

On the Average Running Time of Odd–Even Merge Sort

Christine Rüb^{*,†}

Max-Planck-Institut für Informatik Im Stadtwald, D-66123 Saarbrücken, Germany

Received April 5, 1995

This paper gives an upper bound for the average running time of Batchers odd–even merge sort when implemented on a collection of processors. We consider the case where n , the size of the input, is an arbitrary multiple of the number p of processors used. We show that Batchers odd–even merge (for two sorted lists of length n each) can be implemented to run in time $O((n/p)(\log(2 + p^2/n)))$ on the average,¹ and that odd–even merge sort can be implemented to run in time $O((n/p)(\log n + \log p \log(2 + p^2/n)))$ on the average. In the case of merging (sorting), the average is taken over all possible outcomes of the merge (all possible permutations of n elements). That means that odd–even merge and odd–even merge sort have an optimal average running time if $n \geq p^2$. The constants involved are also quite small. © 1997 Academic Press

1. INTRODUCTION

This paper is concerned with a well-known parallel algorithm for sorting, namely Batchers odd–even merge sort. Originally this algorithm was described as a comparator network of size $O(n \log^2 n)$ that sorts n elements in time $O(\log^2 n)$ [1, 11]. However, it can also be used as a sorting procedure on a parallel computer. In this case, there will, in general, be fewer processors than input elements. Odd–even merge can then be implemented by substituting all comparisons (and exchanges) between two elements by splits of two sorted lists at the median of their union [11]. If this is done in a straightforward way (that is, two processors that communicate exchange all their elements), the running time will be $\Theta((n/p)(\log(n/p) + \log^2 p))$, where p is the number of processors used.

^{*}Supported in part by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität, and in part by the ESPRIT Basic Research Actions Program of the EU under contract No. 7141 (project ALCOM II).

[†]E-mail: rueb@mpi-sb.mpg.de

¹In this paper, $\log x$ always means $\log_2 x$.

In practice, when comparing different sorting algorithms [4, 9, 6], odd-even merge sort has not often been used. Instead, another sorting algorithm proposed by Batcher [1], namely bitonic merge sort, has been used in the comparisons. Because of its small constant factors, bitonic merge sort is quite efficient on many parallel machines. Perhaps the reason for this preference for bitonic merge sort is that the communication pattern of odd-even merge sort does not directly correspond to any of the generally used interconnection networks.

In this paper we will show, however, that odd-even merge sort can be implemented (by keeping the communication between processors to a minimum) in such a way that it performs on the average much better than bitonic merge sort. More precisely, we will show that the average running time of odd-even merge sort can be $O((n/p)(\log n + \log p \log(2 + p^2/n)))$ (with small constant factors), whereas the average running time of bitonic merge sort (in the generally used form) is $\Theta((n/p)(\log n + \log^2 p))$. (Here n is the size of the input and p is the number of processors used. The average is taken over all possible ways to store the input elements evenly distributed among the processors.) In particular, odd-even merge sort needs on the average much less communication than bitonic merge sort; this is important since communication is still relatively expensive with existing parallel machines. Here we assume that all required connections between processors are present, e.g., that the processors form a completely connected graph. In Section 5 we will comment on this and present some implementation results.

To my surprise, I could not find a reference to this result in the literature. (Perhaps this is the case because odd-even merge sort has not often been used in practice.) Several papers do exist about the average number of exchanged elements for the comparator network, i.e., for the case that $n = p$ [15, 8, 12]. In these papers it is shown that odd-even merge needs $\Theta(m \log m)$ exchanges of elements where both lists contain m elements. Another related result is presented in [2]. The authors consider an algorithm (called Gray Sort) for sorting n elements using p processors that consists of $\log p$ phases. Each phase consists of the first and the last step of (several instances of) odd-even merge on subsets of the processors. (The authors do not seem to be aware of this; odd-even merge is not mentioned in the paper.) They show that this algorithm sorts with high probability if $n \geq cp^2 \log p$ for a constant $c > 18 \ln(2)$.

This paper is organized as follows. In Section 2 we analyze the average running time of odd-even merge. These results are used to derive an upper bound for odd-even merge sort in Section 3. Section 4 revisits bitonic merge sort, and Section 5 contains implementation results and draws some conclusions.

2. THE AVERAGE RUNNING TIME OF ODD-EVEN MERGE

In this section we will derive an upper bound on the average running time of odd-even merge on a collection of p processors, each with its own local memory. The processors communicate by exchanging messages. For the sake of simplicity, we assume in this and the following section that the processors form a complete graph, i.e., each pair of processors is directly connected (we will comment on this in Section 5). However, in one step, a processor may only communicate with one of its neighbors. At the beginning the elements are distributed evenly among the processors. We assume that every outcome of the merging is equally likely, that p is a power of 2, and that all input elements are distinct.

We will use the following argument. Consider the odd-even merging network for two sorted lists of $p/2$ elements each. It consists of $\log p$ steps where elements are compared and rearranged. We can use this procedure to merge two sorted lists, each of length m (m a multiple of $p/2$), using p processors by substituting all comparisons (and exchanges) between two elements by splits of two sorted lists at the median of their union. If this is done in a straightforward way (that is, two processors that communicate exchange all their elements), the running time will be $\Theta((m/p)\log p)$. However, the average running time of an implementation can be reduced considerably if the splitting of lists is done in a more efficient way, i.e., if the communication between processors is kept to a minimum. For example, if two processors have to communicate in a step, they will first test whether they hold any elements that have to be exchanged. If this is the case for any pair of processors that communicate in a given step, we say that this step is *expensive*. We will see that if m is large enough, with high probability only few of the $\log p$ steps of odd-even merge are expensive. More precisely, we show that the average number of expensive steps in odd-even merge is $O(\log(2 + p^2/m))$. Note that a step that is not expensive results in costs of $O(1)$, whereas an expensive step may produce costs of up to $O(m/p)$ (if two processors exchange all their elements). Since we want to derive an upper bound on the average running time, we change costs $O(m/p)$ for each expensive step. It follows from this that the average running time of this implementation of odd-even merge is $O((m/p)(\log(2 + p^2/m)) + \log p) = O((m/p)(\log(2 + p^2/m)))$.

