On Batcher's Merge Sorts as Parallel Sorting Algorithms

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

Abstract

In this paper we examine the average running times of Batcher's bitonic merge and Batcher's odd-even merge when they are used as parallel merging algorithms. It has been shown previously that the running time of odd-even merge can be upper bounded by a function of the maximal rank difference for elements in the two input sequences. Here we give an almost matching lower bound for odd-even merge as well as a similar upper bound for (a special version of) bitonic merge. From this follows that the average running time of odd-even merge (bitonic merge) is $\Theta((n/p)(1+\log(1+p^2/n)))$ ($O((n/p)(1+\log(1+p^2/n)))$, resp.) where n is the size of the input and p is the number of processors used. Using these results we then show that the average running times of odd-even merge sort and bitonic merge sort are $O((n/p)(\log n + (\log(1+p^2/n))^2))$, that is, the two algorithms are optimal on the average if $n \geq p^2/2^{\sqrt{\log p}}$. The derived bounds do not allow to compare the two sorting algorithms directly, thus we also present experimental results, obtained by a simulation program, for various sizes of input and numbers of processors.

1 Introduction

Batcher's bitonic merge sort and odd-even merge sort are two well known comparator networks for sorting [1, 11]. The two networks can easily be converted into parallel sorting algorithms where each processor holds more than one element by replacing comparisons between input elements by a split–and–merge procedure [11]. The running time of a straightforward implementation will then be $O((n/p)(\log n + \log^2 p))$, where n is the size of the input and p is the number of processors used. Although this is not optimal, the constants involved are small and the two algorithms can be the fastest available for small input sizes.

In contrast to odd-even merge sort, bitonic merge sort has often been used in comparative studies of sorting algorithms for parallel computers [3, 9, 5, 7]. Because of the small constant factors, here bitonic merge sort proved to be the fastest sorting algorithm for small input sizes.

In [12] it has been shown that the average running time of the odd-even merge (odd-even merge sort) algorithm can be improved much by keeping the amount of communication to a minimum. The resulting running times are $O((n/p)(1 + \log(1 + p^2/n)))$ for merging and

*Supported by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität.

$O((n/p)(\log n + \log p \log(1+p^2/n)))$ for sorting. (In the case of merging (sorting) we assume that each outcome of the merging (each permutation of the input elements, resp.) is equally likely.) In the meantime this version of odd-even merge sort has been used in two comparative studies of sorting [4, 14]. In both cases, odd-even merge sort was the fastest sorting algorithm among those considered for some input size.

In [4] the derived bounds from [12] were also used to predict the running time of the implementation. However, it turned out that the predictions were much too pessimistic. In fact, as we will show in this paper, the average running time of the odd-even merge sort algorithm can be bounded by $O((n/p)(\log n + (\log(1+p^2/n))^2))$, and thus the odd-even merge sort algorithm is optimal on the average if $n \geq p^2/2^{\sqrt{\log p}}$. We will also show that the running time of the odd-even merge algorithm is closely related to the maximal rank difference for elements in the two input sequences. From this we derive nearly matching upper and lower bounds for its average running time.

In [13] it was pointed out that the average running time of bitonic merge sort can be improved by storing the input elements such that the smaller indexed processor always receives the smaller elements and again keeping the amount of communication at a minimum (the corresponding merging network is known as the balanced merger [6]). This means that an already established ordering among elements will be preserved. In this paper we will show that the average running time of the order-preserving bitonic merge algorithm can be upper bounded by a function of the maximal rank difference of elements in the input sequence. From this we obtain $O((n/p)(1+\log(1+p^2/n)))$ ($O((n/p)(\log n + (\log(1+p^2/n))^2))$, resp.) as upper bound for the average running time of the order-preserving bitonic merge (bitonic merge sort, resp.) algorithm.

The results obtained in this paper do not allow to compare the behaviour of the odd-even merge sort algorithm and the order-preserving bitonic merge sort algorithm directly. Thus we also present experimental results that compare the average running times of the two algorithms for varying sizes of input and numbers of processors. For these experiments we do not use a specific computer model; rather, we assume that the processors form a complete graph and that the number of exchanged elements determines the running time. The actual average running time of an implementation depends strongly on the parallel machine used and such an investigation is beyond the scope of this paper. However, we hope the results obtained here will help to estimate actual average running times in the future.

The obtained upper bounds on the running times show that the two merge sort algorithms can be very fast with a large number of processors and a moderate input size. The experimental results show that the upper bounds are too pessimistic; this is largely due to the fact that we wanted to obtain a closed formulation.

This paper is organized as follows. Section 2 defines comparator network based parallel algorithms and Section 3 shows how we define running time. Section 4 is concerned with odd-even merge, Section 5 with bitonic merge, and Section 6 with the two merge sort algorithms. Section 7 gives some experimental results and Section 8 contains some conclusions.

2 Comparator Network Based Parallel Algorithms

The algorithms considered here are based on comparator networks. A comparator network consists solely of comparators and wires. Each comparator network for an input of size n can be drawn as a diagram consisting of n horizontal lines that are connected by vertical links, where each vertical link corresponds to a comparator [11].

A comparator network for merging (sorting) p elements can be turned into a parallel algorithm for merging (sorting, resp.) $n = pr$ elements using p processors as follows. The processors are assigned to the horizontal lines and each comparator that connects the lines of two processors is replaced by a procedure that sends the smallest r elements stored at both processors together to one processor and the other r elements to the other processor.

To improve the running time of the algorithm, the exchange procedure can be implemented as follows. If the procedure is called for processors P_i and P_j, they first determine whether they have to exchange any elements in this step. This can be done in time $O(1)$ by sending the largest and smallest elements of the lists. If elements have to be exchanged, the processors either exchange all their elements at once or use binary search to determine how many elements have to be exchanged and afterwards exchange exactly these elements. As we will see, this can reduce the average running time of an algorithm much.

In the remainder of this paper we mean by odd-even merge (sort) and bitonic merge (sort) parallel procedures that use the above explained exchange procedure. Additionally we imply that the input elements are stored such that the smaller elements are always sent to the processor with the smaller index. (In the case of bitonic merge the corresponding merging network is known as the balanced merger [6]. In [2] it was shown that the bitonic merge network and the balanced merger are essentially the same networks.) We will also assume that the number p of processors used is a power of two. The algorithms considered in this paper can also be used if p is not a power of two: let p' be the next larger power of two. Use the algorithm for p' processors. Every time a processor P_i has to communicate with a processor P_j where $j \geq p$, P_i does nothing. Since the larger elements are always sent to the larger indexed processor, this procedure will merge or sort the input correctly. The given bounds on the running times hold with p replaced by p'.

3 What do we measure?

Any parallel algorithm consists of two parts, namely computation and communication. In many parallel computers existing today the constants involved in communication are much larger than the constants involved in computation (e.g., Intel's Paragon or Cray's T3D), but there are also computers where the constants are approximately the same (e.g., MasPar's MP-1). Thus the constants involved in the two parts can be quite different and we treat them separately.

We will make statements of the following kind. Let *alg* stand for an algorithm, let n be the size of the input and let p be the number of processors used. Each processor stores $r = n/p$ elements of the input. The algorithm consists of T steps where in each step pairs of processors communicate and possibly exchange elements. We define the functions t_{alg} and R_{alg} as follows (for ease of notation later on we number the steps from T down to 1).

Definition 1 $t_{alg}(T+1, j) = 0$, $0 \leq j \leq p-1$.
$t_{alg}(i,j) = \max\{t_{alg}(i+1, j), t_{alg}(i+1, k)\} + x(j,k)$, $1 \leq i \leq T$, *where* P_k *is the processor that communicates with* P_j *in step* i *and* $x(j,k)$ *is the number of elements* P_j *and* P_k *exchange. If* P_j *does not communicate with any processor in step* i, $t_{alg}(i,j) = t_{alg}(i+1,j)$.
Finally, $R_{alg}(r,p) = \max\{t(1,j);\, 0 \leq j \leq p-1\}$.

A lower bound for R_{alg} will always be a lower bound for the running time of the algorithm. For our upper bounds on R_{alg} we assume in most cases that $x(i,j) = r$ whenever the two communicating processors exchange an element. By doing this, we neglect the communication network but we account for the local computation time. In this formulation we also neglect the time it takes to determine whether two processors exchange elements which leads to additional $\Theta(T)$ time. We do this because the constant involved in this part of the algorithms can be much larger than the other constants because of startup times.

4 Odd-even merge

In this and the following sections p always denotes the number of processors used.

In this section we show that the running time of odd-even merge is closely related to the maximal rank difference of elements in the two input sequences. Namely, we show the following. Let A and B be the two sorted sequences to be merged and let $d_{max} = \max\{|$ rank of x in A $-$ rank of x in $B|;\ x \in A \cup B\}$. Then the running time of odd-even merge is $\Theta((n/p)(1 + \log(1 + d_{max}p/n)))$ where n is the size of the input. (A similar, but somewhat weaker, upper bound has been proved in [13].) From these bounds upper and lower bounds for the average running time of odd-even merge follow.

From the odd-even merge network we can derive the following parallel merge procedure (see Figure 1). Assume we want to merge two sorted lists A and B of length m such that the even indexed processors hold subsequences of A and the odd indexed processors hold subsequences of B. Each processor holds $2m/p =: r$ elements. The following procedure will merge the two lists.

