Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines*

Gianfranco Bilardi[1]
Alexandru Nicolau[2]

TR 86-769

August 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

1 This work is supported in part by NSF grant DCR-8602256.
2 This work is supported in part by NSF grant DCR-8502884, and the Cornell NSF Supercomputing Center.
* A preliminary version of this paper has been presented at the *20th Annual Conference on Information Systems and Sciences*, March 1986.

ADAPTIVE BITONIC SORTING: An Optimal Parallel Algorithm for Shared Memory Machines

Gianfranco Bilardi[1]and Alexandru Nicolau[2]

Department of Computer Science

Cornell University

Ithaca, NY 14853

Abstract

We propose a parallel algorithm, called *adaptive bitonic sorting*, which runs on a PRAC, a shared-memory multiprocessor where fetch and store conflicts are disallowed. On a P processors PRAC, our algorithm achieves optimal performance $TP = O(N \log N)$, for any computation time T in the range $\Omega(\log^2 N) \leq T \leq O(N \log N)$. Adaptive bitonic sorting has also a small constant factor, since it performs less than $2N \log N$ comparisons, and only a handful of operations per comparison.

Keywords: sorting, parallel computation, shared-memory machines, bitonic sequence, time $\times$ processors optimality.

AMS subject classifications: nonnumerical algorithms (68Q20), analysis of algorithms (68Q25), modes of computation (68Q10).

Abbreviated title: Adaptive bitonic sorting.

[1]This work is supported in part by NSF grant DCI-8602256.

[2]This work is supported in part by NSF grant DCR-8502884, and the Cornell NSF Supercomputing Center. A preliminary version of this paper has been presented at the 20th Annual Conference on Information Systems and Sciences, March 1986.

1 Introduction

Bitonic sorting [Ba68] is an interesting parallel algorithm based on a merge-sort scheme and an ingenious merging technique known as bitonic merging. Originally proposed for a network of comparators, bitonic sorting has been considered for implementation on a variety of architectures like the shuffle-exchange [St71], the binary cube [Pe77], the mesh [TK77], [NS79], the cube-connected cycles [PV81] and its pleated version [BP84], and on array processors [St78]. Various properties of bitonic networks have been investigated (e.g., [Kn73], [HS82], [Pr83], [Bl85]).

On an input of N elements, the bitonic merger performs $\Theta(N \log N)$ operations (comparisons and exchanges), coming within a constant factor of a lower bound due to R.W.Floyd for merging networks of comparators ([Kn73], pp. 230). However, in other models of parallel computation merging can be done with $O(N)$ operations [BH85], and therefore the bitonic algorithm is not optimal in these models.

In this paper we present procedures for merging and sorting, which we propose to call ***adaptive bitonic algorithms.*** Like Batcher's, our algorithms are based on bitonic sequences, but unlike Batcher's, they perform a set of comparisons that is a function of the input values. As a result, our approach cannot be used in the context of a network of comparators, but will be shown to be more efficient than Batcher's, in terms of the number of operations performed, on a general purpose shared-memory machine. When necessary to avoid confusion, we shall refer to Batcher's algorithm as the non-adaptive or the network algorithm.

Adaptive bitonic merging [sorting] can be implemented on a PRAC of P processors in time $T = O(N/P)$ $[T = O((N \log N)/P)]$, for $1 \leq P \leq N/2^{\lfloor \log \log N \rfloor}$. The PRAC [LPV81] is a shared-memory multi-processor where simultaneous access of the same memory location is disallowed. The product TP is optimal for both merging and sorting.

The merging algorithm also achieves the $T = \Omega(\log N)$ lower bound established in [Sn85] for merging on a PRAC. To our knowledge, ours is the first algorithm that attains the minimum time $T = O(\log N)$ with an optimal number of processors $P = O(N/\log N)$, on the PRAC model. An algorithm recently proposed in [AS85] achieves an optimal TP product, but for $T = \Omega(\log^2 N)$.