The odd-even merge network works as follows:

Let $A = A_0, A_1, A_2, \dots, A_{m-1}$ and $B = B_0, B_1, B_2, \dots, B_{m-1}$ be the two sequences to be merged and denote the merge outcome by E_0, \dots, E_{2m-1} .

If $m = 1$, compare and exchange, if necessary, A_0 and B_0 . Else recursively merge sublists of A and B containing only elements with even or odd indexes. More precisely, merge $A_{\text{even}} = A_0, A_2, A_4, \dots$ with $B_{\text{even}} = B_0, B_2, B_4, \dots$ into $C = C_0, C_1, C_2, \dots$ and $A_{\text{odd}} = A_1, A_3, \dots$ with $B_{\text{odd}} = B_1, B_3, \dots$ into $D = D_0, D_1, D_2, \dots$. After this is done, compare and exchange, if necessary, D_i with C_{i+1} to form elements E_{2i+1} and E_{2i+2} of the output, $i \geq 0$.

We want to use this procedure for merging using a collection of processors and thus have to determine where the elements are stored. (The average number of exchanged elements depends strongly on how this is done.) As it turns out, the best way to do this is the way already proposed by Batcher. To begin with, we assume that $2m$ processors are available. Thus we store the list A_0, A_1, A_2, \dots at processors P_0, P_2, P_4, \dots , the list B_0, B_1, B_2, \dots at processors P_1, P_3, P_5, \dots , and the list $E_0, E_1, E_2, E_3, \dots$ at processors $P_0, P_1, P_2, P_3, \dots$ (see Fig. 1). By unfolding the recursion we get the following procedure (see, e.g., [15]). Let $p = 2m$.

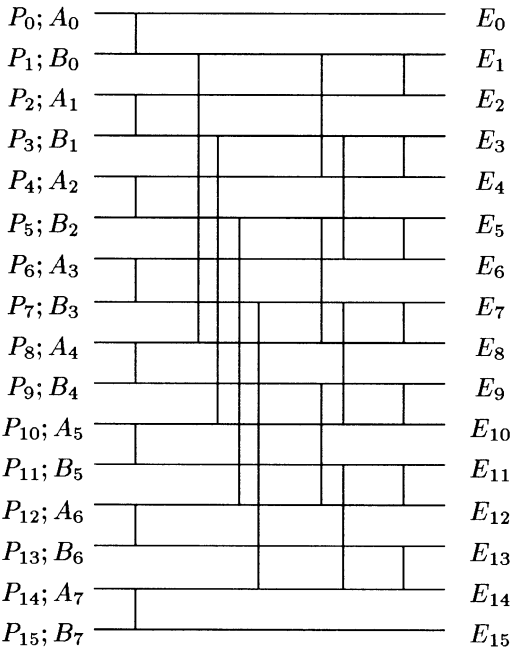


FIG. 1. Odd-even merge with 16 processors.

```

procedure Odd_Even_Merge( $p$ );
for all  $i$ ,  $0 \leq i < p/2$ , pardo
    compare-exchange( $P_{2i}, P_{2i+1}$ );
for  $i = \log p - 1$  downto 1 do
    for all  $j$ ,  $1 \leq j \leq (p - 2^i)/2$ , pardo
        compare-exchange( $P_{2j-1}, P_{2j+2^i-2}$ );

```

Compare-exchange (P_i, P_j) denotes a procedure where P_i gets the smaller of the two elements stored at P_i and P_j and P_j gets the larger.

This procedure can be generalized to the case where $p < 2m$, i.e., where each processor holds more than one element. We assume that p divides $2m$ and that the sequences A and B are stored evenly distributed at the processors so that the even-indexed processors hold subsequences of A and the odd-indexed processors hold subsequences of B . For the remainder of this section, let $r = 2m/p$. Then we can use the same procedure as above, where compare-exchange (P_i, P_j) now denotes a procedure where P_i gets the smallest r elements of all elements stored at P_i and P_j together and P_j gets the largest r elements of these.

The following illustration shows that the above procedure merges correctly. In [11], Subsection 5.3.4, Exercise 38, it is shown that every comparator network for sorting n elements can be transformed into a sorting algorithm for xn elements by replacing each comparator by the compare-exchange procedure mentioned above. Here the input consists of n sorted lists of x elements each. It can be concluded from this that the same is true for any merging comparator network. Consider the sorting comparator network that divides the input into two equally sized sets, sorts them recursively and then merges the two sorted lists. Since the transformed sorting network sorts correctly and each input for the merging network is possible, the transformed merging network merges each input correctly.

The procedure compare-exchange(P_i, P_j) can be realized as follows. Remember that processor P_i will get the smaller elements and processor P_j will get the larger elements. First we determine whether any elements have to be exchanged by comparing the largest element of P_i with the smallest element of P_j . If elements have to be exchanged, P_i and P_j exchange all their elements. (For our upper bound this suffices; to achieve a better running time, the number of elements that have to be exchanged can first be determined with the help of binary search; see Section 5.) Note that if any elements have to be exchanged between P_i and P_j , both processors have to merge two sorted lists to get the sorted list of the elements now assigned to them.

