting list is sorted after this step. In brief, the flow of the algorithm is given below.

Algorithm.

Step 1: Divide L_1 into P sublists where the ith sublist contains elements at locations i, $P+i$, $2P+i,\ldots, N-P+i$. Also divide L_2 into P sublists similarly.

Step 2: Have P_i merge the ith sublist from L_1 and the ith sublist from L_2 and put the result back to the locations originally occupied by these two sublists for $1 \leqslant i \leqslant P$. All P processors work simultaneously.

Step 3: Group the resulting list after Step 2 into $2P$ groups with P consecutive elements in each group. Number these groups from 1 to $2P$. Have P_i merge groups $2i-1$ and $2i$ and put the result back to the locations originally by these two groups for $1 \leqslant i \leqslant P$. All P processors work simultaneously.

Step 4: Group the resulting list after Step 3 into $2P$ groups with P elements in each group. Number these groups from 1 to $2P$. Have P_i merge groups $2i$ and $2i+1$ and put the result back to the locations originally occupied by these two groups for $1 \leqslant i \leqslant P-1$. All $P-1$ processors work simultaneously.

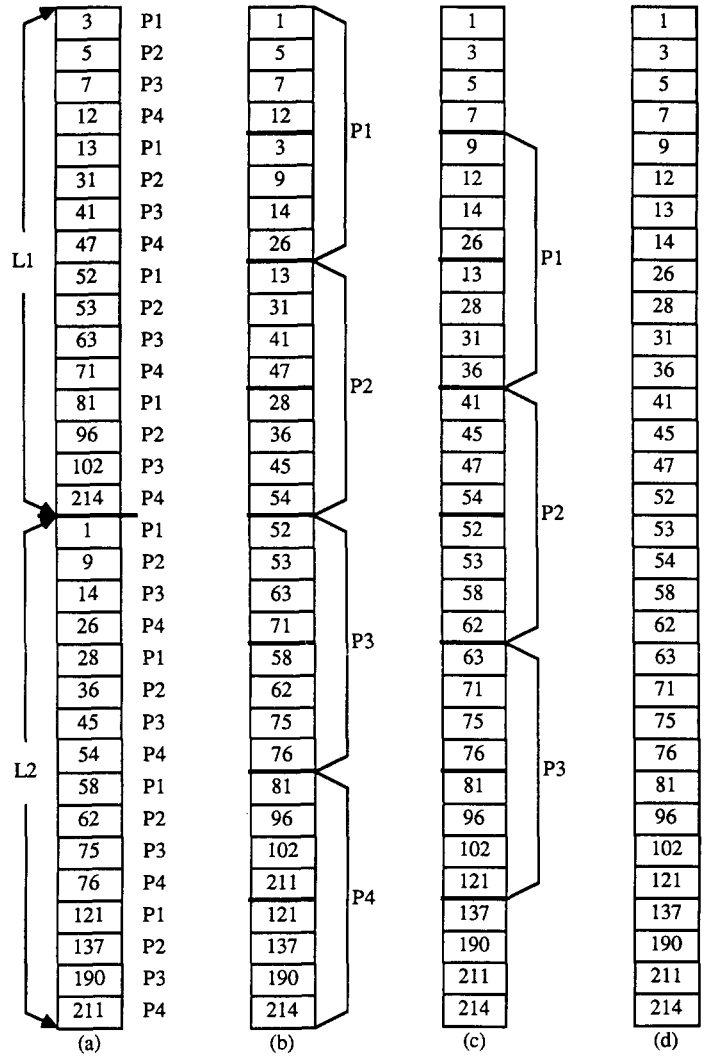

Fig. 1. An example.

Example. Figure 1 shows two sorted lists of equal size with $N = 16$ and $P = 4$ as shown in Fig. 1(a). After Step 2 in the algorithm we have the list as shown in Fig. 1(b). After Step 3 we have the list as shown in Fig. 1(c). After Step 4 we have the final sorted list as shown in Fig. 1(d) and the algorithm is completed.