As for sorting on the PRAC, the question of asymptotic complexity was settled by the network of [AKS83], which is logspace uniform [L85], and therefore yields a PRAC algorithm with $TP = O(N \log N)$ for $T = \Omega(\log N)$. Very recently, the same asymptotic performance has been achieved by an adaptive algorithm proposed in [C86], building on the previous work of [P78] and [Kr83]. Our algorithm also achieves an optimal rate of growth for the TP measure, for $T = \Omega(log^2 N)$, with a much smaller constant factor than the AKS network, and probably smaller than the algorithm in [C86] (a precise estimate of the latter is not yet available). Indeed, even for time $T = O(\log^2 N)$, adaptive bitonic sorting performs only about twice as many operations as the fastest sequential sorting algorithms.

Our algorithms can obviously be implemented, with the same performance, on other shared-memory models with less restrictive memory access mechanisms. For results on the complexity of merging and sorting on some of these models see, for example, [BH85], [HH81], [Kr83], [SV81], [V75].

In the bitonic merging network, both the number of comparisons and the number of exchanges are $O(N \log N)$. Adaptive bitonic merging achieves a reduction of both numbers to O(N), based on two properties established in the paper. First, there exists a subset of less than $2N$ of the comparisons performed by the network sufficient to determine the result of all the others. Second, there is a regularity in the pattern of exchanges that can be exploited by using a data structure, which we call *the bitonic tree*, whereby many element exchanges can be accomplished by a small number of subtree (i.e., pointers) exchanges. The properties of bitonic sequences exploited by our algorithm are discussed in Section 2. The adaptive bitonic-merging algorithm is developed in Section 3, where a sequential model is adopted for simplicity.

Parallel versions of merging and sorting are described in Section 4. Here the main difficulty consists in avoiding a loss in time performance with respect to the network algorithm. In fact, in the PRAC implementation, adaptive bitonic merging emulates the k^{th} stage of comparisons of the merging network in time $O(\log N - k)$, for $k = 0, 1, ..., \log N - 1$. If the stages are executed in sequence, the resulting time is $O(\log^2 N)$. However, a careful analysis of data-dependencies shows that $O(\log N)$ time can be achieved by a suitable overlapping of the stages.

The adaptive algorithms of Sections 3 and 4, as well as the network algorithms of [Ba68], assume that the length of the input sequence N is a power of two. The obvious strategy of padding the input sequence so that its length becomes a power of two leads to an increase of the complexity by a constant factor. In Section 5 we show how this increase can be avoided by developing a variant of the algorithm that works for arbitrary N.

Besides attaining an optimal rate of growth, the performance of the algorithms presented here exhibits also very small constant factors. The sorting algorithms perform less than $2N \log N$ comparisons, independently of the number of processors. The overhead incurred in distributing the algorithm among P processors is proportional to $P \log N$, contributing a lower order term to the total number of operations. Thus, adaptive bitonic sorting appears to be attractive for practical implementation. Some indication of the practical behaviour is given in Section 6.

2 Properties of Bitonic Sequences

Let $\underline{x} = (x_0, ..., x_{N-1})$ be a sequence of N (hereafter N is assumed even) elements from a totally ordered set. We introduce the following operators on $\underline{x}$:

$$S_j\underline{x} \triangleq (x_{j \bmod N}, x_{(j+1) \bmod N}, ..., x_{(j+N-1) \bmod N}), \tag{1}$$

$$L\underline{x} \triangleq (\min\{x_0, x_{N/2}\}, ..., \min\{x_{N/2-1}, x_{N-1}\}), \tag{2}$$

$$U\underline{x} \triangleq (\max\{x_0, x_{N/2}\}, ..., \max\{x_{N/2-1}, x_{N-1}\}). \tag{3}$$

A sequence $\underline{x}$ is *bitonic* if, for some j, the sequence $S_j\underline{x} = (y_0, ..., y_{N-1})$ consists of a non-decreasing portion followed by a non-increasing one. Bitonic merging is based on the fact that, if $\underline{x}$ is bitonic,

$$sort(\underline{x}) = (sort(L\underline{x}), sort(U\underline{x})). \tag{4}$$

This relation, due to [Ba68], leads to a recursive algorithm whose complexity is determined by that of computing $L\underline{x}$ and $U\underline{x}$. Theorem 1 below states four properties of L and U on bitonic operands. Property $P1$ is a lemma for the others. Properties $P2$ and $P3$, obtained by Batcher, imply Equation 4 above. Relation $P4$ is crucial here since it provides the basis for a method to compute $L\underline{x}$ and $U\underline{x}$ that is more efficient than the direct application of definitions 2 and 3 above, which are used in the bitonic algorithm in [Ba68].