DEFINITION 1. For $X \in \{A, B, E\}$, let $X^y = X_{yr}, X_{yr+1}, X_{yr+2}, \dots, X_{yr+r-1}$.

To derive an upper bound for the average number of expensive steps in odd-even merge, we will use the following argument. Assume that m is large compared with p . Consider the above iterative description of odd-even merge. In the first step, corresponding subsequences of A and B are compared and rearranged. If we choose the input at random, most of the elements in any subsequence of A (namely those lying in the middle of the subsequence) will be compared with their neighbors in B and vice versa. This is because, on the average, the rank of an element in A will not differ much from its rank in B if r is large enough. Moreover, these elements will, after the first step, be stored at the “correct” processors (see Fig. 1). After the first step processors P_{2i} and P_{2i+1} hold the elements in A (respectively, B) with ranks in A (respectively, B) between ir and $(i+1)r-1$. At the end, processors P_{2i} and P_{2i+1} hold E^{2i} and E^{2i+1} , i.e., all elements with overall rank between $2ir$ and $2(i+1)r-1$. Thus, for most inputs, most of the elements will be stored at the correct processors after the first step. (This observation shows why it is best to store the input and the output as suggested by Batcher and as is done here.)

Next consider the for-loop of the algorithm. If there are many steps where elements are exchanged there has to be a large i where this happens. Denote by i_{\max} the first (or largest) i where at least one element has to be exchanged between two processors. We will show that there exists an element x such that the difference between the rank of x in A and the rank of x in B is large, namely at least $(2^{i_{\max}-1}-1)r+1$. For example, if all $\log p-1$ steps of the for-loop are expensive, at least one element exists with a difference of almost $m/2$ between its rank in A and its rank in B . However, there are not many inputs for merging with this property if m/p and i_{\max} are large.

Next we will analyze this property more closely and then show what it means for the average running time of odd-even merge.

LEMMA 1. *Assume that at least two elements are exchanged in the for-loop of the algorithm for a fixed i . There then exists an element x in $A \cup B$ where the rank of x in A differs from the rank of x in B by at least $(2^{i-1}-1)r+1$.*

Proof. Let $i_{\max} = \max \{i; \text{at least two elements are exchanged in the for-loop}\}$, and let $\delta = 2^{i_{\max}}$. Thus, before the execution of the for-loop with $i = i_{\max}$, processors P_{2j} and processors P_{2j+1} hold all elements in $A^j \cup B^j$, $0 \leq j < p/2$. Let P_{2k+1} and $P_{2k+\delta}$ be two processors that exchange elements in this step. Since P_{2k+1} holds only elements of $A^k \cup B^k$ and $P_{2k+\delta}$ holds only elements of $A^{k+\delta/2} \cup B^{k+\delta/2}$, one of the processors sends only elements from A , and the other one only elements from B . Note that this is not necessarily the case for later steps. Then it might happen that P_{2k+1} sends and receives only elements from A (or B).

Assume that P_{2k+1} sends an element from A . The largest element from A that P_{2k+1} holds is $x = A_{(k+1)r-1}$, and the smallest element from B that $P_{2k+\delta}$ holds is $B_{(k+\delta/2)r}$. Thus the rank of x in A differs from the rank of x in B by at least $(k + \delta/2)r - ((k+1)r - 1) = (\delta/2 - 1)r + 1$. Since $i_{\max} \geq i$, the claim follows. ■

Next we upper-bound the probability that an element $x \in A \cup B$ exists such that the rank of x in A differs from the rank of x in B by Δ .

LEMMA 2. *Let A and B be two sorted sequences of length m each, and let each outcome of merging A and B be equally likely. Then the probability that an element x in $A \cup B$ exists where the rank of x in A differs from the rank of x in B by Δ is at most*

$$2 \binom{2m}{m+\Delta} / \binom{2m}{m}.$$

Proof. This can be shown with the help of the reflection principle (see, e.g., [7], pp. 72–73) as follows. We can represent each input as a path from point $(0, 0)$ to point $(2m, 0)$ (see Fig. 2).

We are interested in all paths that touch or cross the line $y = \Delta$ or the line $y = -\Delta$; these correspond to inputs where a difference in rank of at least Δ exists. According to the reflection principle, the number of paths that touch or cross the line $y = a$, $a > 0$, is equal to the number of paths from $(0, 0)$ to $(2m, 2a)$. Thus the number of inputs with a difference in rank of at least Δ is at most $2 \binom{2m}{m+\Delta}$, and the probability that an input has this property is at most $2 \binom{2m}{m+\Delta} / \binom{2m}{m}$. ■

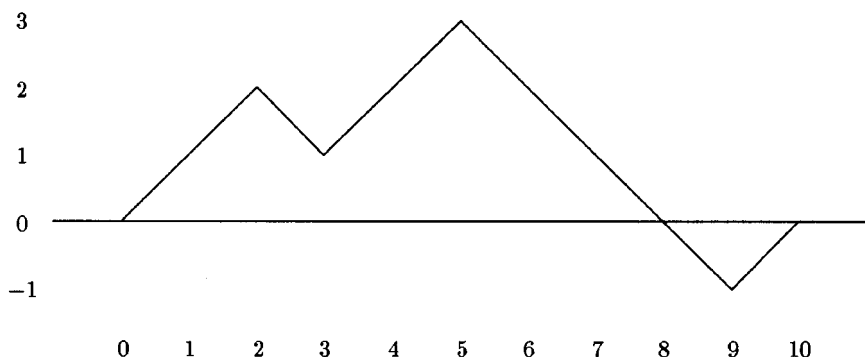


FIG. 2. $m = 5$. The path shown corresponds to the input where the outcome of the merging is *AABAABBBBA*.