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 smallest r elements stored at P_i and P_j, and P_j gets the largest r elements.

First we will analyze the running time of the for-loop of the odd-even merge algorithm. For ease of notation, we call the step of the for-loop where the index i has a certain value k, step k of the for-loop or step k of the algorithm. Correspondingly, the first step of odd-even merge will also be called step $\log p$.

Let $A^j = A_{jr}, A_{jr+1}, ..., A_{(j+1)r-1}$ and $B^j = B_{jr}, B_{jr+1}, ..., B_{(j+1)r-1}$, $0 \le j \le p-1$. We define i_{max} as the first (or maximal) i where at least two elements are exchanged in the for-loop of the algorithm and e_{max} as the maximal number of elements that any processor exchanges in step i_{max}. The following lemma gives upper and lower bounds on d_{max} that depend on i_{max} and e_{max}.

Lemma 1
1. $d_{max} \ge (2^{i_{max}-1} - 1)r + 2e_{max}$.
2. *If* $e_{max} < r$ $(e_{max} = r)$, $d_{max} \le 2^{i_{max}-1}r + 2e_{max}$ $(d_{max} \le 2^{i_{max}}r$, *resp.).*
3. *Let* $(2^{j-1} + 2)r \le d_{max} \le (2^j - 1)r + 1$. *Then* $i_{max} = j$ *and* $e_{max} = r$.

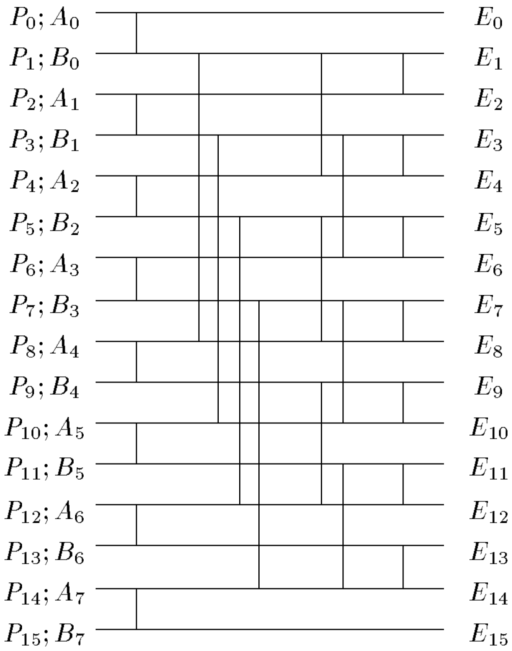


Figure 1: Odd-even merge with 16 processors.

Proof. Let $\delta = 2^{i_{max}}$. Before the execution of step i_{max} processors P_{2j} and P_{2j+1} hold the elements of $A^j \cup B^j$, $0 \leq j < p/2$. Also, in step i_{max} only elements from A will be exchanged by elements from B (this is not necessarily the case in later steps).

1. (A simpler version of this claim has been proved in [12].) Let P_{2j+1} and $P_{2j+\delta}$ be two processors that exchange e_{max} elements in step i_{max}. Assume that P_{2j+1} sends e_{max} elements from A. The largest element from A that is sent by P_{2j+1} is $A_{(j+1)r-e_{max}} =: z$ and the smallest element from B that is sent by $P_{2j+\delta}$ is $B_{(j+\delta/2)r+e_{max}-1}$. Thus, the rank of z in A differs from the rank of z in B by $(j+\delta/2)r + e_{max} - ((j+1)r - e_{max}) = (\delta/2 - 1)r + 2e_{max} = (2^{i_{max}-1} - 1)r + 2e_{max}$ and the claim follows.

2. First suppose that r elements from B are moved from processor P_{2j+1} to processor $P_{2j+\delta}$ (the case that elements from A are moved is symmetric). At the end of the for-loop these elements can not be stored at a larger processor than $P_{2j+\delta}$ (a proof of this fact can be found in Lemma 2). Thus the smallest of these r elements, namely B_{jr}, has rank at most $(2j+\delta)r - jr = (j+\delta)r$ in A, and its rank difference is at most $(j+\delta)r - jr = \delta r = 2^{i_{max}}r$. Note that this is the largest possible rank difference in this case.

Next assume that $e_{max} < r$ and suppose that e_{max} elements from B are moved from processor P_{2j+1} to processor $P_{2j+\delta}$ (again, the other case is symmetric). If $2j+\delta = p-2$, the exchanged elements from B belong to $B^{p/2-1-\delta/2}$. Since their rank in A is at most $p/2 - r$, the claim follows by a simple calculation. Thus suppose that $2j+\delta < p-2$. Then the e_{max} exchanged elements form B^j can only be larger than e_{max} elements of $A^{j+\delta/2+1}$, since else processors P_{2j+3} and $P_{2j+2+\delta}$ would exchange more than e_{max} elements. Thus the rank of the smallest of the exchanged elements, namely $B_{(j+1)r-e_{max}}$, in A is at most $(j+1+\delta/2)r+e_{max}$ and its rank difference is at most $(j+1+\delta/2)r+e_{max}-((j+1)r-e_{max}) = (\delta/2)r+2e_{max} = 2^{i_{max}-1}r+2e_{max}$. Again observe that this is the largest rank difference possible in this case.

3. First assume that indeed $i_{max} = j$, but that $e_{max} < r$. According to Claim 2, $d_{max} \leq 2^{i_{max}-1} + 2e_{max} < (2^{i_{max}-1} + 2)r = (2^{j-1} + 2)r$, which is a contradiction.

Next assume that $i_{max} < j$. According to Claim 2, $d_{max} \leq 2^{i_{max}} r \leq 2^{j-1} r < (2^{j-1} + 2)r$, which is again a contradiction.

Finally assume that $i_{max} > j$. According to Claim 1, $d_{max} \geq (2^{i_{max}-1} - 1)r + 2 > (2^j - 1)r + 1$, and thus Claim 3 holds. $\square$

Lemma 1.1 gives immediately rise to an upper bound on the running time that depends on d_{max}: since i_{max} is the first step of the for-loop that is executed, no processor can exchange more than $(i_{max} - 1)r + e_{max}$ elements during the execution of the for-loop. Lemma 1.2 shows that d_{max} cannot be too large; however, this does not give rise to a lower bound of the running time: the remaining $i_{max} - 1$ steps could be executed in less than $(i_{max} - 1)r$ time.

In the following we proof a lower bound for the running time of odd-even merge that depends on d_{max}. To this end we first examine which paths an element takes during the execution of odd-even merge, and which paths are particularly expensive.

Lemma 2

1. *Let x be stored at processor P_j before a step of the for-loop and at processor P_k after the step, $j \neq k$. Then x cannot move farther away from P_j than P_k in the remaining steps of the for-loop, and it can never return to P_j.*

2. *Let x be stored at processor P_j before the execution of the for-loop and at processor P_k at the end of the for-loop. Then the path x takes during the execution of the for-loop is uniquely determined by j and k.*

Proof.

1. To see this, first observe that in all steps of the for-loop processors with odd indexes communicate with processors with even and larger indexes. Thus x, if it is exchanged, alternates between odd and even indexed processors. Since the distances between the processors become smaller in each step, x can never be moved to a processor that is farther away from P_j than P_k or return to P_j.

2. This follows from part 1 and the fact that the distance between communicating processors is more than halved in each step of the for-loop. $\square$

Lemma 3 *Let x be stored at P_j at the beginning of the for-loop and at $P_{j+\delta}$ at the end of the for-loop where $\delta = \pm 2/3(2^i - 1 + \alpha)$, $\alpha \in \{0, 1/2\}$ and $i \in N$. Then x will be moved to another processor exactly in the last i steps of the for-loop.*

Proof. First note that i is even iff $\delta = \pm 2/3(2^i - 1)$ and i is odd iff $\delta = \pm 2/3(2^i - 1/2)$. This follows because δ and i are natural numbers.

According to Lemma 2, the steps of the for-loop in which x is moved to a different processor are uniquely determined by j and $j + \delta$. Thus it suffices to show that $j + \delta$ is the processor that can be reached from P_j during the last i steps of the for-loop. First suppose that $\alpha = 0$. Let P_k be the processor that can be reached from P_j by using all of the last i steps. Then

$$\begin{aligned} |k - j| &= (2^i - 1) - (2^{i-1} - 1) + ... + (2^2 - 1) - (2^1 - 1) \\ &= 2/3(2^i - 1). \end{aligned}$$

Next suppose that $\alpha = 1/2$. Again, let P_k be the processor that can be reached from P_j by using all of the last i steps. Then

$$\begin{aligned} |k-j| &= (2^i - 1) - (2^{i-1} - 1) + \ldots - (2^2 - 1) + (2^1 - 1) \\ &= 2/3(2^i - 1/2). \end{aligned}$$ □

The following two lemmas show that if one element is moved a certain number z of processors forward during the for-loop, we can always find a subsequence Z of the input (i.e., $Z = A^j$ or $Z = B^j$) where all elements of Z are moved y or $1 + y$ processors forward for all $y \leq z - 2$. From this we can then derive a lower bound for the running time of odd-even merge.