Lemma 2.1. *After Step* 2, *every element is within* $\pm P$ *positions from its final position.*

Proof. In Step 1, we define L_{1i} to be the sublists assigned to P_i for merging from L_1. Similarly, L_{2i} is the sublist assigned to P_i from L_2. Without loss of generality, we examine the elements assigned to P_i. We assume that X is the nth element in L_{2i} (i.e., X is the $[i + (n-1)P]$th element in L_2) as shown in Fig. 2. Figure 2(a) shows lists L_1 and L_2 before Step 2 and Fig. 2(b) shows the result after Step 2. For X, one of the following three cases may happen. Case 1: there are elements A and B in L_{1i} where A is the mth element in L_{1i} (i.e., A is the $[i + (m-1)P]$th element in L_1) and B is the $(m+1)$st element in L_{1i} (i.e., B is the $[i + mP]$th

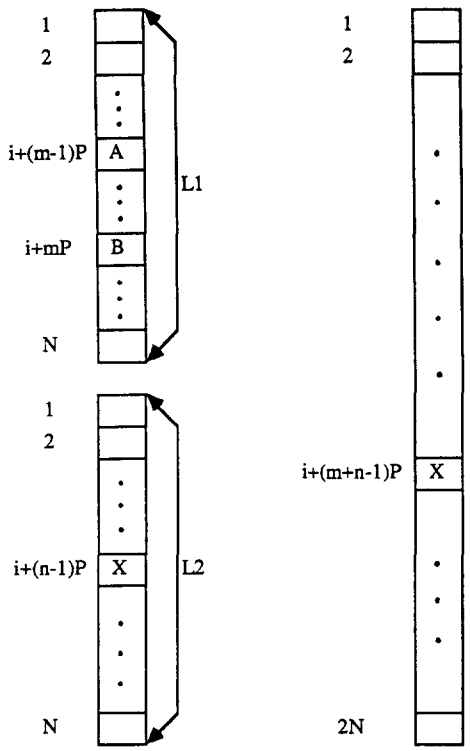

Fig. 2.

element is L_1) and $A < X < B$. Case 2: B is the first element in L_{1i} and $X < B$. Case 3: A is the last element in L_{1i} and $A < X$. We will first prove this lemma for Case 1.

(1) Number of elements smaller than X is at least

$$[(m-1)P+i]+[(n-1)P+i-1]=[(m+n-1)P+i]+(i-P-1)$$

where $(m-1)P+i$ elements are from L_1 (i.e., elements smaller than or equal to A) and $(n-1)P+i-1$ elements are from L_2 (i.e., elements smaller than X).

(2) Number of elements smaller than X is at most

$$[(m+1-1)P+i-1]+[(n-1)P+i-1]=[(m+n-1)P+i]+(i-2)$$

where $(m+1-1)P+i-1$ elements are from L_1 (i.e., elements smaller than B) and $(n-1)P+i-1$ elements are from L_2.

From (1) and (2), the ranking of X in all elements is between $[(m+n-1)P+i]+(i-P)$ and $[(m+n-1)P+i]+(i-1)$. From our algorithm, the position of X after Step 2 is $(m+n-1)P+i$. Therefore, element X is within a distance P from its final position.

Using the same argument, we can easily prove this lemma for Cases 2 and 3 and this lemma is hence proved. □

Lemma 2.2. *After Step* 2, *every group is sorted.*

Proof. In Step 1, we denote the $2P$ elements assigned to P_i for merging to be $E_{i,1}$, $E_{i,2}$, ..., $E_{i,2P}$ where $E_{i,k}$ stands for the kth element assigned to P_i. Clearly the elements from $E_{i,1}$ to

$E_{i,P}$ are from L_1 and the rest are from L_2. We further define the kth element from P_i *after Step 2* to be the kth element in ordering among all $E_{i,k}$ for $1 \leqslant k \leqslant 2P$ after P_i finishes merging the $2P$ elements.

To prove this lemma, we prove it for the ith group without loss of generality. Note that there are P elements in this group and the jth element ($1 \leqslant j \leqslant P$) in the group is put in by P_j after Step 2. Also note that the jth element in this group is the ith element from P_j *after Step* 2 for $1 \leqslant j \leqslant P$. In group i, we prove this group is sorted by showing that the jth element, say X, is smaller than the $(j+1)$st element, say Y, for all j from 1 to $P-1$.

