INFORMATIK

On the Average Running Time of Odd-Even Merge Sort

Christine Rüb

MPI–I–95–1-010 April 1995

ISSN: 0946 - 011X

Forschungsberichte des

Max-Planck-Instituts für Informatik

On the Average Running Time of Odd-Even Merge Sort

Christine Rüb

MPI–I–95–1-010 April 1995

On the Average Running Time of Odd-Even Merge Sort

Christine Rüb*
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
email: rueb@mpi-sb.mpg.de

Abstract

This paper is concerned with the average running time of Batcher's 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 Batcher's 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, 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 merging (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.

1 Introduction

This paper is concerned with a long-known parallel algorithm for sorting, namely Batcher's 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 (for details see Section 2). If this

*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).

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.

In practice, when comparing different sorting algorithms [4, 9, 6], odd-even merge sort has not been used often. 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 of 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 been used often 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. for two sorted lists of length m each 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 analyze the average running time of odd-even merge on a collection of processors. We consider the following machine model: Given are p processors, each with its own local memory. The processors communicate by exchanging messages. For ease of explanation, 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 neighbours. 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 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 of length m each (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 (for details see below). 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 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 *executed.* We will see that if m is large enough, with high probability only few of the $\log p$ steps of odd-even merge will be executed. More precisely, we show that the average number of steps executed by odd-even merge is $O(\log(2+p^2/m))$. From this the conclusion follows that the average running time of this implementation of odd-even merge is $O((m/p)(\log(2+p^2/m)))$.

The odd-even merge network works as follows:
Let $A = A_0, A_1, A_2, ..., A_{m-1}$ and $B = B_0, B_1, B_2, ..., B_{m-1}$ be the two sequences to be merged and denote by $E_0, ..., E_{2m-1}$ the outcome of the merging.
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 with odd indexes. More precisely, merge $A_{\text{even}} = A_0, A_2, A_4, ...$ with $B_{\text{even}} = B_0, B_2, B_4, ...$ into $C = C_0, C_1, C_2, ...$ and $A_{\text{odd}} = A_1, A_3, ...$ with $B_{\text{odd}} = B_1, B_3, ...$ into $D = D_0, D_1, D_2, ...$ 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 on how this is done; we will comment on this later.) 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 $A_0, A_1, A_2, \ldots$ at processors $P_0, P_2, P_4, \ldots$, the list $B_0, B_1, B_2, \ldots$ at processors $P_1, P_3, P_5, \ldots$, and the list $E_0, E_1, E_2, E_3, \ldots$ at processors $P_0, P_1, P_2, P_3, \ldots$ (see Figure 1). By unfolding the recursion we get the following procedure (see, e.g., [15]). Let $p = 2m$.

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

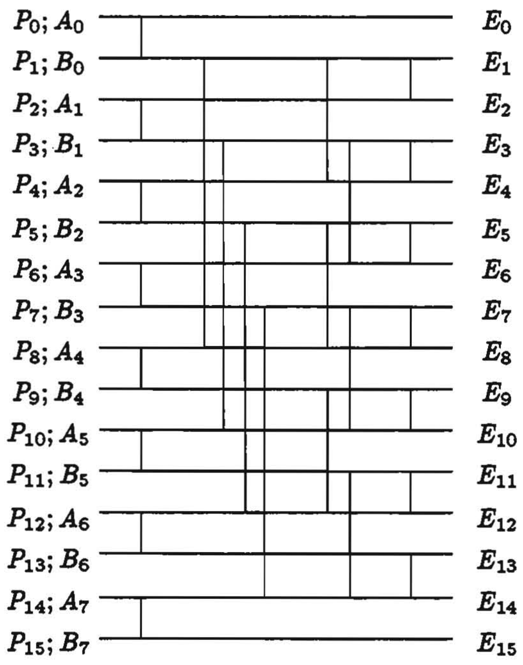


Figure 1. Odd-even merge with 16 processors.

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 such that the even indexed processors hold subsequences of A and the odd indexed processors hold subsequences of B. 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 $2m/p$ elements of all elements stored at P_i and P_j together and P_j gets the largest $2m/p$ elements of these.