Theorem 1. If $\underline{x}$ is a bitonic sequence (of even length) then the following properties hold:

P1. There is a shifted version $S_q\underline{x} = (z_0, ..., z_{N-1})$ of $\underline{x}$ such that

$$L\underline{x} = S_{(-q \bmod N/2)}(z_0, ..., z_{N/2-1}), \tag{5}$$

$$U\underline{x} = S_{(-q \bmod N/2)}(z_{N/2}, ..., z_{N-1}). \tag{6}$$

P2. Each element of $U\underline{x}$ is no smaller than each element of $L\underline{x}$.

P3. Sequences $L\underline{x}$ and $U\underline{x}$ are bitonic.

P4. Let q be as in $P1$ and let $t = q \bmod N/2$. Let $\underline{x} = (\underline{x}', \underline{x}'', \underline{x}''', \underline{x}'''')$ with $\underline{x}'$ and $\underline{x}'''$ of length t, and $\underline{x}''$ and $\underline{x}''''$ of length $N/2 - t$. If $q < N/2$ $(t = q)$, then

$$(L\underline{x}, U\underline{x}) = (\underline{x}''', \underline{x}'', \underline{x}', \underline{x}''''). \tag{7}$$

If $q \geq N/2$ $(t = q - N/2)$, then

$$(L\underline{x}, U\underline{x}) = (\underline{x}', \underline{x}'''', \underline{x}''', \underline{x}''). \tag{8}$$

Before proving Theorem 1, we show two relations among the operators L, U, and S_j.

Lemma 1. For any $\underline{x}$ and j,

$$L\underline{x} = S_{(-j \bmod N/2)}\; L\; S_j\underline{x}, \tag{9}$$

$$U\underline{x} = S_{(-j \bmod N/2)}\; U\; S_j\underline{x}. \tag{10}$$

Proof: We prove only (9), the argument for (10) being similar. For $j = N/2$, Equation (9) becomes $L\underline{x} = L\; S_{N/2}\underline{x}$, which can be trivially verified. Thus, index j in (9) can always be taken modulo $N/2$ and it is sufficient to consider $j < N/2$. In this case we have $-j \bmod N/2 = N/2 - j$, and

$$L\; S_j\underline{x} = (min\{x_j, x_{j+N/2}\}, ..., min\{x_{N/2-1}, x_{N-1}\}, min\{x_0, x_{N/2}\}, ..., min\{x_{j-1}, x_{j-1+N/2}\}),$$

or, in compact form, $L\; S_j\underline{x} = S_j\; L\underline{x}$, from which Equation (9) follows by applying $S_{-j \bmod N/2}$ to both sides. □

Proof of Theorem 1: Let the *median* of a sequence $\underline{x} = (x_0, ..., x_{N-1})$ be defined as the minimum value m such that less than $N/2$ elements of $\underline{x}$ are greater than m. Let $\underline{x}$

be bitonic, and let $\underline{y} = S_j\underline{x} = (y_0, ..., y_{N-1})$ consist of a non-decreasing sequence followed by a non-increasing sequence. In general, $\underline{y}$ is the concatenation of five (possibly empty) subsequences respectively containing N_1 elements smaller than m, N_2 elements equal to m, N_3 elements larger than m, N_4 elements equal to m, and N_5 elements smaller than m.