Note that every element assigned to P_{j+1} in Step 1 is greater than the element whose memory address is one less than it and which is assigned to P_j (e.g., $E_{j+1,k} > E_{j,k}$ for $1 \leqslant k \leqslant P$). Since Y is the ith element from P_{j+1} after Step 2 and each element assigned to P_{j+1} is greater than an element assigned to P_j in Step 1, it is hence clear that Y is greater than at least i elements which are assigned to P_j. Since the ith element from P_j after Step 2 is X; hence $X < Y$ and the lemma is proved. □

Lemma 2.3. *After Step* 2, *every element in group i is smaller than all elements in group j for* $j \geqslant i+2$.

Proof. We prove this lemma by showing that the largest element, say X, in the ith group is smaller than the smallest element, say Y, in the jth group where $j \geqslant i+2$. From Lemma 2.2, X is the last element in the ith group and Y is the first element in the jth group. The approach is similar to the proof of Lemma 2.2.

First note that X is assigned to P_P and Y is assigned to P_1 in Step 1. Also note that every element but two assigned to P_1 in Step 1 is greater than the element whose memory address is one less than it and which is assigned to P_P (i.e., $E_{1,k} > E_{P,k-1}$ for $2 \leqslant k \leqslant P$). The two exceptions are the two first elements assigned to P_1 from both lists (i.e., elements $E_{1,1}$ from L_1 and $E_{1,P+1}$ from L_2).

Since Y is the jth element from P_1 after Step 2, Y is greater than at least $j-2$ elements which are assigned to P_P. Since the ith element from P_P after Step 2 is X and $j-2 \geqslant i$; hence $X < Y$. □

Theorem 2.4. *After Step* 4, *the entire list is sorted.*

Proof. From Lemma 2.3 it is shown that to sort the entire list, an element in group i after Step 2 has to compare with elements only in the $(i-1)$st group and the $(i+1)$st group since all elements in group j where $j \leqslant i-2$ are smaller than it and all elements in group k where $k \geqslant i+2$ are greater than it. This is exactly done in Steps 3 and 4 where we merge group i with groups $(i-1)$ and $(i+1)$ respectively. Hence, after Step 4, the entire list is sorted. □

Complexity analysis

In Step 2, we have all P processors merge two sublists each with size N/P concurrently; hence, the time required for this step is $2N/P$ time units. This is also true for Step 3. In Step 4, we have all $P-1$ processors also merge two sublists each with size N/P concurrently; hence, the time required for this step is also $2N/P$ time units. This shows that the total time complexity of this parallel merging algorithm is $O(N/P)$. Further, we see that the scale constant of this time complexity is as small as 3.

2.2. Cases when $N > P^2$

The multi-way parallel merging algorithm can easily be applied in cases when $N > P^2$ with minor modification. The algorithm is basically the same as the cases when $N = P^2$ except that

Steps 3 and 4 are slightly modified. To explain the algorithm, we assume $N = MP$ where $M > P$ (i.e., $N > P^2$).

Algorithm.

Step 1: Divide L_1 into P sublists where the ith sublist contains elements at locations i, $P+i$, $2P+i, \ldots, N-P+i$. Also divide L_2 into P sublists in the same way.

Step 2: For all i from 1 to P, have P_i merge the ith sublist from L_1 and the ith sublist from L_2 and put the result back to the locations originally occupied by these two sublists. All P processors work simultaneously.

Step 3: Group the resulting list Step 2 into $2M$ groups with P consecutive elements in each group. Number these group from 1 to $2M$. For all i from 1 to P, assign P_i to merge groups $2i-1$ and $2i$ and put the result back to the locations originally occupied by these two groups. After this is done, for all i from 1 to P, assign P_i to merge groups $2P+(2i-1)$ and $2P+2i$. Repeat this procedure until every two consecutive groups are merged.

Step 4: Group the resulting list after Step 3 into $2M$ groups with P elements in each group. Number these groups from 1 to $2M$. For all i from 1 to P, assign P_i to merge groups $2i$ and $2i+1$ and put the result back to the locations originally occupied by these two groups. After this is done, for all i from 1 to P, assign P_i to merge groups $2P+2i$ and $2P+(2i+1)$. Repeat this procedure until every two consecutive groups are merged except the first and the last groups.

It is easy to show that the time complexity above is still $O(N/P)$. Hence, we conclude that for cases when $N \geqslant P^2$, the multi-way parallel merging algorithm achieves the optimal time complexity $O(N/P)$.