The following lemma gives an upper bound for $\binom{2m}{m+\Delta} / \binom{2m}{m}$.

LEMMA 3. For all $m \geq 0$ and $\Delta < m$,

$$\binom{2m}{m+\Delta} / \binom{2m}{m} \leq e^{-(\Delta^2/m)} \sqrt{\frac{(2m)^2}{\pi(m+\Delta)(m-\Delta)}}.$$

Proof. This can be shown with the help of Stirling's approximation of $n!$ and a Taylor series expansion of $\ln x$; the Appendix gives more details. ■

Now we are ready to prove an upper bound for the average number of expensive steps in odd-even merge.

LEMMA 4. The average number of expensive steps in odd-even merge is bounded by $1.39 + \lceil \log(1 + \sqrt{p^2/m}) \rceil$. The probability that at least $1 + \lceil \log(1 + \sqrt{p^2/m}) \rceil + \delta$, $\delta \geq 0$, steps are expensive is bounded by $e^{-2^{2\delta}}$.

Proof. Let $\text{prob}(i)$ be the probability that a step of the for-loop is expensive for a fixed i . Remember that a step of odd-even merge is called expensive if at least two processors have to exchange elements. The average number of expensive steps is bounded by

$$1 + \sum_{i=1}^{\log p - 1} i(\text{prob}(i) - \text{prob}(i+1)) = 1 + \sum_{i=1}^{\log p - 1} \text{prob}(i) =: A(m, p),$$

where $\text{prob}(\log p) = 0$.

We know from Lemma 1 that there is a difference in rank of at least $(2^{i-1} - 1)r + 1$ if a step of the for-loop is expensive for a fixed i . Together with Lemmas 2 and 3 this gives an upper bound of

$$1 + \sum_{i=0}^{\log p - 2} \min \left(1, 2e^{-4(2^i - 1)^2 m / p^2} \sqrt{\frac{4p^2}{\pi(p^2 - 4(2^i - 1)^2)}} \right)$$

for $A(m, p)$. Let $c = 8/\sqrt{3\pi}$. Since $i \leq \log p - 2$, this is less than or equal to

$$1 + \sum_{i=0}^{\log p - 2} \min(1, e^{-4(2^i - 1)^2 m / p^2 + \ln(c)}).$$

The last term of this behaves in the following way. As long as the absolute value of the exponent is smaller than 1, the value of the term lies

between 1 and $1/e$; when the absolute value becomes larger, the value of the term drops rapidly towards 0. Since there are $\lfloor \log(1 + \sqrt{(1 + \ln(c))p^2/4m}) \rfloor$ terms of the sum where the absolute value of the exponent is smaller than 1, we can expect that the value of the sum is $O(1 + \log(1 + \sqrt{p^2/m}))$. To actually derive an upper bound for $A(m, p)$ we cut the sum into two parts, one containing the “large” terms and one containing the “small” terms. Let $x = \log(1 + \sqrt{(1 + \ln(c))p^2/4m})$ and let $i_x = \lceil x \rceil$, i.e., i_x is the smallest value of i where the absolute value of the exponent is larger than 1. With this we get

$$\begin{aligned}
 A(m, p) &\leq 1 + \sum_{i=0}^{i_x-1} 1 + \sum_{i=i_x}^{\log p-2} e^{-4(2^i-1)^2 m / p^2 + \ln(c)} \\
 &\leq 1 + i_x + \sum_{i=0}^{\log p-2-i_x} e^{-4(2^{x2^i-1})^2 m / p^2 + \ln(c)} \\
 &= 1 + i_x + \sum_{i=0}^{\log p-2-i_x} e^{-4(2^i(1 + \sqrt{(1 + \ln(c))p^2/4m}) - 1)^2 m / p^2 + \ln(c)} \\
 &\leq 1 + i_x + \sum_{i=0}^{\log p-2-i_x} e^{-2^{2i}} \\
 &\leq 1 + i_x + 0.39 \leq 1.39 + \left\lceil \log(1 + \sqrt{p^2/m}) \right\rceil.
 \end{aligned}$$

The second claim follows directly from the above discussion. ■

Table 1 gives some examples. For various numbers of processors, it gives the required size of the input such that with probability at least 0.99 (respectively, 0.999) not more than the listed number of steps are expensive. The size of the input is given as the number of elements per processor. The numbers are computed directly from Lemmas 2 and 3.

TABLE 1

No. proc. \ No. steps	0.99					0.999				
	2	3	4	5	6	2	3	4	5	6
16	23	6	1	—	—	64	8	1	—	—
64	180	20	4	1	—	252	28	6	2	1
256	714	80	16	4	1	1008	112	22	6	2
1024	2850	318	60	14	4	4028	448	84	18	6
4096	11394	1266	234	52	12	16110	1790	330	72	18

Since each step of odd-even merge can be executed in time $O(m/p)$, we get the following upper bound for the average running time of odd-even merge.