Obviously, $N_1 + ... + N_5 = N$. Also, by the definition of m, $N_3 < N/2$ and $N_2 + N_3 + N_4 \geq N/2$. By simple arithmetic, $N_1 \leq N/2 - N_5$ and there exists a k, with $N_1 \leq k < N/2 - N_5$, such that the sequence $(y_k, y_{k+1}, ..., y_{k+N/2-1})$ contains all the elements larger than m and none of those smaller than m.

We now consider the sequence $\underline{z} = S_{k+N/2}\underline{y} = S_q\underline{x}$, where $q = (j + k + N/2)\ mod\ N$. If we write $\underline{z} = (\underline{z}', \underline{z}'')$, with $\underline{z}'$ and $\underline{z}''$ sequences of $N/2$ elements, it is easy to see that $\underline{z}'' = (y_k, ..., y_{k+N/2-1})$.

Thus all the elements of $\underline{z}''$ are no smaller than the elements of $\underline{z}'$, which implies that $(L\underline{z}, U\underline{z}) = \underline{z}$. Lemma 1 applied to the latter relation yields $L\underline{x} = S_{(q\ mod\ N/2)}\underline{z}'$ and $U\underline{x} = S_{(-q\ mod\ N/2)}\underline{z}''$, which are equivalent to (5) and (6) and establish $P1$.

Property $P2$ follows from (5), (6), and the fact that $max\{z_0, ..., z_{N/2-1}\} \leq min\{z_{N/2}, ..., z_{N-1}\}$.

Sequences $\underline{z}'$ and $\underline{z}''$ are bitonic since they are subsequences of $\underline{z}$, which is bitonic. Then from (5) and (6), $L\underline{x}$ and $U\underline{x}$ are shifts of bitonic sequences, and $P3$ is proved.

If $q < N/2$ and $\underline{x} = (\underline{x}', \underline{x}'', \underline{x}''', \underline{x}'''')$, as defined in $P4$, then $S_q\underline{x} = (\underline{x}'', \underline{x}''', \underline{x}'''', \underline{x}')$. From (6) and (7), cyclically shifting each half of $S_q\underline{x}$ q positions to the right, we obtain $L\underline{x} = (\underline{x}''', \underline{x}'')$ and $U\underline{x} = (\underline{x}', \underline{x}'''')$. This proves (7). A similar argument yields (8), completing the proof of $P4$. □

3 Adaptive Bitonic Merging with $O(N)$ Comparisons

In this section we present a linear-time version of bitonic merging. For simplicity, we describe the algorithm in a sequential setting and we assume that N, the sum of the lengths of the sequences being merged, is a power of two. Parallelism and arbitrary input size are discussed in subsequent sections.

3.1 Analysis of Comparisons

Let $\underline{x}$ be the bitonic sequence obtained by concatenating, in opposite order, two sorted sequences to be merged. The classical bitonic merging consists of the following steps:

1. Compute $L\underline{x}$ and $U\underline{x}$ by $N/2$ (parallel) comparisons (according to definitions (2) and (3)).

2. Recursively sort $L\underline{x}$ and $U\underline{x}$, in parallel.

The comparisons performed by the above algorithm are data-independent and hence can be hardwired in a network of comparator-exchangers, *the bitonic merger*. In the terminology of [Kn73], the bitonic merger has N (a power of two) lines numbered $0, 1, ..., N-1$ and $\log N$ stages each of $N/2$ comparator-exchangers. In the k^{th} stage ($k = 0, 1, ..., \log N - 1$) lines i and j are connected by a comparator-exchanger if and only if the binary spellings of i and j differ exactly in the $(\log N - k)^{th}$ rightmost bit. Comparators output the smaller of the two inputs on the line with lower number.