That the above procedure merges correctly can be seen as follows. In [11], page 241, exercise 38, it is shown that every comparator network for n elements can be transformed into a sorting algorithm for xn elements by replacing each comparator by the compare–exchange procedure above. Here the input consists of n sorted lists of x elements each. It follows 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 together. Since the transformed sorting network sorts correctly and each input for the merging network is possible, the transformed merging network merges each input correctly.

We still have to specify how to implement *compare–exchange*(P_i, P_j). We want to minimize the amount of communication (which is always relatively expensive). This can be done in the following way. Remember that processor P_i will get the smaller elements and processor P_j will get the larger elements. In the first step, P_i sends its largest element to P_j and P_j sends its smallest element to P_i. This is repeated until the element sent by P_i is smaller than the element sent by P_j. Thus the number of elements exchanged here is at most two plus the number of elements that have to move to a different processor.

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 For $X \in \{A, B, E\}$, let $X^y = X_{y(2m/p)}, X_{y(2m/p)+1}, X_{y(2m/p)+2}, \ldots, X_{y(2m/p)+(2m/p)-1}$.

To derive an upper bound for the average running time of 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 neighbours 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 $(2m/p)$ is large enough. Moreover, these elements will, after the first step, be stored at the "correct" processors (see Figure 1): After the first step processors P_{2i} and P_{2i+1} hold the elements in A (B, resp.) with ranks in A (B, resp.) between $i(2m/p)$ and $(i+1)(2m/p)-1$. 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 $2i(2m/p)$ and $2(i+1)(2m/p)-1$. 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 the first (or largest) i where at least one element has to be exchanged between two processors by i_{max}. 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 $(2^{i_{max}-1} - 1)(2m/p) + 1$. (For example, if all $\log p - 1$ steps of the for-loop are executed, there exists at least one element 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.

Definition Let X be a sorted list. The rank of x in X is the number of elements in X that are smaller than x.

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* $(2^{i-1} - 1)(2m/p) + 1$.

Proof. Let $i_{max} = max\{i$; at least two elements are exchanged in the for-loop$\}$. Thus, before the execution of the 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 $\delta = 2^{i_{max}}$. Let P_k and $P_{k+\delta-1}$ be two processors that exchange elements in this step. Then one of them sends only elements from A and the other one only elements from B (this follows from the observation above.) Assume that P_k sends only elements from A. Let $k = 2j + 1$. The largest element from A that P_k holds is $x = A_{(j+1)(2m/p)-1}$, and the smallest element from B that $P_{k+\delta-1}$ holds is $B_{(j+\delta/2)(2m/p)}$. Thus the rank of x in A differs from the rank of x in B by at least $(j + \delta/2)(2m/p) - ((j + 1)(2m/p) - 1) = (\delta/2 - 1)(2m/p) + 1$. Since $i_{max} \geq i$, the claim follows. □

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

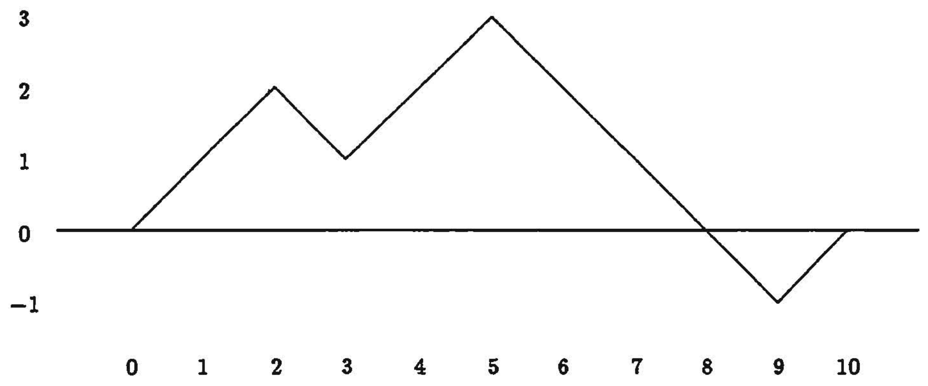


Figure 2: $m = 5$. The path shown corresponds to the input where the outcome of the merging is $AABAABBBBA$.

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 there exists an element x in $A \cup B$ where the rank of x in A differs from the rank of x in B by Δ is at most*

$$2\binom{2m}{m+\Delta+1} \Big/ \binom{2m}{m}.$$

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 Figure 2).