LEMMA 5. *The average running time of odd-even merge is $O((n/p)\log(2 + p^2/n))$, where p is the number of processors and n is the size of the input.*

3. THE AVERAGE RUNNING TIME OF ODD-EVEN MERGE SORT

Building on the results from Section 2 we can now derive an upper bound for the running time of odd-even merge sort. Odd-even merge sort works in the following way. Let n be the size of the input, let p be the number of processors, let p divide n , and let the processors be numbered from 0 to $p - 1$. At the beginning each processor stores n/p of the elements. In the first step each processor locally sorts its portion of the input. In the following $\log p$ steps, these sorted lists are merged in the following way. In step i , $0 \leq i \leq \log p - 1$, the lists stored at processors $P_j, P_{j+\delta}, P_{j+2\delta}, \dots$ and processors $P_{j+\delta/2}, P_{j+\delta/2+\delta}, P_{j+\delta/2+2\delta}, \dots$ are merged, where $\delta = p/2^i$ and $0 \leq j < p/2^{i+1}$ (see Fig. 3). Note that this means that in each instance of odd-even merge the elements are stored as required in the previous section.

Again we will derive an upper bound for the number of steps in which elements have to be exchanged.

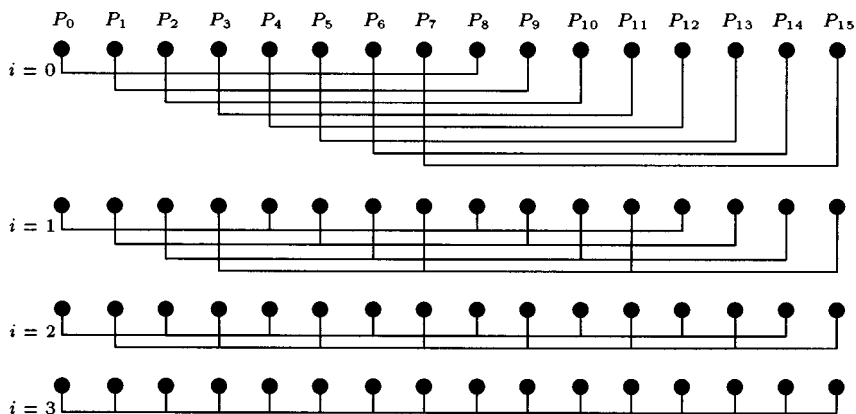


FIG. 3. The processors that work together in step i are shown connected.

LEMMA 6. Let $c = 8/\sqrt{3\pi}$. In the j -th call of odd-even merge, $0 \leq j \leq \log p - 1$, the average number of steps in which elements are exchanged is bounded by $1.39 + \lceil \log(1 + \sqrt{(1 + \ln(cp/2^{j+1}))}p^{2^j}/n) \rceil$.

Proof. In the j -th call of odd-even merge, $p/2^{j+1}$ independent mergings are performed, each with 2^{j+1} processors and $(n/p)2^{j+1}$ elements. Thus, the for-loop in odd-even merge consists of j steps. The probability that at least two elements are exchanged in any of the $p/2^{j+1}$ mergings during the execution of the for-loop for a specific i , $j \geq i \geq 1$, is bounded by

$$\min\left(1, \frac{cp}{2^{j+1}} e^{-2(2^{i-1}-1)^2 n / (p2^{j+1})}\right),$$

and the average number of steps with data exchanges is bounded by

$$\begin{aligned} 1 + \sum_{i=0}^{j-1} \min\left(1, \frac{cp}{2^{j+1}} e^{-2(2^i-1)^2 n / (p2^{j+1})}\right) \\ = 1 + \sum_{i=0}^{j+1} \min(1, e^{-2(2^i-1)^2 n / (p2^{j+1}) + \ln(cp/2^{j+1})}). \end{aligned}$$

Let $x = \log(1 + \sqrt{(1 + \ln(cp/2^{j+1}))p2^j/n})$ and let $x = \lceil x \rceil$. As in the proof of Lemma 4, we split the sum into “small” and “large” terms and get as upper bound

$$\begin{aligned} 1 + z + \sum_{i=z}^{j-1} e^{-2(2^i-1)^2 n / (p2^{j+1}) + \ln(cp/2^{j+1})} \\ \leq 1 + z + \sum_{i=0}^{j-1-z} e^{-2(2^{x+i}-1)^2 n / (p2^{j+1}) + \ln(cp/2^{j+1})} \\ = 1 + z + \sum_{i=0}^{j-1-z} e^{-2(2^i(1 + \sqrt{(1 + \ln(cp/2^{j+1}))p2^j/n}) - 1)^2 n / (p2^{j+1}) + \ln(cp/2^{j+1})} \\ \leq 1 + z + \sum_{i=0}^{j-1-z} e^{-2^{2i}} \\ \leq 1.39 + z. \quad \blacksquare \end{aligned}$$

Now we can derive an upper bound for the average number of steps with data exchanges in odd-even merge sort.

LEMMA 7. *The average number of steps in odd–even merge sort in which data are exchanged is bounded by $\log p(1.39 + \lceil \log(1 + \sqrt{p^2/n}) \rceil)$, where n is the size of the input and p is the number of processors.*

Proof. Let $c = 8/\sqrt{3\pi}$. According to Lemma 6, the average number of steps with data exchanges is

$$\begin{aligned} & \sum_{i=0}^{\log p - 1} \left(1.39 + \left\lceil \log \left(1 + \sqrt{(1 + \ln(cp/2^{i+1}))p2^i/(n)} \right) \right\rceil \right) \\ & \leq \log p \left(1.39 + \left\lceil \log \left(1 + \sqrt{(1 + \ln(c))p^2/(2n)} \right) \right\rceil \right) \\ & \leq \log p \left(1.39 + \left\lceil \log \left(1 + \sqrt{p^2/n} \right) \right\rceil \right). \quad \blacksquare \end{aligned}$$

Since each step in odd–even merge (except for the initial local sorting) can be executed in linear time, we arrive at the following theorem.