Lemma 4 *Let B_j have rank $j + \delta$ in A and let B_k have rank $k + \gamma$ in A, $j < k$ and $\delta > \gamma$. Let $\delta > \beta > \gamma$. Then there exists an index l, $j < l < k$, such that B_l has rank $l + \beta$ in A, that is, every rank difference between δ and γ occurs between B_j and B_k. A similar claim holds for A and B interchanged.*

Proof. Let B_q have rank $q + \alpha$ in A. Consider B_{q+1}. Then either B_{q+1} follows directly after B_q in $A \cup B$, or $z \geq 1$ elements of A lie between B_q and B_{q+1}. In the first case B_{q+1} has rank $q + \alpha = (q+1) + (\alpha - 1)$ in A and in the second case it has rank $q + \alpha + z = (q+1) + \alpha + (z-1)$ in A. That is, in the first case the rank difference is decreased by one whereas in the second case the rank difference can not be decreased.
Since the rank difference can not decrease by more than one, it follows that all rank differences between δ and γ occur between B_j and B_k. □

Lemma 5 *Let x be an element that is moved from processor P_{2j+1} to processor $P_{2j+1+\alpha}$ during the execution of the for-loop, $\alpha \geq 2$. Let $0 \leq \beta \leq \alpha - 2$. Then there exists a processor P_{2k+1}, $k > j$, where all elements stored at processor P_{2k+1} before the execution of the for-loop will be stored at processor $P_{2k+1+\beta}$ or at processor $P_{2k+2+\beta}$ at the end of the for-loop.*

Proof. Assume that $x \in B$. Then x belongs to B^j (the case that $x \in A$ is symmetric). Let $q > j$ be maximal such that B^t is stored at processor P_{2t+1} for all t, $j < t \leq q$, before the execution of the for-loop. The index q exists because $\alpha \geq 2$: Assume that $x = B_{(j+1)r-1}$. The rank of x in $A \cup B$ is at least $(2j + 1 + \alpha)r$ and thus $x > A_{(j+\alpha)r}$. Since $\alpha \geq 2$, $B_{(j+1)r} > A_{(j+2)r}$ and $P_{2(j+1)+1}$ stores B^{j+1} before the execution of the for-loop. We will show that $j < k \leq q$.

First observe that either $q \leq p-2$ and $B_{(q+1)r} < A_{(q+2)r-1}$, and thus B_{qr}'s rank difference is at most $(q+2)r - 1 - qr = 2r - 1$, or that else $q = p - 1$ and thus the rank difference for B_{qr} is r. On the other hand, the difference of ranks for x is at least $(2j+1+\alpha)r - 2((j+1)r - 1) = (\alpha - 1)r + 2$.

We next show that there exists an index u, $j < u \leq q$, where the first element of B^u will be stored at processor $P_{2u+1+\beta}$ at the end of the for-loop. Assume otherwise. Then, for all t, $j < t \leq q$, B_{tr}'s rank in $A \cup B$ will be either at least $(2t + 2 + \beta)r$ or at most $(2t + 1 + \beta)r - 1$. That means that B_{tr}'s rank in A is at least $(t + 2 + \beta)r$ or at most $(t + 1 + \beta)r - 1$, and that the difference of ranks for B_{tr} is at least $(2 + \beta)r$ or at most $(1 + \beta)r - 1$. On the other hand, the rank difference for $B_{(j+1)r}$ is at least $(\alpha - 1)r + 1 > (1 + \beta)r - 1$ and thus at least $(2 + \beta)r$.

Since $(2+\beta)r-((1+\beta)r-1)=r+1$, if follows from Lemma 4 and by induction that the rank difference for B_{tr} is at least $(2+\beta)r \geq 2r$ for all t, $j < t \leq q$, which is a contradiction.

Thus, let w, $j < w \leq q$ be maximal such that B_{wr} ends in $P_{2w+1+\beta}$. Assume there exists an element in B^w that does not end in $P_{2w+1+\beta}$ or in $P_{2w+2+\beta}$. Then $B_{(w+1)r}$ will end at least in $P_{2w+\beta+3} = P_{2(w+1)+1+\beta}$. However, because w is chosen maximal, this can not happen. Thus $B_{(w+1)r}$ lands at least in $P_{2w+4+\beta} = P_{(2(w+1)+2+\beta)}$ and $B_{(w+1)r}$'s rank difference is at least $2((w+1)+2+\beta)r-(2(w+1)r)=(2+\beta)r$. This is a contradiction, since we can again argue that B_{qr}'s rank difference is at least $(2+\beta)r$. □

By combining Lemma 3 and Lemma 5, we can show a relationship between d_{max} and the maximal distance a subsequence of A or B has to travel.

Lemma 6 *If $d_{max} \geq (2/3)(2^i+7/2)r$, $i \geq 2$, there exists a subsequence A^j of A or a subsequence B^j of B, where A^j (B^j, resp.) is moved to a different processor during steps i through 2 of the for-loop.*

Proof. Let $\beta = (2/3)(2^i-1)$if i is even and let $\beta = (2/3)(2^i-2)$ if i is odd. Let $\alpha = \beta+2 \leq (2/3)(2^i+2)$. Let x be an element with rank difference d_{max}. We assume that $x = B_s$ and $B_s > A_s$. Thus x's rank in $A \cup B$ is at least $2s+(2/3)(2^i+7/2)r$. Let $s = tr+u$, $0 \leq u < r$. Then $2s+(2/3)(2^i+7/2)r = 2(tr+u)+(2/3)(2^i+7/2)r \geq (2t+(2/3)(2^i+2)+1)r \geq (2t+1+\alpha)r$. Thus x will be moved from processor P_{2t+1} to at least processor $P_{2t+1+\alpha}$. According to Lemma 5, there will be a subsequence of B that is moved from processor P_{2j+1} to processors $P_{2j+1+\beta}$ and $P_{2j+2+\beta}$ during the execution of the for-loop. Because of the choice of β, this subsequence will thus be moved to a different processor during steps i through 2 of the for-loop: if i is even (odd), processor $P_{2j+2+\beta}$ (processor $P_{2j+1+\beta}$, resp.) will be reached by step 2, and processor $P_{2j+1+\beta}$ (processor $P_{2j+2+\beta}$, resp.) will be reached by step 1. □

Next we put together the results shown above to proof lower and upper bounds on the running time of the for-loop of odd-even merge that depend on d_{max}.

Definition 2 *Let $t_{\text{for}}(i,j)$, $1 \leq i \leq \log p, 0 \leq j \leq p-1$ and $R_{\text{for}}(r,p)$, be defined as described in Section 3 where $alg =$* for *stands for the for-loop of odd-even merge.*

Theorem 1 *Let $T = (\lfloor \log(\max\{2, d_{max}/r \cdot 3/2 - 7/2\}) \rfloor - 1)r$, and let $S = (\lfloor \log(d_{max}/r + 1) \rfloor + 1)r$.*
1. $T \leq R_{\text{for}}(r,p) \leq S$.
2. *If $d_{max} \leq r$, $R_{\text{for}}(r,p) \leq d_{max}/2$.*
3. *Let $d_{max}/r = 2^x y - 1$, $1 \leq y < 2$ (thus $S = (x+1)r$). Let $d_{max}/r \cdot 3/2 - 7/2 \geq 0$. If $(3/2)y - 5/2^x \geq 2$, $S - T = r$. If $(3/2)y - 5/2^x < 2$, $S - T = 2r$.*
4. $S - T \in \{r, 2r\}$.

Proof. Claim 1 follows from Lemma 1.1 and Lemma 6 and Claim 2 follows from Lemma 1.1. Claim 3 follows from Claim 1 by simple calculations, and Claim 4 follows directly from Claim 3. □

We want to use the relationship between the running time of odd-even merge and the maximal rank difference in the input from Theorem 1 to derive bounds for the average running time of odd-even merge. Till now we only considered the running time of the for-loop of odd-even merge; now we include the first step, before the for-loop.

Definition 3 *Let $t_{\text{odd}}(i,j)$, $1 \le i \le \log p + 1, 0 \le j \le p-1$ and $R_{\text{odd}}(r,p)$, be defined as described in Section 3 where $alg = \text{odd}$ stands for odd-even merge.*
Let $P(\Delta)$ be 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.
For $\Delta > 0$, let $Q_A(\Delta)$ be the probability that there exists an element x in A where the rank of x in B minus the rank of x in A is Δ.

Theorem 2

$$r\left(Q_A(2r-1) + \sum_{i=2}^{\lfloor \log((3/2)p-7)\rfloor -1} Q_A\left(\left(2^{i+1}+7\right)r/3\right)\right)$$
$$\le \operatorname{Exp}(R_{\text{odd}})$$
$$\le r\left(1 + \sum_{i=1}^{\log p-1} P\left(\left(2^{i-1}-1\right)r\right)\right)$$

Proof. If an element x in A has rank difference d_{max} and x's rank in B is larger than x's rank in A, the subsequence A_j from Lemma 6 will be moved in the first step of odd-even merge. Also, if $d_{max} \ge 2r-1$, at least one subsequence of A will be sent to a different processor in the first step. This can be seen as follows. Assume that A_x has rank difference $2r-1$. If $x = jr$ for a $j \ge 0$, A^j will be moved in the first step. If $x = jr + k$, $q \le k < r$, $A_{(j+1)r}$ has a rank difference of at least r and thus A^{j+1} will be moved in the first step. Thus the two inequalities follow from Lemma 6 and Lemma 1. □

Theorem 2 gives bounds for the average running time of odd-even merge in terms of the probability that a certain rank difference occurs. To arrive at a closed formulation, we have to substitute the probabilities by closed formulas. Since the same has to be done for bitonic merge, we do this in Section 6 for both merging algorithms together.