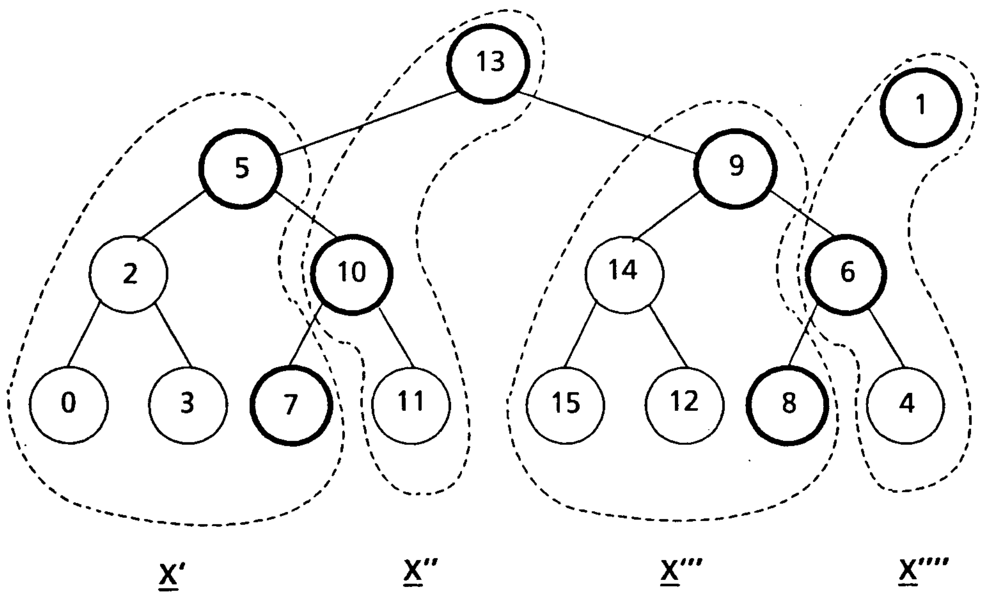
Since, as well known, N comparisons are sufficient to merge two sequences, the set C of the $(N \log N)/2$ comparisons executed by the bitonic merger obviously contains some redundancy. Less obviously, this redundancy can be almost eliminated by executing only a suitable subset of C, as shown in the next theorem. As it will become clear from the proof, this subset is a function of the input sequence $\underline{x}$.

Theorem 2. Let C be the set of the $(N \log N)/2$ comparisons executed by Batcher's merging network. Then, there is a subset $C' \subset C$ of size $|C'| = 2N - \log N - 2$ such that the results of the comparisons in C are uniquely determined by the results of the comparisons in C', as long as the input elements are assumed distinct[3].

Proof: Due to property $P4$ established in Theorem 1, there is a decomposition $(\underline{x}', \underline{x}'', \underline{x}''', \underline{x}'''')$ of $\underline{x}$ such that $(L\underline{x}, U\underline{x})$ is obtained either by exchanging $\underline{x}'$ and $\underline{x}'''$ (7) or by exchanging $\underline{x}''$ and $\underline{x}''''$ (8). Due to $P2$, if $x_{N/2-1} < x_{N-1}$ then (7) holds, else ($x_{N/2-1} > x_{N-1}$) (8) holds.

When (7) holds, $\underline{x}''$ contains x_i if and only if $x_i < x_{N/2+i}$. Thus the leftmost element of $\underline{x}''$ can be found by a binary search driven by comparisons of pairs of the form $(x_i, x_{N/2+i})$. Index i is first set to $N/4 - 1$, and then decremented if $x_i < x_{N/2+i}$ and incremented otherwise. The increment is initialized to $N/8$, and halved at each step. The search terminates after

[3]If they are not, the comparison function can be easily modified to enforce a total ordering on the input elements.



Bitonic tree for the sequence $\underline{x} = (0, 2, 3, 5, 7, 10, 11, 13, 15, 14, 12, 9, 8, 6, 4, 1)$, and the decomposition $(\underline{x}', \underline{x}'', \underline{x}''', \underline{x}'''')$ for the computation of $(L\underline{x}, U\underline{x}) = (\underline{x}', \underline{x}'''', \underline{x}''', \underline{x}'')$. Solid nodes are the ones examined by the binary search.

Figure 1: Bitonic tree

$\log N - 1$ steps, when the increment is zero. Therefore $\underline{x}''$, and thus $(L\underline{x}, U\underline{x})$, is determined by $\log N$ of the $N/2$ comparisons implicit in definitions (2) and (3). The case when (8) holds is completely symmetric.