2.3. Cases when $N < P^2$

In this section we modify the merging algorithm such that it can be applied to cases when $N < P^2$. As mentioned earlier, the time complexity in this case is degraded by a factor of 3^k using $P = N^{(2^k-1)/2^k}$ processors for $k \geqslant 1$.

Recall that in cases when $P = \sqrt{N}$, one processor is used to merge two sublists with a total size $2\sqrt{N}$. If we have more processors (i.e., $P > \sqrt{N}$), we can apply more than one processor to merge the $2\sqrt{N}$ elements. We illustrate the case when $P = N^{3/4}$ as an example. The approach is shown below.

Step 1: Divide each of L_1 and L_2 into $N^{1/2}$ sublists in such a way that each sublist contains elements which are $N^{1/2}$ positions apart. There are $N^{1/2}$ elements in each sublist.

Step 2: We want to merge two sublists using all the processors. Since there are only $N^{1/2}$ pairs of sublists waiting for merging and we have more than $N^{1/2}$ processors (i.e., $P > N^{1/2}$), more than one processor can be assigned to merge two sublists. Also, there are $N^{1/2}$ element in each sublist, we can apply the multi-way parallel merging algorithm mentioned in Section 2.1 by using $(N^{1/2})^{1/2} = N^{1/4}$ processors to merge two sublists. That is, each processor merges two sub-sublists each with $N^{1/4}$ elements. Since there are $N^{1/2}$ independent pairs of merging working concurrently, the total number of processors required is $N^{1/2}N^{1/4} = N^{3/4}$, which is exactly the total number of processors. That means all processors work simultaneously.

Step 3: Group the resulting list after Step 2 into $2\sqrt{N}$ groups with $N^{1/2}$ elements in each group. We want to merge every two neighboring groups as before. Again, we can use

$N^{1/4}$ processors to work on each merging. As in Step 2, there are $N^{1/2}$ independent merging working concurrently, the number of processors required is $N^{3/4}$. That means all processors work simultaneously.

Step 4: Similar to Step 3 except that the group merging starts from group number two.

Here we examine the time complexity of this algorithm. Since we add one more level of the multi-way parallel merging, the time required will be three times more than the original as depicted in Section 2.1. By repeatedly nesting this approach on merging two sublists, it can be shown that we can use $P = N^{(2^k-1)/2^k}$ processors for the merging algorithm to achieve a time complexity of $O(3^k N/P)$.

3. Multi-way parallel sorting algorithm

We construct a *multi-way parallel sorting algorithm* using the merging algorithm described above. In this section we will only discuss cases when $N = P^2$. The cases when $N > P^2$ and $N < P^2$ can be done similar to the procedures given in Section 2. This sorting algorithm is basically a mergesort algorithm except that we use the multi-way parallel merging algorithm to perform the merging.

We use $P = \sqrt{N}$ processors to sort $2N$ elements. For ease of explanation, we assume $P = \sqrt{N} = 2^k$. There are two phases in this algorithm. In the first phase of the algorithm, we assign $2\sqrt{N}$ elements to each processor and have each processor sort its data using any known optimal sequential sorting algorithm, e.g., quicksort. After phase 1, we have P sorted lists.

In the second phase, we recursively merge two sorted lists into one larger sorted list until there is only one list which is totally sorted. This is exactly what mergesort does. In mergesort, we define a *run* as merging every two neighboring lists into one larger sorted list for all lists. Hence, each time a run is performed, the number of sorted lists is reduced by half. After phase one, there are $P = 2^k$ sorted listed; therefore, we need to perform k merge runs to finish the mergesort. Hence, we divide phase two into k steps. At the beginning of the ith step ($i = 1,2,\ldots, k$), there are $2^k/2^{i-1}$ sorted lists each with size 2^{k+i}. The number of pairs of sorted lists waiting for merging in this step is $2^k/2^{i-1}$ divided by 2, which equals $2^k/2^i$. Since there are totally $P = 2^k$ processors, the number of processors used to merge every two lists is hence 2^i. In each step, we concurrently merge pairs of sorted lists using the processors associated with every two sorted lists. After k steps of merging, there is only one sorted list remained and the algorithm is completed.