The following lemma gives an expensive input for odd-even merge.

Lemma 7 *Let $A_0 > B_m$. Then processor P_1 and processor P_{p-2} exchange their elements in all steps of odd-even merge.*

Proof. First note that all processors exchange all their elements in the first step. By induction it can be shown that before step i of the for-loop, $\log p - 1 \ge i \ge 1$, the first 2^{i+1} processors store elements of the following kind. The even indexed processors store a sorted sequence F^i and and the odd indexed processors store a sorted sequence G^i where the first element of F^i is larger than the last element of G^i. (Note that $F^{\log p-1} = B$ and $G^{\log p-1} = A$.) A similar claim holds for the last 2^{i+1} processors. Thus processor P_1 and processor P_{p-2} have to exchange all their elements in all steps of the for-loop. □

5 Bitonic merge

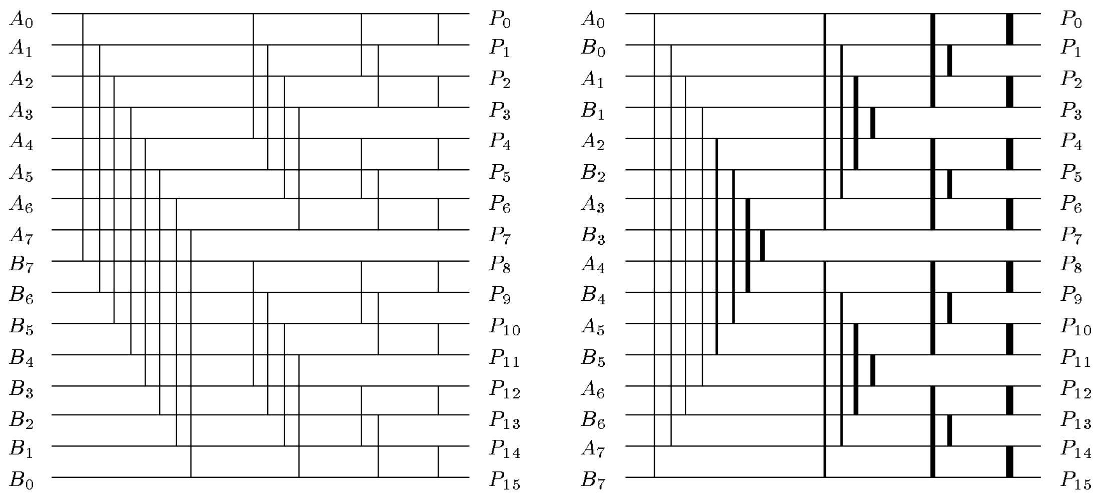
The bitonic merge network uses the following recursion.
Let $A = A_0, A_1, A_2, ..., A_{m-1}$ and $B = B_0, B_1, B_2, ..., B_{m-1}$ be the two sequences to be merged and denote the outcome of the merge by $E_0, ..., E_{2m-1}$. If $m = 1$, compare and exchange, if necessary, A_0 and B_0. Else, merge $A_{\text{even}} = A_0, A_2, A_4, ...$ with $B_{\text{odd}} = B_1, B_3, ...$ into

$C = C_0, C_1, C_2, ...$ and $A_{\text{odd}} = A_1, A_3, ...$ with $B_{\text{even}} = B_0, B_2, B_4, ...$ 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, $i \geq 0$.

To obtain a network from this recursion we have to decide how to store the input. Figure 2.a shows the network given in [1] where B is stored in reversed order behind A, and Figure 2b shows the network we use. It is obtained by storing A and B alternating (this network is also known as the balanced merging network [6]). From it we can again derive a parallel merging algorithm by substituting all comparisons by the Compare-Exchange procedure of Section 4.

procedure Bitonic_Merge(p);
for $i = \log p$ **downto** 1 **do**
  { mask $= 2^i - 1$;
   for all j, $j \& 2^{i-1} = 0$, **pardo**
     compare–exchange(P_j, $P_{j\hat{}maske}$);
  }

Here, & denotes bitwise AND and ˆ denotes bitwise XOR. That is, in step i each processor P_j communicates with the processor whose index is obtained by flipping the rightmost i bits of the binary representation of j. (Similar to odd-even merge, we denote the step of bitonic merge where the index i has value k by step k.)



a) original bitonic merge   b) modified bitonic merge

Figure 2: Bitonic merge with 16 processors.

Note that the distances between communicating processors in a given step of Bitonic_Merge differ much: the distances ly between 1 and $2^i - 1$ in step i. Since we want to establish a relationship between d_{max} (see Section 4) and the running time of Bitonic_Merge, we divide the communicating pairs of processors into $\log p$ batches. The membership to a batch is determined by the difference between the indexes of the subsequences of A and B the two processors hold at the beginning of the algorithm. Batch i covers all (odd) differences δ where $2^{i-2} \leq \delta < 2^{i-1}$ if $i \geq 2$ and where $\delta = 0$ if $i = 1$. Thus, Batch 1 contains all communicating pairs of step 1, and Batch i, $i \geq 2$, contains all communicating pairs (P_j, P_k) of steps $\log p$ through 2 where $2^{i-1} < k - j < 2^i$ $(0 < k - j < 4$ if $i = 2)$. Note that communicating pairs of Batch i, $i \geq 2$, occur in steps $\log p$ through i. Figure 2 shows the balanced merger of size 16 where membership to a batch is indicated by the thickness of the vertical line connecting the two horizontal lines corresponding to the communicating processors. We want to establish a relationship between the maximal i where a pair of processors belonging to Batch i exchange elements and d_{max} and the running time of Bitonic_Merge. The following lemmas will do this.

Lemma 8 *Each processor P_j can belong to at most one communicating pair of Batch i if $i \geq 3$ and to at most two communicating pairs of Batch 2.*

Proof. First assume that $i \geq 3$. Let P_j belong to Batch i in step l of the for-loop. In step l P_j communicates with P_k where k can be produced from j by flipping the rightmost l bits of j. In other words, $j - k = j \bmod 2^l - k \bmod 2^l = 2(j \bmod 2^l) - 2^l + 1$. Since j and k belong to Batch i, we know that $2^{i-1} < |j-k| < 2^i$ and thus $2^{i-1} < |2(j \bmod 2^l) - 2^l + 1| < 2^i$. From this we can derive that either $2^{i-1} + 2^l - 1 < 2(j \bmod 2^l) < 2^i + 2^l - 1$ or else $2^l - 2^i - 1 < 2(j \bmod 2^l) < 2^l - 2^{i-1} - 1$. Thus either $2^{l-1} + 2^{i-2} \leq j \bmod 2^l \leq 2^{l-1} + 2^{i-1} - 1$ or else $2^{l-1} - 2^{i-1} \leq j \bmod 2^l \leq 2^{l-1} - 2^{i-2} - 1$.

Let v be maximal where P_j belongs to Batch i in step v and let $i \leq q < v$. Since one of the above inequalities holds for v, 2^q divides 2^{v-1}, and $q \geq i$, either

$$2^{i-2} \leq j \bmod 2^q \leq 2^{i-1} - 1$$

or else

$$2^q - 2^{i-1} \leq j \bmod 2^q \leq 2^q - 2^{i-2} - 1.$$

Now assume that P_j belongs to Batch i in step q. Then either

$$2^{q-1} + 2^{i-2} \leq j \bmod 2^q \leq 2^{q-1} + 2^{i-1} - 1$$

or else

$$2^{q-1} - 2^{i-1} \leq j \bmod 2^q \leq 2^{q-1} - 2^{i-2} - 1.$$

Thus, we have two sets of inequalities where one of each set has to hold. However, then either

$$2^{i-1} + 2^{i-2} \leq 2^{q-1} + 2^{i-2} \leq j \bmod 2^q \leq 2^{i-1} - 1,$$

or else

$$2^{i-2} \leq j \bmod 2^q \leq 2^{q-1} - 2^{i-2} - 1 \text{ and } 2^{q-1} - 2^{i-1} \leq j \bmod 2^q \leq 2^{i-1} - 1$$

(i.e., $q \geq i + 1$ and $q \leq i$), or else

$$2^q - 2^{i-1} \leq j \bmod 2^q \leq 2^{q-1} + 2^{i-1} - 1 \text{ and } 2^{q-1} + 2^{i-2} \leq j \bmod 2^q \leq 2^q - 2^{i-2} - 1$$

(i.e., $q \leq i$ and $q \geq i + 1$), or else

$$2^{q-1} \leq 2^q - 2^{i-1} \leq j \bmod 2^q \leq 2^{q-1} - 2^{i-2} - 1,$$

neither of which is possible.

Next assume that $i = 2$. Let P_j belong to Batch 2 in step l. Now we can conclude that either $2^{l-1} \leq j \bmod 2^l \leq 2^{l-1} + 1$ or else $2^{l-1} - 2 \leq j \bmod 2^l \leq 2^{l-1} - 1$.