We are interested in all paths that touch or cross the line $y = \Delta + 1$ or the line $y = -(\Delta + 1)$: 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 $2\binom{2m}{m+\Delta+1}$, and the probability that an input has this property is at most $2\binom{2m}{m+\Delta+1}/\binom{2m}{m}$. □

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

Lemma 3

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

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 steps executed in odd-even merge.

Lemma 4 *The average number of steps executed 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 executed is bounded by $e^{-2^{2\delta}}$.*

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

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

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

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

for $A(m,p)$. Let $c = 2\sqrt{16/3\pi}$. Since $i \le \log p - 2$, this is

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

The last term of this behaves as follows: 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 $\lfloor \log(1+\sqrt{(1+\ln(c))p^2/4m})\rfloor$ i's 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

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

The second claim follows directly from the above discussion. □

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

	0.99					0.999				
no. proc.\ no. steps	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 as follows. 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 processors locally sorts its portion of the input. In the following $\log p$ steps these sorted lists are merged together in the following way. In step i, $0 \leq i \leq \log p - 1$, the lists stored at processors $P_j, P_{j+\delta}, P_{j+2\delta}, \ldots$ and processors $P_{j+\delta/2}, P_{j+\delta/2+\delta}, P_{j+\delta/2+2\delta}, \ldots$ are merged, where $\delta = p/2^i$ and $0 \leq j < p/2^{i+1}$ (see Figure 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 where elements have to be exchanged.

Lemma 6 *Let* $c = 2\sqrt{16/3\pi}$. *In the* j*-th call of odd-even merge,* $0 \leq j \leq \log p - 1$, *the average number of steps where elements are exchanged is bounded by* $1.39 + \lceil \log(1 + \sqrt{(1 + \ln(cp/2^{j+1}))p2^j/n}) \rceil$.

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

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

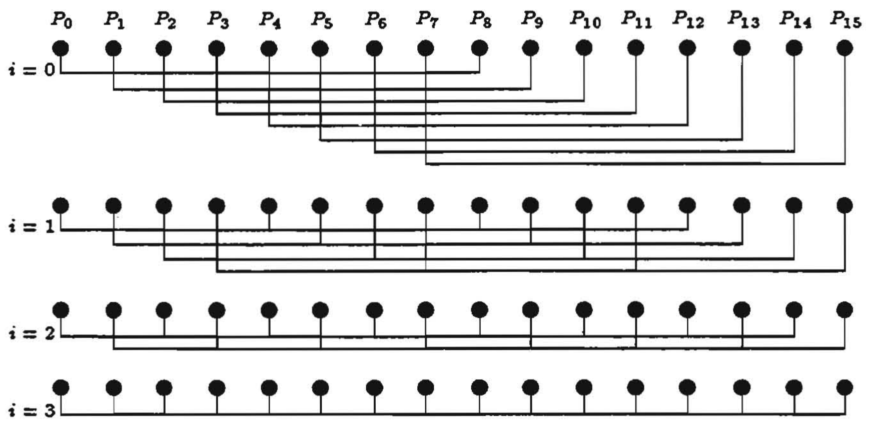

P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}
$i = 0$
$i = 1$
$i = 2$
$i = 3$

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

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

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

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

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

$$1 + z + \sum_{i=z}^{j-1} e^{-2\left(2^i-1\right)^2 n/\left(p2^{j+1}\right) + \ln\left(cp/2^{j+1}\right)}$$

$$\leq 1 + z + \sum_{i=0}^{j-1-z} e^{-2\left(2^x 2^i-1\right)^2 n/\left(p2^{j+1}\right) + \ln\left(cp/2^{j+1}\right)}$$

$$= 1 + z + \sum_{i=0}^{j-1-z} e^{-2\left(2^i\left(1+\sqrt{(1+\ln(cp/2^{j+1}))p2^j/n}\right)-1\right)^2 n/\left(p2^{j+1}\right) + \ln\left(cp/2^{j+1}\right)}$$

$$\leq 1 + z + \sum_{i=0}^{j-1-z} e^{-2^{2i}}$$

$$\leq 1.39 + z. \quad \square$$

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 where 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 $c = 2\sqrt{16/3\pi}$. According to Lemma 6, the average number of steps with data exchanges is

$$\sum_{i=0}^{\log p-1}\left(1.39 + \left\lceil \log\left(1 + \sqrt{(1+\ln(cp/2^{i+1}))p2^i/(n)}\right)\right\rceil\right)$$

$$\leq \log p\left(1.39 + \left\lceil \log\left(1 + \sqrt{(1+\ln(c))p^2/(2n)}\right)\right\rceil\right)$$

$$\leq \log p\left(1.39 + \left\lceil \log\left(1 + \sqrt{p^2/n}\right)\right\rceil\right).\square$$