We define $N^{(i)}$ to be the size of each list to be merged and $P^{(i)}$ to be the number of processors used to merge two lists in step i. If we can show $N^{(i)} \geq P^{(i)^2}$ for all i, all steps achieve an optimal time complexity using the multi-way parallel merging algorithm. This can easily be proved as follows: $N^{(i)} = 2^{(k+i)} \geq 2^{i+i} = (2^i)^2 = P^{(i)^2}$. From the above proof, it is shown that every merging in the mergesort obtains an optimal time complexity; therefore, the entire sorting algorithm is optimal.

Complexity analysis

Since each processor in phase one sorts two lists each with size $\sqrt{N}$ and all processors work concurrently, the time complexity of phase one equals the time complexity of any optimal sequential sorting algorithm which equals $O(2\sqrt{N} \log 2\sqrt{N})$. By neglecting the constant multiplier, the complexity above equals $O((N \log \sqrt{N})/\sqrt{N})$ which in turn equals $O((N \log N)/P)$.

In phase two, the time complexity of the merging in the ith step is $O(N^{(i)}/P^{(i)}) = O(2^{k+i}/2^i) = O(2^k) = O(N/P)$. Note that this time complexity is the same for all steps and

independent of i. Since there are k steps in phase two and $k = \log P = \log \sqrt{N} = \frac{1}{2} \log N$, the total time complexity of phase two equals $O((N/P)k) = O((N \log N/P)$. Since both phases have a time complexity as $O((N \log N)/P)$, the total time complexity of this *multi-way parallel sorting algorithm* is $O((N \log N)/P)$, which is optimal.

4. Conclusion

Multi-way parallel merging and sorting algorithms provide an optimal time complexity using $P \leqslant \sqrt{N}$ processors. Further, these algorithms do not require reading from or writing into the same memory location concurrently; hence, they can be implemented on any kind of parallel computing systems (EREW, ERCW, CREW, or CRCW). As mentioned earlier, another contribution of these algorithm is the simplicity and regularity of the structure. In addition, we show that for $P = N^{(2^k-1)/2^k}$, the complexities of the merging algorithm and the sorting algorithm are $O(3^k N/P)$ and $O(3^k(N \log N/P)$ respectively.

References

[1] M. Ajtai, J. Komlos and E. Szemeredi, An $O(N \log N)$ sorting network, in: *Proc. 15th ACM Symp. Theory Comput.* (1983) 1–9.

[2] S.G. Akl, *Parallel Sorting Algorithms* (Academic, Orlando, FL, 1985).

[3] S.G. Akl, *The Design and Analysis of Parallel Algorithms* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[4] S.G. Akl and N. Santoro, Optimal parallel merging and sorting without memory conflicts, *IEEE Trans. Comput.* **36** (10) (1987) 1367–1369.

[5] K. Batcher, Sorting networks and their application, in: *Proc. AFIPS Spring Joint Comput. Conf.* (1968) 307–314.

[6] G. Bilardi and F. Preparata, A minimum VLSI network for O(log N) time sorting, *IEEE Trans. Comput.* **34** (4) (1985) 336–343.

[7] A. Borodin and J. Hopcroft, Routing, merging and sorting on parallel models of computation, *J. Comput. System Sci.* **30** (1985) 130–145.

[8] D.M. Echstein, Simultaneous memory accesses, Tech. Rep. 79-6, Dept. Comput. Sci., Iowa State Univ., Ames, IA, 1979.

[9] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching.* (Addison-Wesley, Reading, MA, 1973).

[10] C. Kruskal, Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.* **32** (10) (1983) 942–946.

[11] S. Lakshmivarahan, S.K., Dhall and L.L. Miller, Parallel sorting algorithms, in: M.C. Yovits, ed., *Advances in Computers* (Academic Press, New York, 1984) 295–354.

[12] T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Trans. Comput.* **34** (4) (1985) 344–354.

[13] F.P. Preparata, New parallel-sorting schemes, *IEEE Trans. Comput.* **27** (7) (1978) 669–673.

[14] Y. Shiloach and U. Vishkin, Finding the maximum, merging and sorting in a parallel computation model, **2** (1981) 88–102.

[15] L.G. Valiant, Parallelism in comparison problems, *SIAM J. Comput.* **4** (1975) 348–355.