Let v be maximal where P_j belongs to Batch 2 in step v, and let $2 \leq q < v$. Since 2^q divides 2^{v-1} and the above condition holds for v, either $0 \leq j \bmod 2^q \leq 1$, or else $2^q - 2 \leq j \bmod 2^q \leq 2^q - 1$.

Again assume that P_j belongs to Batch i in step q. Then $2^{q-1} - 2 \leq j \bmod 2^q \leq 2^{q-1} + 1$. This can only be the case if $q = 2$ and thus there exist at most two steps where P_j belongs to Batch 2. □

To derive an upper bound for the running time of bitonic merge Lemma 8 does not suffice: it might happen that a processor has to wait for other processors that belong to batches with smaller numbers. The following lemma shows that no processor has to wait for a long time.

Lemma 9 *Let $P_{j_{\log p+1}}, P_{j_{\log p}}, ..., P_{j_{i+1}}, P_{j_i}$ be a path an element x can take during steps $\log p$ through i of the algorithms, $i \geq 2$ (that means that x is stored at processor P_{j_k} after the execution of step k).*

Then there exists at most one index t, $\log p \geq t > i$, where P_{j_t} and $P_{j_{t+1}}$ belong to Batch i or smaller in step t.

Proof. Let processor P_k belong to Batch i or smaller in step q, $q \geq i$. Then $|k - l| < 2^i$ where l is produced from k by flipping the rightmost q Bits. In other words, $k \bmod 2^q + l \bmod 2^q = 2^q - 1$, and $l = k + 2^q - 1 - 2(k \bmod 2^q)$. Thus $|k - l| = |k \bmod 2^q - l \bmod 2^q| = |2(k \bmod 2^q) - 2^q + 1| < 2^i$, or $2^q - 2^i - 1 < 2(k \bmod 2^q) < 2^q + 2^i - 1$. It follows that $2^{q-1} - 2^{i-1} \leq k \bmod 2^q \leq 2^{q-1} + 2^{i-1} - 1$. That means that $k = k_1 2^q + 2^{q-1} + k_2$ or $k = k_1 2^q + 2^{q-1} - 2^{i-1} + k_2$, where $0 \leq k_2 < 2^{i-1}$.

Assume that $v = j_t$, $\log p \geq t > i$, and P_v belongs to Batch i or smaller in step t. Let t be chosen minimal among all indexes with this property. We know that $v = v_1 2^t + 2^{t-1} + v_2$ or $v = v_1 2^t + 2^{t-1} - 2^{i-1} + v_2$ where $0 \leq v_2 < 2^{i-1}$ (see Figure 2).

Assume further that $u = j_s$, $s > t$, and P_u belongs to Batch i or smaller in step s. Since P_v can be reached from P_u in steps s through t (in all of these steps, either the rightmost t bits are flipped or all of them stay the same), u is of the form $u = u_1 2^t + 2^{t-1} + u_2$ or $u = u_1 2^t + 2^{t-1} - 2^{i-1} + u_2$, where $0 \leq u_2 < 2^{i-1}$. From this follows that $s = t$, which is a contradiction. □

Definition 4 *Let i_{max} be the maximal i where a communicating pair of Batch i exchanges an element and let e_{max} be the maximal number of elements exchanged in Batch i_{max}.*
Let $t_{bit}(i, j)$, $1 \leq i \leq \log p + 1, 0 \leq j \leq p - 1$ and $R_{bit}(r, p)$, be defined as described in Section 3 where alg = bit stands for bitonic merge.

Lemma 10 *$d_{max} \geq 2^{i_{max}-2} r + 2e_{max}$ if $i_{max} \geq 3$ ($d_{max} \geq 1$ if $i_{max} = 2$).*

Proof. For $i_{max} = 2$ the claim is obvious. Thus assume that $i_{max} > 2$.
Let P_g and P_h be two processors that exchange elements in step $v > i_{max}$ and that belong to Batch b_{max} in step v. We distinguish two cases.
Case 1. Processors P_g and P_h hold the same elements they held at the beginning of the for-loop. Then the two processors hold two sequences A_j and B_k where $|j - k| \geq 2^{i_{max}-2} + 1$. Thus

Since P_v belongs to Batch i or smaller in step t, v has one of the following forms (the first line shows the bit positions):

		s	s-1	s-2		t	t-1				i-2		0
x	...					x	1	0	...	0	x	...	x
x	...					x	0	1	...	1	x	...	x

Since P_u belongs to Batch i or smaller in step s, u has one of the following forms:

x	...	x	1	0	...		0		...	0	x	...	x
x	...	x	0	1	...		1		...	1	x	...	x

Since P_v can be reached from P_u in steps s through t, u has one of the following forms:

x	...					x	1	0	...	0	x	...	x
x	...					x	0	1	...	1	x	...	x

If $s > t$, this is a contradiction.

Figure 3: Binary representation of v and u (an x stands for either 1 or 0)

there exists a rank difference of at least $(2^{i_{max}-2}+e_{max})r-(r-e_{max}) = 2^{i_{max}-2}r+2e_{max}-1$.
Case 2. Processors P_g and P_h hold elements they did not hold at the beginning of the for-loop. Then at least one of them exchanged an element in step w where $w > v$. Let P_h be this processor. According to Lemmas 8 and 9, P_h belongs to Batch $i_{max}-1$ or smaller in step w. Let P_j be the processor P_h is connected to in step w. Then, before the execution of step w, P_h and P_j both hold the elements they held at the beginning of the for-loop (this follows from Lemma 9). Assume that $h\&2^{v-1}=0$, that is, h is the smaller of the two communicating processors in step v (the other case is symmetric). Assume further that P_h belongs to Batch l in step w, $l < i_{max}$. Then $h = h_1 2^w + 2^{w-1} + h_2$ or $h = h_1 2^w + 2^{w-1} - 2^{l-1} + h_2$ where $0 \le h_2 < 2^{l-1}$ (see proof of Lemma 9). Since $h\&2^{v-1}=0$ and $l < v < w$, $h = h_1 2^w + 2^{w-1} + h_2$ where $0 \le h_2 < 2^l$, that is, P_h is the larger of the two processors in step w. Accordingly, P_j is the smaller of the two processors and after the execution of step w P_h can only store elements that have an even larger rank difference. □

Lemma 11 $R_{bit} \le (i_{max}+1)r$.

Proof. It is clear that the rightmost i_{max} steps can be executed in time $i_{max}r$. According to Lemma 9, each processor has to wait for at most time r before it can begin to execute step i_{max} and thus the claim follows. □

Comment. The above bound on R_{bit} cannot be improved easily: if a processor has to wait before it can start with the execution of step i_{max}, this might be because of a rank difference some distance away and we cannot argue for a larger d_{max} than in Lemma 11.

Now we can prove the main results of this section.

Theorem 3
1. *If* $d_{max} \leq r$, $R_{bit}(r,p) \leq d_{max} + \min\{r, r/2 + d_{max}\}$.
2. $R_{bit}(r,p) \leq (3 + \max\{0, \lfloor \log((d_{max}-1)/r) \rfloor\})r$.

Proof. This follows from Lemma 10 and Lemma 11.

Theorem 4

$$\mathrm{Exp}\left(R_{bit}(r,p)\right) \leq r\left(3 + \sum_{i=1}^{\log p - 3} P\left(2^i r\right)\right).$$

Proof. This follows from Lemma 10, Lemma 11, and the definition of $P(\Delta)$ from Section 4.

6 Asymptotic Bounds

In this section we derive closed formulations for the bounds on the average running times of the merging and sorting algorithms. We will use the following two lemmas.

Lemma 12 *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*

$$P(\Delta) = 2\left(\binom{2m}{m+\Delta} - \sum_{k\geq 1}\left(\binom{2m}{m+2k\Delta} - \binom{2m}{m+(2k+1)\Delta}\right)\right) \Bigg/ \binom{2m}{m},$$

$$\frac{\sqrt{\pi}}{2}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\left(\frac{\Delta^2}{m}+(2\ln 2-1)\frac{\Delta^4}{m^3}\right)} \leq P(\Delta) \leq \frac{4}{\sqrt{\pi}}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\frac{\Delta^2}{m}},$$

and

$$\frac{\sqrt{\pi}}{2}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\left(\frac{\Delta^2}{m}+(2\ln 2-1)\frac{\Delta^4}{m^3}\right)} \leq Q_A(\Delta).$$

Proof.
The first claim follows from the repeated reflection principle (see [8]), and the second claim follows from the first one and the following lemma.

Lemma 13

$$\frac{\sqrt{\pi}}{2}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\left(\frac{\Delta^2}{m}+(2\ln 2-1)\frac{\Delta^4}{m^3}\right)} \leq \frac{\binom{2m}{m+\Delta}}{\binom{2m}{m}} \leq \frac{2}{\sqrt{\pi}}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\frac{\Delta^2}{m}}.$$

The proof for this lemma can be found in the appendix.