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.*

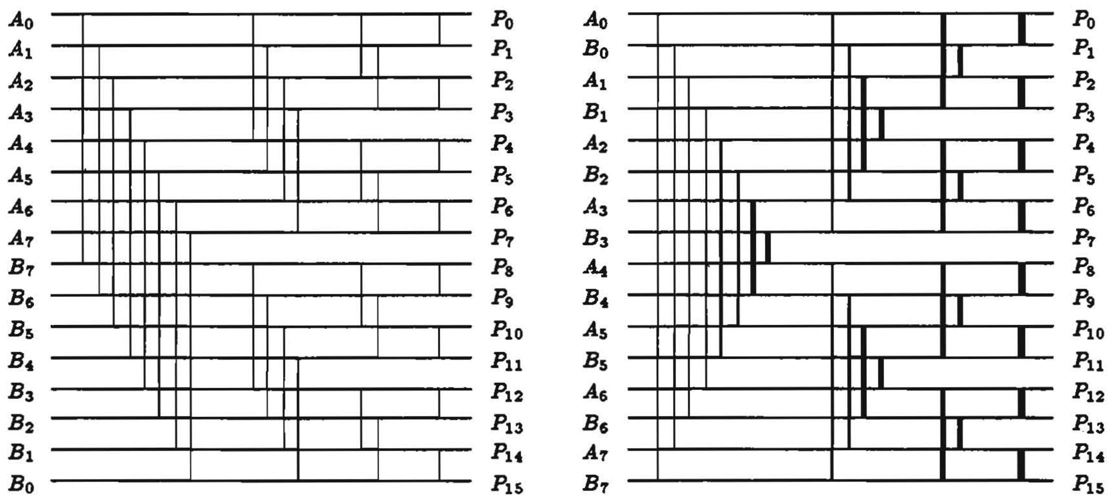
4 Revisiting Bitonic Merge Sort

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 $A = A_0, A_1, A_2, ..., A_{m-1}$ and $B = B_0, B_1, B_2, ..., B_{m-1}$ be the two sequences to be merged and denote the outcome of the merging by $E_0, ..., E_{2m-1}$. If $m = 1$, compare and exchange, if necessary, A_0 and B_0. Else, merge $A_{even} = A_0, A_2, A_4, ...$ with $B_{odd} = B_1, B_3, ...$ into $C = C_0, C_1, C_2, ...$ and $A_{odd} = A_1, A_3, ...$ with $B_{even} = B_0, B_2, B_4, ...$ into $D = D_0, D_1, D_2, ...$ 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 reversed order. Figure 4.a 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,



A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}
A_0 B_0 A_1 B_1 A_2 B_2 A_3 B_3 A_4 B_4 A_5 B_5 A_6 B_6 A_7 B_7
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}

a) original bitonic merge b) modified bitonic merge

Figure 4. Bitonic merge with 16 processors.

this will take time $\Omega((n/p)\log p)$. Thus, the problem is that one of the lists is stored in reversed order.

But what happens if we store the input elements in the same order as we did for odd-even merge? Figure 4.b shows the communication pattern for this alternative when using 16 processors. The links between the processors are drawn such 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 endpoint of at most two links drawn in the same thickness. We can again argue that the running time of this merging procedure is $O((n/p)\log \Delta)$, 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.

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 where 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 maximal 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; e.g., the speedup on the iPSC/860 for 100,000 longs per processor and using 16 processors is 12.3 or 77% of the maximal possible; for 1,600,000 long per processor it is 14.5 or 90.6% of the maximal possible. For small inputs the speedups are not as high as one could, perhaps, expect from the asymptotic upper bounds. E.g., for 300 longs per processor and 16 processors of the iPSC/860 the speedup is 1.5, that is, 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), the full bandwidth of the communication links can be utilized. (All of the above listed speedups 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. However, 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 make 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 "presorted" 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 executes fewer steps but sends more elements.)