THEOREM 1. *The average running time of odd–even merge sort is bounded by $O((n/p)(\log n + \log p \log(2 + p^2/n)))$, where n is the size of the input and p is the number of processors used.*

4. REVISITING BITONIC MERGE SORT

In the introduction we claimed that odd–even merge sort is much faster than bitonic merge sort for random inputs. However, this is only true if the input elements are stored as suggested by Batcher. In this section, we show how to modify bitonic merge to achieve a running time that is comparable to that of odd–even merge.

The bitonic merge network works as follows.

Let $A = A_0, A_1, A_2, \dots, A_{m-1}$ and $B = B_0, B_1, B_2, \dots, B_{m-1}$ be the two sequences to be merged and denote the outcome of the merge by E_0, \dots, E_{2m-1} . If $m = 1$, compare and exchange, if necessary, A_0 and B_0 . Else, merge $A_{\text{even}} = A_0, A_2, A_4, \dots$ with $B_{\text{odd}} = B_1, B_3, \dots$ into $C = C_0, C_1, C_2, \dots$ and $A_{\text{odd}} = A_1, A_3, \dots$ with $B_{\text{even}} = B_0, B_2, B_4, \dots$ into $D = D_0, D_1, D_2, \dots$. After this is done, compare and exchange, if necessary, C_i with D_i to form elements E_{2i} and E_{2i+1} of the output, $i \geq 0$.

As in the case of odd–even merge we can use this as a merging procedure for a collection of processors. Again, every comparison and exchange between two elements is replaced by the split of two sorted lists at their median.

In this high-level description, bitonic merge looks very similar to odd-even merge. However, it leads to a very different-looking network. In the original version, the list A is stored before list B , and the list B is stored in reverse order. Figure 4a shows the bitonic merging procedure with 16 processors. If the input is stored like this, the running time of bitonic merge will be $\Omega((n/p)\log p)$ on the average, where n is the size of the input. This can be seen as follows. On the average, half of the elements in B_0 will have to be moved to P_0 . However, since B_0 is stored at P_{p-1} at the beginning, this will take time $\Omega((n/p)\log p)$. Thus, the problem is that one of the lists is stored in reverse order.

But what happens if we store the input elements in the same order as used for odd-even merge? Figure 4b shows the communication pattern for this alternative when using 16 processors. The links between the processors are drawn so that their thickness increases with the probability that they are used (i.e., that elements have to be exchanged across them). Although links drawn in the same thickness occur in several steps of the algorithm, they do not interfere much with each other; except for the last three steps, every processor is an endpoint of at most two links drawn in the same thickness. We can again argue that the running time of this merging procedure is $O((n/p)\log(2 + \Delta))$, where Δ is the largest difference of rank in A and B . With this we get asymptotically the same average running time as for odd-even merge.

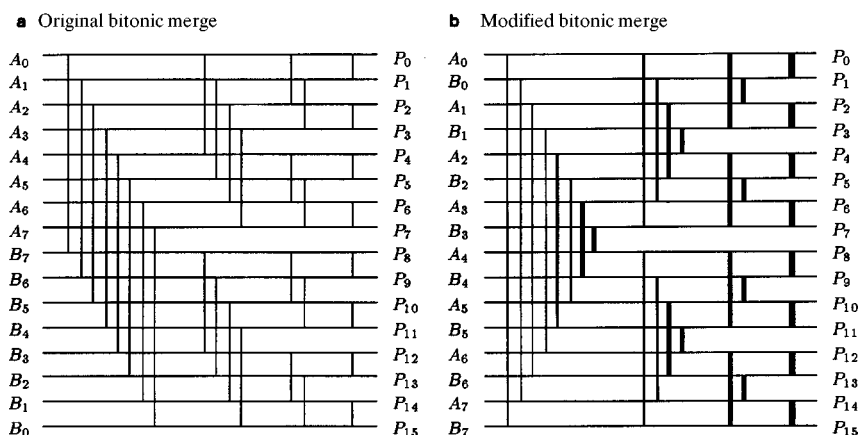


FIG. 4. Bitonic merge with 16 processors.

5. IMPLEMENTATION AND COMPARISON WITH OTHER PARALLEL SORTING ALGORITHMS

In the previous sections, we assumed for the sake of simplicity that all processors that have to communicate in odd-even merge sort are connected. On existing parallel computers this is, in general, not the case. However, many existing parallel machines can be viewed as complete graphs as long as the number of processors is not too large [5]. Other machine models in which the running time from Section 3 can be achieved are the hypercube (by using gray code [14]) and the hypercube with E-router [10]. A hypercube with E-router can perform shifts without conflicts, i.e., the running time for a shift operation will be $O(l)$, where l is the maximum length of a message sent, if $l \geq \log p$. (Intel's iPSC/860 and NCube's NCube are hypercubes with E-router.)

We have implemented the above algorithm to run on the iPSC/860 from Intel in the following way. If two processors have to communicate in a step, they first check whether they have to exchange any elements. If this is the case, they use binary search to determine which elements have to be exchanged (the latter method was proposed in [16].) This is done in order to be able to send the data in larger blocks. (On existing parallel computers this is, in general, much cheaper than sending the same data in small blocks). The elements at each processor are stored as a linear list; when new elements arrive, they are merged with the elements already present. If one of the two lists that have to be merged in a step is short, binary search is used to determine where the elements in the shorter list belong, as has been proposed in [16]. Communication is done using asynchronous send and receive; on the iPSC/860 this allows communication and local computation to be performed in parallel and for part of the communication time to be "hidden."