Theorem 5

$$\frac{2m}{p}\left(0.84\max\left\{0, \left(\left\lfloor \log\left(\max\left\{1, 0.339\sqrt{p^2/m} - 7\right\}\right)\right\rfloor - 2\right)\right\} + 0.886e^{-\left(\frac{4m}{p^2}+(2\ln 2-1)\frac{16m}{p^4}\right)}\right)$$

$$\leq \mathrm{Exp}\left(R_{odd}(r,p)\right) \leq$$

$$\frac{2m}{p}\left(1.17 + \left\lceil \log\left(1 + 0.84\sqrt{p^2/m}\right)\right\rceil\right)$$

Theorem 6

$$\mathrm{Exp}\left(R_{bit}(r,p)\right) \le \frac{2m}{p}\left(3.17 + \max\left\{0, \left\lceil \log\left(0.42\sqrt{p^2/m}\right)\right\rceil\right\}\right)$$

Proof of Theorem 5 and Theorem 6.

First we proof the upper bounds. In Theorem 2 and Theorem 4 we gave upper bounds for $R_{odd}(r,p)$ and $R_{bit}(r,p)$ that contain summands of the form $P(\Delta_i)$ where $\Delta_i = (2^{i-1}-1)r$, $1 \le i \le \log p - 1$, for R_{odd} and $\Delta_i = 2^i r$, $1 \le i \le \log p - 3$, for R_{bit}. By Lemma 12 we know that $P(\Delta_i) \le 8/\sqrt{3\pi}e^{-\Delta_i^2/m}$ (here we made use of the fact that $\Delta_i \le m/2$). Note that $\Delta_{i+1}/\Delta_i \ge 2$ (for odd-even merge this is true if $i \ge 2$). Let i_α be the smallest i such that $e^{-\Delta_i^2/m} \le \alpha$, $\alpha < 1$. Then $e^{-\Delta_{i_\alpha+j}^2/m} \le \alpha^{(2^j)^2}$ where $j \ge 1$. In the case of bitonic merge we choose $\alpha = 0.5$ and bound the first i_α terms of the sum by 1, and in the case of odd-even merge we choose $\alpha = 0.5^4 = 0.0625$ and bound the first $i_\alpha - 1$ terms of the sum by 1. This leads to the claimed upper bounds.

This leaves the lower bound for odd-even merge. In Theorem 2 we gave a lower bound for R_{odd} that contains summands of the from $Q_A(\Delta_i)$ where $\Delta_i = (2^{i+1}+7)r/3$, $2 \le i \le \lfloor (3/2)p - 7\rfloor - 1$. Since $\Delta_i \ge 0$ we know by Lemma 12 that

$$Q_A(\Delta_i) \ge \sqrt{\pi}/2e^{-\Delta_i^2/m+(2\ln 2-1)\Delta_i^4/m^3} \ge \sqrt{\pi}/2e^{-\Delta_i^2/m+(2\ln 2-1)(\Delta_i^2/m)^2}.$$

Let i_β be the largest i such that $e^{-\Delta_i^2/m} \ge \beta$, $\beta > 0$. Then

$$\begin{aligned} R_{\mathrm{odd}}(r,p) &\ge 2m/p(Q_A(2r-1) + \sqrt{\pi}\beta/2 \cdot i_\beta - 1) \\ &= 2m/p(Q_A(2r-1) + \sqrt{\pi}\beta/2(\lfloor \log(3\sqrt{-\ln\beta m}/r - 7)\rfloor - 2)). \end{aligned}$$

By choosing $\beta = 0.95$ and substituting the formula from Lemma 12 we arrive at the claimed bound. □

Definition 5 *Let $R_{oddSort}(r,p)$ for odd-even merge sort and $R_{bitSort}(r,p)$ for bitonic merge sort be defined as described in Section 3.*

Theorem 7

$$\mathrm{Exp}\left(R_{oddSort}(r,p)\right) \le 2\log p - 1 + 0.195$$

if $n \ge 2p^2 \ln 2$, and

$$\mathrm{Exp}\left(R_{oddSort}(r,p)\right) \le 2\log p + 0.195 + \left\lceil 0.5\log\left(\frac{1.39p^2}{n}\right)\right\rceil\left(1 + \left\lceil 0.5\log\left(\frac{2.78p^2}{n}\right)\right\rceil\right)$$

if $n < 2p^2 \ln 2$.

$$\mathrm{Exp}\left(R_{oddSort}(r,p)\right) \le 3\log p - 3 + 0.195$$

if $n \ge 0.5p^2 \ln 2$, and

$$\mathrm{Exp}\left(R_{oddSort}(r,p)\right) \le 3\log p - 2 + 0.195 + \left\lceil 0.5\log\left(\frac{0.35p^2}{n}\right)\right\rceil\left(1 + \left\lceil 0.5\log\left(\frac{0.7p^2}{n}\right)\right\rceil\right)$$

if $n > 0.5p^2 \ln 2$.

Proof. Both algorithms work by dividing the input into two sets, sorting the two sets recursively and then merging the resulting lists. Before the final merging step can start, both recursive calls must be finished. That is, we have to wait for the maximum running time of two independent sorts.

Let R_{merge} stand for either R_{odd} or R_{bit} and let R_{sort} stand for either $R_{oddSort}$ or $R_{bitSort}$. Above we have shown that $\mathrm{Exp}(R_{merge}(r,p)/r)$ can be upper bounded by a function $f(r,p)$ and that the probability that $R_{merge}(r,p)/r$ is at least $f(r,p)+j-1+1/r$ is bounded by $(8/\sqrt{3\pi})2^{-(2^j)^2}$, $j \geq 1$. We treat the contribution of $f(r,p)$ and the running times exceeding $f(r,p)$ separately. The contribution of the second part can be represented as follows.

Given is a complete binary tree with $p/2$ leaves where each node stands for a call of the merging procedure. The nodes are labeled with numbers in $\{0,1,...,\log p\}$ where a label of 0 denotes a running time of at most $f(r,p')$ (p' the appropriate number of processors) and where a label of j denotes a running time between $f(r,p')+j-1+1/r$ and $f(r,p')+j$. The running time of the sorting algorithm corresponds to the "heaviest" path from the root to a leaf in this tree, where the weight of a path is the sum of the labels of its nodes.

Let us examine the labels of the nodes and their probabilities more closely. (For this, we are going to use the notation from the proof of Theorem 5 and Theorem 6.) Firstly, for odd-even merge sort (bitonic merge sort) the labels of nodes corresponding to 2 or 4 (2 to 8, resp.) processors are always 0. Secondly, the probability that a node v has a certain label depends on the relationship between r and the number of processors used in the corresponding merge. Suppose that the number of processors p_α is such that $i_\alpha - 1 \leq 1$ in the case of odd-even merge ($i_\alpha \leq 1$ in the case of bitonic merge sort), and suppose that p_α is maximal with this property. Let $\beta = 0.5$. Then the probability that v has label j, $j \geq 1$, is bounded by $(8/\sqrt{3\pi})\beta^{(2^j)^2}$. This is also true for all nodes on the same level as or on a higher level than v. On the other hand, consider a node w that corresponds to a merge with $p' = p_\alpha/2^k$ processors. Then the probability that w is labeled with j, $j \geq 1$, is bounded by $(8/\sqrt{3\pi})\beta^{(2^j)^2 2^k}$. Consider the subtree with root v and suppose that v_1 and v_2 are the children of v. We will show that it is possible to replace v_1 (v_2, resp.) by one node with the same probabilities as the nodes on higher levels. First we unite all nodes with depth k, $k \geq 1$, in the subtree rooted at v_1 into one node, that means we replace the subtree rooted at v_1 by a path. (In our notation the depth of the root node is 1). For the new node a depth k the probability that its label is j is bounded by $(8/\sqrt{3\pi})\beta^{(2^j)^2 2^{k+1}} 2^k = (8/\sqrt{3\pi})\beta^{(2^j)^2 2^{k+1}-k}$. Next we bound the probability that the weight of the path starting at v_1 is j, $j \geq 1$. By induction it can be shown that this probability is bounded by $(8/\sqrt{3\pi})\beta^{(2^j)^2}$. Now we are given a complete binary tree of depth $\log p - \log p_\alpha + 2$ where for any node the probability that its label is j, $j \geq 1$, is $(8/\sqrt{3\pi})0.5^{(2^j)^2}$.

Let $T(d,i)$ be the probability that the weight of the root of the tree with depth d is at least i. We can bound $\mathrm{Exp}(R_{sort}(r,p)/r)$ by

$$\sum_{i=1}^{\log p} f(r,2^i) + \sum_{i=1}^{\log p(\log p+1)/2} T\left(\min\left(\log p - c, \log(p/p_\alpha)+2, i\right)\right),$$

where $c = 2$ and $p_\alpha = 2^{\lfloor \log(r/(2ln2)) \rfloor}$ for odd-even merge sort, and $c = 3$ and $p_\alpha = 2^{\lfloor \log(2r/ln2) \rfloor}$ for bitonic merge sort. Thus it suffices to upper bound $T(d,i)$. Instead of doing this directly we will lower bound $S(d,i) = 1 - T(d,i+1)$, i.e., $S(d,i)$ is the probability that the weight of the root is at most i. It is easy to see that

$$S(d,i)=\sum_{j=0}^{\min(i,\log p)} y_j(S(d-1,i-j))^2$$

where y_j is the probability that a node is labeled with j, $j\geq 0$.