For large inputs, sample sort (see, e.g., [3]) will have a smaller 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.

Acknowledgements

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

Appendix

Lemma 3

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

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

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

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

$$\ln(x) = \ln(0.5) + 2(x-0.5) - \frac{2^2}{2}(x-0.5)^2 + \frac{2^3}{3}(x-0.5)^3 - \frac{2^4}{4}(x-0.5)^4 + \ldots$$

and thus

$$H_2\left(\frac{m+\Delta}{2m}\right) = -\frac{m+\Delta}{2m}\log\left(\frac{m+\Delta}{2m}\right) - \frac{m-\Delta}{2m}\log\left(\frac{m-\Delta}{2m}\right)$$

$$= -\log(e)\left(\frac{m+\Delta}{2m}\left(\ln(0.5) + 2\frac{\Delta}{2m} - \frac{2^2}{2}\left(\frac{\Delta}{2m}\right)^2 + \frac{2^3}{3}\left(\frac{\Delta}{2m}\right)^3 - \frac{2^4}{4}\left(\frac{\Delta}{2m}\right)^4 + \ldots\right) + \right.$$
$$\left. \frac{m-\Delta}{2m}\left(\ln(0.5) - 2\frac{\Delta}{2m} - \frac{2^2}{2}\left(\frac{\Delta}{2m}\right)^2 - \frac{2^3}{3}\left(\frac{\Delta}{2m}\right)^3 - \frac{2^4}{4}\left(\frac{\Delta}{2m}\right)^4 - \ldots\right)\right)$$
$$= 1 - \log e\left(\left(1-\frac{1}{2}\right)\left(\frac{\Delta}{m}\right)^2 + \left(\frac{1}{3}-\frac{1}{4}\right)\left(\frac{\Delta}{m}\right)^4 + \left(\frac{1}{5}-\frac{1}{6}\right)\left(\frac{\Delta}{m}\right)^6 + \ldots\right)$$
$$= 1 - \log e\left(\frac{1}{2}\left(\frac{\Delta}{m}\right)^2 + \frac{1}{12}\left(\frac{\Delta}{m}\right)^4 + \frac{1}{30}\left(\frac{\Delta}{m}\right)^6 + \ldots\right).$$

Putting this together yields the claimed inequality. □

References

[1] K.E.. Batcher. Sorting networks and their applications. *Proceedings, AFIPS Spring Joint Computer Conference*, 307–314, 1968.

[2] D.T. Blackston, A. Ranade. SnakeSort: a family of simple optimal randomized sorting algorithms. *Proc., Int. Conf. on Parallel Processing*, Vol. III, 201–204, 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. *Proc., Symp. on Parallel Algorithms and Architectures*, 3–16, 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. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming: PPOPP*, 1993.

[6] R. Diekmann, J. Gehring, R. Lülling, B. Monien, M. Nübel, R. Wanka. Sorting large data sets on a massively parallel system. Technical Report, Universität Paderborn.

[7] W. Feller. An introduction to probability theory and its applications, Vol. 1. New Wiley series in probability and mathematical statistics, 1957, Wiley, New York.

[8] R. Flajolet, L. Ramshaw. A note on gray code and odd-even merge. *SIAM J. Comput.*, Vol. 9, 142–158, 1980.

[9] W.L. Hightower, J.F. Prins, J.H. Reif. Implementations of randomized sorting on large parallel machines. *Proc. Symp. on Parallel Processing*, 158–167, 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 series in computer science and information processing, 1973, Addison Wesley, Reading, Mass.

[12] G. Larcher, R.F. Tichy. A note on gray code and odd-even merge. *Discrete Applied Mathematics 18*, 309–313, 1987.

[13] F.J. MacWilliams, N.J. Sloane, The Theory of Error Correcting Codes. North-Holland Mathematical Library,Vol. 16, North-Holland, Amsterdam - New York - Oxford, 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*, Vol. 19, 58–63, 1993.

[15] R. Sedgewick. Data movement in odd-even merging. *SIAM J. Comput.*, Vol. 7, 239–272, 1978.

[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.