Let C' be the set of comparisons resulting from recursively applying the above method to $L\underline{x}$ and $U\underline{x}$. Considering that at the k^{th} level of recursion ($k = 0, 1, ..., \log N - 1$) there are 2^k sequences each requiring k comparisons, we can evaluate the cardinality of C' as

$$M(N) \triangleq |C'| = \sum_{k=0}^{\log N - 1} 2^k \log(N/2^k) = 2N - \log N - 2. \quad \square \tag{11}$$

3.2 The Data Structure

Although Theorem 2 gives a way to obtain all the information needed to merge two sequences with a linear number of comparisons, the problem remains of how to efficiently achieve the data rearrangement that in Batcher's network requires $\Omega(N \log N)$ exchanges in the worst case. We solve this problem with the adoption of a suitable data structure, which we call bitonic tree.

A *bitonic tree* (see Figure 1) is a binary tree where each node contains an element from a totally ordered set, and the sequence of elements encountered in the inorder traversal of the tree is bitonic. The bitonic tree is a simple generalization of the binary search-tree. In fact, an inorder traversal of the latter yields a monotonic (sorted) sequence.

Given a bitonic sequence of length N (a power of two), we adopt a representation in which the first $N-1$ elements are stored in a fully balanced bitonic tree of $\log N$ levels, and the last element is kept in a spare node. In this representation, the decomposition $\underline{x} = (\underline{x}', \underline{x}'', \underline{x}''', \underline{x}'''')$ considered in Theorem 1 corresponds to a decomposition of the bitonic tree into four forests (see Figure 1). The roots of the trees in these forests form two parallel paths in the main subtrees of the bitonic tree. The exchange of $\underline{x}'$ and $\underline{x}'''$ [$\underline{x}''$ and $\underline{x}''''$] required by (7) [(8)] to compute $(L\underline{x}, U\underline{x})$ can be accomplished with $O(\log N)$ exchanges of values and pointers in the bitonic tree.

The relation between the bitonic tree and the bitonic network can be viewed as a one-to-one correspondence between nodes and lines, the i^{th} line being associated with the i^{th} node encountered in the inorder traversal of the tree. However, a node and the corresponding line are guaranteed to hold the same value only at the beginning of merging and immediately after the completion of the computation of the sequences $(L\underline{x}, U\underline{x})$ at each level of the recursion.

3.3 The Algorithm

Based on the preceding observations we give below a procedure *bimerge*(*root*, *spare*, *dir*) that sorts $\underline{x}$ by transforming the bitonic tree into a binary search tree. Parameters *root* and *spare* are pointers to the root of the tree and to the spare node, respectively. Parameter *dir* is boolean and represents the direction in which the sequence is to be sorted (*dir* = *false* for ascending, *dir* = *true* for descending).

Each node of the tree has three fields, value, left, and right, respectively storing an element of the sequence (or a pointer to it) and pointers to the left and right subtrees. The procedure bimerge given in Figure 2 is written in pseudo-Pascal. The identifiers not explicitly defined are self-explanatory.

We now briefly comment on procedure bimerge. At the beginning, the root contains $x_{N/2-1}$ and the spare node contains x_{N-1}. After statement 1, rightexchange is *false* when (7) holds (i.e., $\underline{x}'$ and $\underline{x}'''$ are exchanged), and *true* when (8) holds (i.e., $\underline{x}''$ and $\underline{x}''''$ are

```
procedure bimerge (root, spare, dir);
begin
1.  rightexchange : = (root↑value > spare↑value) XOR dir;
2.  if rightexchange then swap-value(root, spare);
3.  pl : = root↑left; pr : = root↑right;
4.  while (pl ≠ nil) do begin
5.      elementexchange : = (pl↑value > pr↑value) XOR dir;
6.      if rightexchange then                 /* X'' and X'''' exchange          */
7.          if elementexchange then begin /* swap values and right subtrees;
                                             search path goes left           */
8.              swap-value(pl, pr);
9.              swap-right(pl, pr);
10.             pl : = pl↑left; pr : = pr↑left
                end
11.         else begin                      /* search path goes right          */
12.             pl : = pl↑right; pr : = pr↑right
                end
13.     else                                /* X' and X''' exchange            */
14.         if elementexchange then begin /* swap values and left subtrees;
                                             search path goes right          */
15.             swap-value(pl, pr);
16.             swap-left(pl, pr);
17.             pl : = pl↑right; pr : = pr↑right
                end
18.         else begin                      /* search path goes left           */
19.             pl : = pl↑left; pr : = pr↑left
                end
    end; /* while */
20.if (root↑left ≠ nil) then begin
21.     bimerge(root↑left, root, dir);
22.     bimerge(root↑right, spare, dir)
    end
  end; /* bimerge*/
```