The speedups achieved for large inputs are quite high. For example, the speedup on the iPSC/860 for 100,000 longs per processor and using 16 processors is 12.3 or 77% of the maximum possible; for 1,600,000 longs per processor it is 14.5 or 90.6% of the maximum possible. For small inputs the speedups are not as high as one could, perhaps, expect from the asymptotic upper bounds. For example, for 300 longs per processor and 16 processors of the iPSC/860 the speedup is 1.5, i.e., the parallel algorithm is only slightly faster than the sequential algorithm. (For 5,000 longs per processor it is already 55%.) The reason for this is that the start-up time for messages is quite long; only if long messages are sent (much more than 300 longs), can the full bandwidth of the communication links be utilized. (All of the speedups listed above are for randomly generated input.)

How does this algorithm compare with other parallel sorting algorithms? As mentioned in the introduction, odd-even merge sort is, on the average,

faster than the original bitonic merge sort. Figure 5 shows a comparison between the running times of odd-even merge sort and the original bitonic merge sort for 10,000 randomly generated longs per processor on an iPSC/860. For both implementations the exchange of elements between processors is done as above. Although the maximum number of available processors was only 32, the difference in the running times is already quite large. As seen in the previous section, if we change the order in which the input elements are stored, bitonic merge sort becomes much faster for random inputs. On the iPSC/860, the running times for these two modified merge sort algorithms were very similar.

Note that this way of storing the elements in a parallel merge sort also makes the sorting algorithm adaptive; if the input is nearly sorted in the sense that no element is stored far away from its position in the sorted list, this will be reflected by the running time.

We have also compared odd-even merge sort with the sorting algorithm proposed in [16]. In the first phase of this algorithm, the input is "pre-sorted" by a hypercubic descend algorithm; in the second phase, the sorting is then finished by odd-even merge sort. The authors claim that this algorithm performs much better than odd-even merge sort alone; however, we cannot confirm this. According to our experiments, the running time is almost the same. (For large inputs, the algorithm with presorting has fewer expensive steps but sends more elements.)

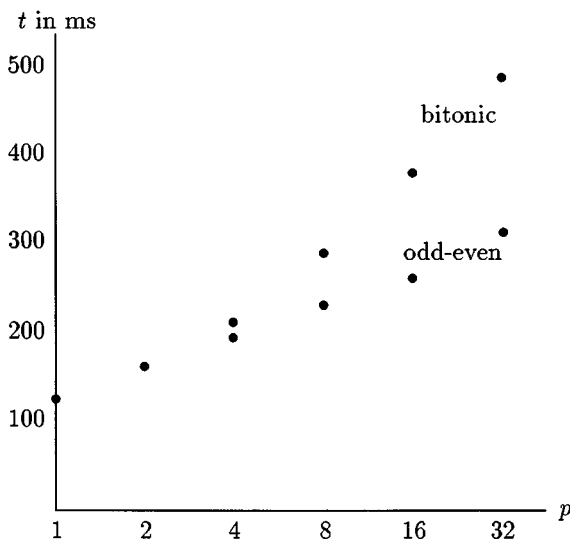


FIG. 5. Running times for sorting 10,000 longs per processors.

For large inputs, sample sort (see, e.g., [3]) will have a shorter running time. The idea of sample sort is as follows. First splitters are determined by sampling the input elements. These splitters are sorted and used to form buckets. The buckets are then distributed among the processors and each input element is sent to the processor responsible for the bucket that contains the element. One advantage of this method is that each input element is moved only once. On the other hand, this method does not work well if the input is small or if there are many identical elements.

6. CONCLUSIONS

In this paper we showed that the average running time of Batchier's odd-even merge sort can be improved considerably by means of a simple modification. A natural question to ask is whether the given bounds are tight. However, the technique used here is not powerful enough to derive a good lower bound. First, Lemma 1 from Section 2 only gives an upper bound on the number of expensive steps. Second, there is no direct relationship between the number of expensive steps and the running time of the algorithm; it is not clear how many elements are exchanged and, moreover, it is possible that several expensive steps could be executed simultaneously if the affected processors are distinct.

The results presented in this paper do not lead to exact running times on existing parallel machines; more parameters of a computer (such as the startup time for sending messages) have to be taken into consideration to get there. However, the results show that it is worthwhile to consider (this variant of) odd-even merge sort when comparing sorting algorithms on parallel computers.

APPENDIX

LEMMA 3.

$$\binom{2m}{m+\Delta} / \binom{2m}{m} \leq e^{-(\Delta^2/m)} \sqrt{\frac{(2m)^2}{\pi(m+\Delta)(m-\Delta)}}.$$

Proof. With the help of Stirling's approximation for $n!$ the following can be shown. Let $n \in \mathbf{N}$ and let $\mu n \in \mathbf{N}$, $0 < \mu < 1$. Then

$$\frac{1}{\sqrt{8n\mu(1-\mu)}} 2^{nH_2(\mu)} \leq \binom{n}{\mu n} \leq \frac{1}{\sqrt{2\pi n\mu(1-\mu)}} 2^{nH_2(\mu)},$$

where $H_2(x) = -x \log x - (1-x) \log(1-x)$. (See, e.g., [13], pp. 308 ff.)