For ease of calculation we assume that labels range from 0 to ∞ and replace y_1 by $q_1=(8/\sqrt{3\pi})2^{-4}$, y_j by $q_j=q_1/2^{12(j-1)}$ for $j\geq 2$, and y_0 by $1-\sum_{j=1}^{\infty}q_j$. Let $\delta=2^{-12}$, $\gamma=q_1/(1-\delta)\approx 0.163$. We will show that $S(d,d-1+i)\geq 1-\gamma^{i+1}$ for $i\geq 0$. This is obvious for $d=1$. For larger p's we have

$$
\begin{aligned}
S(d,d-1+i) &= \sum_{j=0}^{d-1+i} q_j(S(d-1,d-1+i-j))^2 \geq \sum_{j=0}^{i+1} q_j(1-\gamma^{i+2-j})^2\\
&= \left(1-\frac{q_1}{1-\delta}\right)\left(1-\gamma^{i+2}\right)^2+\sum_{j=1}^{i+1} q_1\delta^{j-1}\left(1-\gamma^{i+2-j}\right)^2\\
&= \left(1-\frac{q_1}{1-\delta}\right)\left(1-\gamma^{i+2}\right)^2+q_1\sum_{j=1}^{i+1}\delta^{j-1}\left(1-2\gamma^{i+2-j}+\gamma^{2(i+2-j)}\right)\\
&= \left(1-\frac{q_1}{1-\delta}\right)\left(1-\gamma^{i+2}\right)^2+q_1\frac{1-\delta^{i+1}}{1-\delta}-2q_1\gamma^{i+1}\frac{1-\left(\frac{\delta}{\gamma}\right)^{i+1}}{1-\frac{\delta}{\gamma}}+q_1\gamma^{2(i+1)}\frac{1-\left(\frac{\delta}{\gamma^2}\right)^{i+1}}{1-\frac{\delta}{\gamma^2}}\\
&= 1-2\gamma^{i+2}+\gamma^{2(i+2)}+\frac{q_1}{1-\delta}\left(2\gamma^{i+2}-\gamma^{2(i+2)}-\delta^{i+1}\right)-2q_1\frac{\gamma^{i+1}-\delta^{i+1}}{1-\frac{\delta}{\gamma}}+q_1\frac{\gamma^{2(i+1)}-\delta^{i+1}}{1-\frac{\delta}{\gamma^2}}\\
&\geq 1-\gamma^{i+1}+\gamma^{i+1}\left(1-2\gamma+\gamma^{i+3}\right)+\frac{q_1}{1-\delta}\left(\gamma^{i+1}\left(2\gamma-\gamma^3\right)-\delta^{i+1}\right)-\frac{2q_1}{1-\frac{\delta}{\gamma}}\left(\gamma^{i+1}-\delta^{i+1}\right)\\
&\geq 1-\gamma^{i+1}+\gamma^{i+1}\left(1-2\gamma+\gamma^{i+3}-\frac{2q_1}{1-\frac{\delta}{\gamma}}\right)\\
&\geq 1-\gamma^{i+1}.
\end{aligned}
$$

The last inequality follows because $\gamma\leq 1/2-q_1-\delta$.
Thus we have shown that $T(d,d-1+i)\leq\gamma^i$ for $i\geq 1$ and we get

$$\sum_{i=1}^{\log p(\log p+1)/2} T(d,i)\leq d-2+\frac{1}{1-\gamma}.$$

This leaves the contribution of $f(r,2^i)$. For odd-even merge sort we have

$$\sum_{i=1}^{\log p} f(r,2^i)=\sum_{i=1}^{\log p}\left(1+\left\lceil\log\left(1+\sqrt{\ln 2}\sqrt{2^{i+1}/r}\right)\right\rceil\right),$$

and for bitonic merge sort we have

$$\sum_{i=1}^{\log p} f(r,2^i)=\sum_{i=1}^{\log p}\left(3+\max\left\{0,\left\lceil\log\left(0.5\sqrt{\ln 2}\sqrt{2^{i+1}/r}\right)\right\rceil\right\}\right).$$

Let $y = \log(r/(2\ln 2)) - 1$ ($y = \log(2r/\ln 2) - 1$) in the case of odd-even merge sort (bitonic merge sort, resp.). Note that $\log p_\alpha = \lfloor y \rfloor$. As long as $i \leq y$, each term of the first (second, resp.) sum will be bounded by 2 (3, resp.). Assume that $\log p_\alpha < \log p$. Then we get

$$\sum_{i=\log p_\alpha+1}^{\log p} \left\lceil \log\left(1+\sqrt{\ln 2}\sqrt{2^{i+1}/r}\right)\right\rceil \leq \sum_{i=1}^{\log p - \log p_\alpha} \left\lceil \log\left(1+\sqrt{2^i}\right)\right\rceil$$

$$\leq 1+\log p-\log p_\alpha + \sum_{i=1}^{\lceil(\log p-\log p_\alpha)/2\rceil} 2i \leq 1+\log p-\log p_\alpha + \left\lceil \frac{1}{2}\log\left(\frac{1.39p^2}{n}\right)\right\rceil \left\lceil \frac{1}{2}\log\left(\frac{2.78p^2}{n}\right)\right\rceil$$

and

$$\sum_{i=\log p_\alpha+1}^{\log p} \left\lceil \log\left(0.5\sqrt{\ln 2}\sqrt{2^{i+1}/r}\right)\right\rceil \leq \sum_{i=1}^{\log p - \log p_\alpha} \left\lceil \log\left(\sqrt{2^i}\right)\right\rceil$$

$$= \sum_{i=1}^{\log p-\log p_\alpha} \left\lceil \frac{1}{2} i\right\rceil \leq \sum_{i=1}^{\lceil(\log p-\log p_\alpha)/2\rceil} 2i \leq \left\lceil \frac{1}{2}\log\left(\frac{0.35p^2}{n}\right)\right\rceil \left\lceil \frac{1}{2}\log\left(\frac{0.7p^2}{n}\right)\right\rceil.$$

Putting everything together we arrive at upper bounds for odd-even merge sort of $2\log p - 1 + 0.195$ if $n \geq 2p\ln 2$ and $2\log p + 0.195 + \lceil 0.5\log(1.39p^2/n)\rceil(1+\lceil 0.5\log(2.78p^2/n)\rceil)$ if $n < 2p\ln 2$. The upper bounds for bitonic merge sort are $3\log p - 3 + 0.195$ if $n \geq 0.5p\ln 2$ and $3\log p - 2 + 0.195 + \lceil 0.5\log(0.35p^2/n)\rceil(1+\lceil 0.5\log(0.7p^2/n)\rceil)$ if $n < 0.5p\ln 2$. Here we have taken into account that for 2 and 4 processors only 1 and 2, resp., steps are executed. □

Comment. The upper bound for bitonic merge sort is in most cases much larger than the upper bound for odd-even merge sort. The reason for this is that in each step where two processors exchange elements we assume that they exchange all their elements as explained in Section 3. However, if we use the number of elements actually exchanged instead, the 3 in the above bounds can be replaced by 2 for large n.

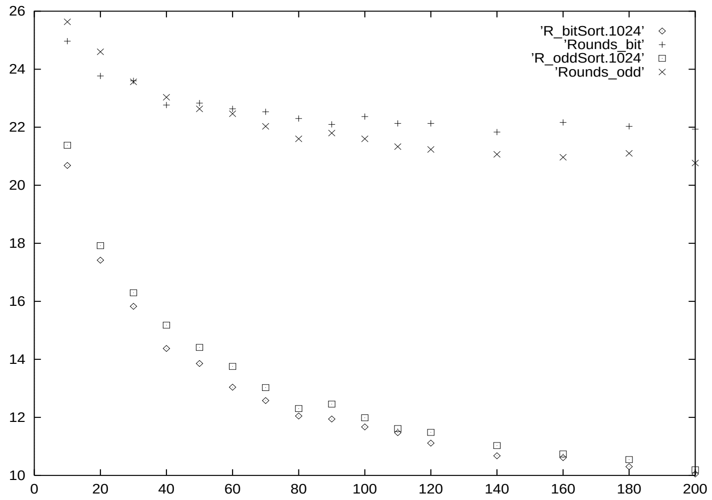


Figure 4: Exchanged elements while sorting with 1024 processors

7 Experimental results

In the above sections we proved bounds on the average running times of odd-even merge (sort) and bitonic merge (sort). These bounds are asymptotically similar and do not allow an actual comparison of odd-even merge and bitonic merge. In this section we present some simulation results for the algorithms. In all cases, odd-even merge (sort) and bitonic merge (sort) were run on the same, randomly generated inputs. We used the mean of 30 runs.

Figure 4 shows $R_{bitSort}/r$ and $R_{oddSort}/r$ for 1024 processor. The number of elements per processor ranges from 10 to 200. One can see that the average number of exchanges per element drops rapidly towards a value of slightly larger than $10 = \log 1024$. Bitonic merge sort performs always a bit better than odd-even merge sort. Figure 4 also shows the average maximum number of rounds: instead of counting the exchanged elements as in R_{alg}, we count a 1 if at least one element is exchanged and a 0 else. This is essentially what we did when deriving the upper bounds. As one can see, the values here are much larger and we cannot expect to derive exact bounds for $R_{oddSort}$ and $R_{bitSort}$ with this method. If we count the number of rounds, odd-even merge sort performs in most cases better than bitonic merge sort. Thus here the relationship between the running times is the same as that of the bounds derived in Section 6.