Figure 2: Procedure Bimerge

exchanged). In the latter case, $x_{N/2-1}$ and x_{N-1} are exchanged (by statement 2). After statement 3, *pl* and *pr* point to the nodes that contain $x_{N/4-1}$ and $x_{3N/4-1}$ respectively.

The binary search for the boundary between $\underline{x}'$ and $\underline{x}''$ (as well as between $\underline{x}''$ and $\underline{x}'''$) is performed by the while-loop (statements 4÷19). At the end of the loop, an inorder traversal of the tree would yield ($L\underline{x}$, $U\underline{x}$). Sequences $L\underline{x}$ and $U\underline{x}$ are recursively sorted by the recursive calls in statements 21 and 22. We observe that $L\underline{x}$ and $U\underline{x}$ are represented, like $\underline{x}$, by a bitonic tree and a spare node: $L\underline{x}$ by the left subtree and the root, $U\underline{x}$ by the right subtree and the original spare node. In general, the recursive calls of depth k (the first call being of depth 0) operate on sequences of length $N/2^k$. These sequences are represented by a subtree with root at level k (the root of the entire tree being at level 0) and a spare node. The spare node belongs to some level less than k or is the spare node of the original tree. More precisely, the i^{th} subtree at level k, from left to right, is paired with the i^{th} node encountered in the inorder traversal of the first k levels of the tree.

A simple analysis of *bimerge* shows that the total number of operations is of the same order as the number of comparisons. Thus, the algorithm runs in linear time.

In the above discussion, specifically Theorem 2, the elements of sequence $\underline{x}$ were assumed to be distinct. In the presence of equal elements ties in comparisons must be broken according to a suitable criterion for the algorithm to work properly. For example, ties can be broken on the basis of the elements' original position in the tree.

4 Parallel Algorithms

In this section we present a parallel version of adaptive bitonic merging and a parallel sorting algorithm based on it. As a model of computation, we choose the PRAC (Parallel Random Access Computer) of [LPV81]. The PRAC is a shared-memory multi-processor machine. Any processor can access any common-memory location in constant time. However, simultaneous access (either read or write) of the same location is illegal and leads to termination error. For more details on the PRAC see [LPV81].

4.1 Merging

We call *stage*(k) of the merge ($k = 0, 1, ..., \log N - 1$) the set of recursive calls of *bimerge* of depth k. There are 2^k such calls, each processing a different subsequence of $N/2^k$ elements. As already observed, this subsequence occupies a subtree with root at level k and a spare node in some level less than k. All the calls in *stage*(k) can be executed in parallel.

We call *phase*(0) of *bimerge* the execution of statements 1, 2, and 3. We call *phase*(i) the execution of the i^{th} iteration of the while-loop. For a call in *stage*(k), i ranges from 1 to $\log N - k - 1$. A phase includes one comparison and a handful of tests and assignment statements. It can be executed in $O(1)$ time.