Using this we get

$$\binom{2m}{m+\Delta} / \binom{2m}{m} \leq \sqrt{\frac{(2m)^2}{\pi(m+\Delta)(m-\Delta)}} 2^{2m(H_2((m+\Delta)/2m)-1)}.$$

Taylor series expansion of $\ln x$ around 0.5 leads to

$$\begin{aligned} \ln(x) &= \ln(0.5) + 2(x - 0.5) - \frac{2^2}{2}(x - 0.5)^2 + \frac{2^3}{3}(x - 0.5)^3 \\ &\quad - \frac{2^4}{4}(x - 0.5)^4 + \dots, \end{aligned}$$

and thus

$$\begin{aligned} H_2\left(\frac{m+\Delta}{2m}\right) &= -\frac{m+\Delta}{2m} \log\left(\frac{m+\Delta}{2m}\right) - \frac{m-\Delta}{2m} \log\left(\frac{m-\Delta}{2m}\right) \\ &= -\log(e) \left(\frac{m+\Delta}{2m} \left(\ln(0.5) + 2\frac{\Delta}{2m} - \frac{2^2}{2}\left(\frac{\Delta}{2m}\right)^2 \right. \right. \\ &\quad \left. \left. + \frac{2^3}{3}\left(\frac{\Delta}{2m}\right)^3 - \frac{2^4}{4}\left(\frac{\Delta}{2m}\right)^4 + \dots \right) \right. \\ &\quad \left. + \frac{m-\Delta}{2m} \left(\ln(0.5) - 2\frac{\Delta}{2m} - \frac{2^2}{2}\left(\frac{\Delta}{2m}\right)^2 \right. \right. \\ &\quad \left. \left. - \frac{2^3}{3}\left(\frac{\Delta}{2m}\right)^3 - \frac{2^4}{4}\left(\frac{\Delta}{2m}\right)^4 - \dots \right) \right) \\ &= 1 - \log e \left(\left(1 - \frac{1}{2}\right)\left(\frac{\Delta}{m}\right)^2 + \left(\frac{1}{3} - \frac{1}{4}\right)\left(\frac{\Delta}{m}\right)^4 \right. \\ &\quad \left. + \left(\frac{1}{5} - \frac{1}{6}\right)\left(\frac{\Delta}{m}\right)^6 + \dots \right) \\ &= 1 - \log e \left(\frac{1}{2}\left(\frac{\Delta}{m}\right)^2 + \frac{1}{12}\left(\frac{\Delta}{m}\right)^4 + \frac{1}{30}\left(\frac{\Delta}{m}\right)^6 + \dots \right). \end{aligned}$$

Putting this together yields the claimed inequality. ■

ACKNOWLEDGMENTS

I would like to thank Frank Emminghaus for implementation work and Hans-Peter Lenhof for helpful discussions.

REFERENCES

1. K. E. Batcher, Sorting networks and their applications, in "Proceedings, AFIPS Spring Joint Computer Conference" (A. Mattox, Ed.), pp. 307–314, AFIPS Press, Reston, Virginia, 1968.
2. D. T. Blackston, A. Ranade, SnakeSort: A family of simple optimal randomized sorting algorithms, in "Proceedings, Int. Conf. on Parallel Processing," Vol. III (S. Hariri, P. B. Berra, Eds.), pp. 201–204, CRC Press, Boca Raton, Florida, 1993.
3. G. E. Blelloch, L. Dagum, S. J. Smith, K. Thearling, M. Zagha, An evaluation of sorting as a supercomputer benchmark, Technical Report RNR-93-002, NAS Applied Research Branch, Jan. 1993.
4. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, M. Zagha, A comparison of sorting algorithms for the Connection Machine CM-2, in "Proceedings, Symp. on Parallel Algorithms and Architectures," pp. 3–16, ACM Press, New York, 1991.
5. D. Culler, R. Karp, M. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: Towards a realistic model of parallel computation, in "Proceedings, ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming: PPOPP," pp. 1–12, ACM Press, New York, 1993.
6. R. Diekmann, J. Gehring, R. Lülling, B. Monien, M. Nübel, R. Wanka, Sorting large data sets on a massively parallel system, in "Proceedings, 6th IEEE Symp. on Parallel and Distributed Computing," pp. 2–9, IEEE Computer Society Press, Los Alamitos, CA, 1994.
7. W. Feller, "An Introduction to Probability Theory and its Applications," Vol. 1, Wiley, New York, 1957.
8. R. Flajolet, L. Ramshaw, A note on gray code and odd-even merge, *SIAM J. Comput.* **9** (1980), 142–158.
9. W. L. Hightower, J. F. Prins, J. H. Reif, Implementations of randomized sorting on large parallel machines, in "Proceedings, Symp. on Parallel Algorithms and Architectures," pp. 158–167, ACM Press, New York, 1992.
10. K. Hwang, "Advanced Computer Architecture: Parallelism, Scalability, Programmability," McGraw-Hill, New York, 1993.
11. D. E. Knuth, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison Wesley, Reading, MA, 1973.
12. G. Larcher, R. F. Tichy, A note on gray code and odd-even merge, *Discrete Appl. Math.* **18** (1987), 309–313.
13. F. J. MacWilliams, N. J. Sloane, "The Theory of Error Correcting Codes," North-Holland Mathematical Library, Vol. 16, North-Holland, Amsterdam, 1978.
14. D. Nassimi, Y. D. Tsai, An efficient implementation of Batcher's odd-even merge on a SIMD-hypercube, *J. Parallel and Distributed Computing* **19** (1993), 58–63.
15. R. Sedgewick, Data movement in odd-even merging, *SIAM J. Comput.* **7** (1978), 239–272.
16. A. Tridgell, R. Brent, An implementation of a general-purpose parallel sorting algorithm, Technical Report TR-CS-93-01, Computer Sciences Laboratory, Australian National University, 1993.