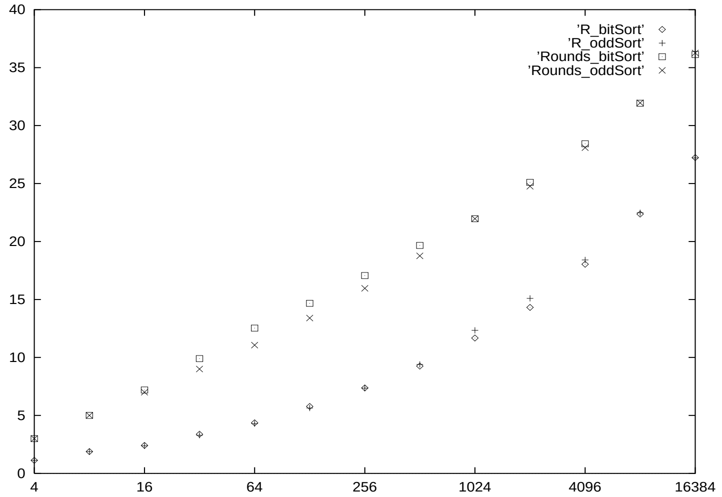


Figure 5: Exchanged elements while sorting 100 elements per processor

Figure 5 shows $R_{bitSort}/100$ and $R_{oddSort}/100$ for 100 elements per processor and various number of processor. The average number of exchanges per element grows very slowly and is for 16384 processors still smaller than $2 \log p$. In most cases, the two values are almost identical. If we consider the average maximum number of rounds, odd-even merge sort performs better in many cases than bitonic merge sort. For large numbers of processors, the bound through the average maximum number of rounds becomes better.

Figure 6 shows $R_{bit}/100$ and $R_{odd}/100$ for merging two lists with 100 elements per processor. At the beginning bitonic merge outperforms odd-even merge, but at the end this changes. Thus it is likely that for very large number of processors odd-even merge sort is faster than bitonic merge sort.

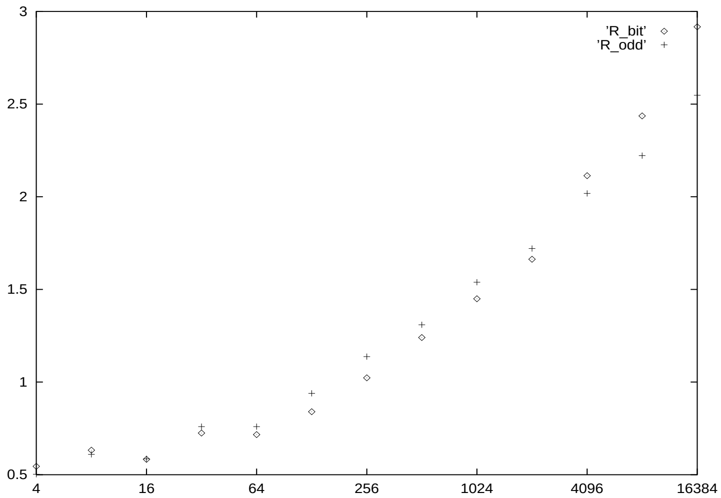


Figure 6: Exchanged elements while merging 100 elements per processor

8 Conclusions

In this paper we have derived new upper bounds for the average running time of special variants of odd-even merge sort and bitonic merge sort. We have shown that for large sizes of input each element will be exchanged at most twice (three times, resp.) with high probability. For several reasons, the derived bounds are not tight. Firstly, we wanted to derive closed formulations. By evaluating the formulas used in Sections 4 to 6 numerically, better bounds are possible. Secondly, we assumed that each time one element has to be exchanged all elements are exchanged (in [4] this was done and called guarded split and merge). If only the minimal number of elements is exchanged, for large sizes of input each element will be sent not much more often than $\log p$ times.

Ways to reduce the amount of communication in the worst case haven been examined in [7] and [10]. There it is suggested to rearrange the input elements regularly to be able to perform more work locally. In [7] ([10], resp.) it is shown how to do this with a total communication volume of $2n \log p$ ($n \log p$, resp.) if $n \geq p^2$ ($n \geq 2^{\log p^2/2}$, resp.). However, if $n \geq p^2$, our version of the bitonic merge sort algorithm will have a communication volume of about $n \log p$ with high probability. Since the local computation in the algorithms from [7] and [10] is more complicated, it is doubtful that these algorithms will outperform the order-preserving bitonic merge sort algorithm for many inputs.

As was already mentioned in [13], for large sizes of inputs sample sort will perform better than either odd-even merge sort or order-preserving bitonic merge sort. However, the simple techniques used here should increase the size of the input up to which the merge sort algorithms are faster.

Appendix

Lemma 3

$$\frac{\sqrt{\pi}}{2}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\left(\frac{\Delta^2}{m}+(2\ln 2-1)\frac{\Delta^4}{m^3}\right)} \leq \frac{\binom{2m}{m+\Delta}}{\binom{2m}{m}} \leq \frac{2}{\sqrt{\pi}}\sqrt{\frac{m^2}{(m^2-\Delta^2)}}e^{-\frac{\Delta^2}{m}}.$$

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. [15], pp. 308 ff.)

Using this we get

$$\sqrt{\frac{\pi m^2}{4(m^2-\Delta^2)}}2^{2m(H_2(\frac{m+\Delta}{2m})-1)} \leq \binom{2m}{m+\Delta}\Big/\binom{2m}{m} \leq \sqrt{\frac{4m^2}{\pi(m^2-\Delta^2)}}2^{2m(H_2(\frac{m+\Delta}{2m})-1)}.$$

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

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

and thus

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

$$= -\log(e)\left(\frac{m+\Delta}{2m}\left(\ln(0.5) + 2\frac{\Delta}{2m} - \frac{2^2}{2}\left(\frac{\Delta}{2m}\right)^2 + \frac{2^3}{3}\left(\frac{\Delta}{2m}\right)^3 - \frac{2^4}{4}\left(\frac{\Delta}{2m}\right)^4 + ...\right) + \right.$$

$$\left.\frac{m-\Delta}{2m}\left(\ln(0.5) - 2\frac{\Delta}{2m} - \frac{2^2}{2}\left(\frac{\Delta}{2m}\right)^2 - \frac{2^3}{3}\left(\frac{\Delta}{2m}\right)^3 - \frac{2^4}{4}\left(\frac{\Delta}{2m}\right)^4 - ...\right)\right)$$

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

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

Since

$$\sum_{i=1}^{\infty}\frac{1}{2i-1} - \frac{1}{2i} = \ln 2,$$

we get

$$1 - \log e\left(0.5\frac{\Delta}{m}^2 + (\ln 2 - 0.5)\frac{\Delta}{m}^4\right) \leq H_2\left(\frac{m+\Delta}{2m}\right) \leq 1 - \log e\left(0.5\frac{\Delta}{m}^2\right),$$

and thus the claim follows.

References

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

[2] G. Bilardi. Merging and sorting networks with the topology of the omega network. *IEEE trans. on comp.*, C-38, 10:1396–1403, 1989.

[3] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, Hilton Head, South Carolina, July 21–24, 1991. SIGACT/SIGARCH.

[4] K. Brockmann and R. Wanka. Efficient oblivious parallel sorting on the MasPar MP-1. In *Proc. 30th Hawaii International Conference on System Sciences (HICSS)*. IEEE, January 1997.

[5] Ralf Diekmann, Joern Gehring, Reinhard Lueling, Burkhard Monien, Markus Nuebel, and Rolf Wanka. Sorting large data sets on a massively parallel system. In *Proc. 6th IEEE-SPDP*, pages 2–9, 1994.

[6] Martin Dowd, Yehoshua Perl, Larry Rudolph, and Michael Saks. The periodic balanced sorting network. *Journal of the ACM*, 36(4):738–757, October 1989.

[7] Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, and Richard P. Martin. Fast parallel sorting under LogP: experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, August 1996.

[8] W. Feller. *An Introduction to Probability Theory and Its Applications I.* John Wiley, New York, second edition, 1950.

[9] William L. Hightower, Jan F. Prins, and John H. Reif. Implementations of randomized sorting on large parallel machines. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 158–167, San Diego, California, June 29–July 1, 1992. SIGACT/SIGARCH.

[10] Mihai Florin Ionescu and Klaus E. Schauser. Optimizing parallel bitonic sort. In *IPPS 97*, pages 303–309, 1997.

[11] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, 1973.

[12] Ch. Rüb. On the average running time of odd-even merge sort. In *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *LNCS*, pages 491–502, Munich, Germany, 2–4 March 1995. Springer.

[13] Ch. Rüb. On the average running time of odd–even merge sort. *Journal of Algorithms*, 22(2):329–346, February 1997.

[14] A. Wachsmann and R. Wanka. Sorting on a massively parallel system using a library of basic primitives: Modeling and experimental results. Technical Report TR-RSFB-96-011, Universität-GH Paderborn, May 1996. also Proc. European Conference in Parallel Processing (Euro-Par); 1997; to appear.

[15] F. J. Mac Williams and N. J. A. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, Amsterdam, 2 edition, 1978.