Since calls of depth k consist of $\log N - k$ phases, *stage*(k) takes $\log N - k$ parallel phases. The total time for executing *stage*(0),...,*stage*($\log N - 1$) in sequence is $O(\sum_{k=0}^{\log N - 1} (\log N - k)) = O(\log^2 N)$. We note a loss in performance with respect to the $O(\log N)$ time of Batcher's network. The loss is due to the difference in the computation of (L $\underline{x}$, U $\underline{x}$). For $\underline{x}$ of length N the $N/2$ comparisons of Batcher's network are data-independent and take only one time step, while the $\log N$ comparisons of our binary-search method are inherently sequential and take $\log N$ steps.

Little can be done to speed up the execution of a single stage without increasing the number of comparisons. However, a careful analysis of the data dependencies between comparisons of the various stages reveals that the execution of different stages can be partially overlapped, with considerable savings in running time.

We observe that, in *stage*(k), *phase*(0) operates on levels $0, 1, ..., k$ of the bitonic tree, and *phase*(i) operates on level $k + i$ ($i = 1, ..., \log N - k - 1$). Thus, *phase*(0) of *stage*(k) can begin as soon as *stage*(0), ..., *stage*($k - 1$) have processed the first k levels of the tree. This condition is satisfied if a new stage is scheduled to begin every other phase step. The entire sequence of $\log N$ stages is completed in $2 \log N - 1$ phase steps, that is in $O(\log N)$ time. An example of the schedule of the phases of different stages is given in Figure 3, for $N = 16$.

In the above schedule, the maximum number of calls simultaneously active is $N/2$, and hence $N/2$ processors are sufficient. We now consider a slightly different schedule that leads to a reduction of the number of processors without substantial degradation in time performance.

PHASE STEP	STAGE (0)		STAGE (1)		STAGE (2)		STAGE (3)	
	phase	tree levels	phase	tree levels	phase	tree levels	phase	tree levels
0	0	0	-	-	-	-	-	-
1	1	1	-	-	-	-	-	-
2	2	2	0	0,1	-	-	-	-
3	3	3	1	2	-	-	-	-
4	-	-	2	3	0	0,1,2	-	-
5	-	-	-	-	1	3	-	-
6	-	-	-	-	-	-	0	0,1,2 ,3

Figure 3: Schedule for the overlapped execution of the stages of *bimerge* for $N = 16$.

Let us assume a number of processors $P = 2^p$, and consider the following strategy.

1. *Stage*(0), ..., *stage*($p-1$) are scheduled to begin one every other step. The total number of calls in these stages is $P - 1$ so that a different processor is available for each call. *Stage*(0) takes $\log N$ phase steps after which a new stage terminates at each phase step. Thus, the first p stages take $t_1 = \log N + p - 1$ phase steps.

2. For the remaining stages, one processor is assigned to each of the P subtrees corresponding to subsequences of length N/P. This will take $t_2 = N/P - 1$ phase steps.

The total number of phase steps is $t = t_1 + t_2 = \log N + p - 2 + N/P$.

If we choose $P = N/2^{\lfloor \log\log N \rfloor} \approx N/\log N$, we obtain $t = 2\log N + 2^{\lfloor \log\log N \rfloor} - \log\log N - 2 < 3\log N$. In conclusion, $T = O(t) = O(\log N)$, with $P = O(N/\log N)$ processors. The product TP is optimal, since N operations are necessary to merge. A similar result is obtained for $1 \leq P \leq N/2^{\lfloor \log\log N \rfloor}$, as summarized in the following theorem.

Theorem 3. Adaptive bitonic merge can be executed on a PRAC of P processors in time $T = O(N/P)$, for $1 \leq P \leq N/2^{\lfloor \log\log N \rfloor}$.

It is of interest to estimate the overhead incurred to distribute the algorithm among P processors. In the parallel version of *bimerge*, the parameters *root*, *spare* and *dir* are

```
procedure bisort(root, spare, dir);
begin
1. If (root↑left ≠ nil) then
      test-and-swap(root,spare,dir)   /* down to leaves - test and swap as needed */
2. else begin
3.      bisort(root↑left, root, dir);
4.      bisort(root↑right, spare, ~dir);
5.      bimerge(root, spare, dir)
6.      end
end;
```

Figure 4: Procedure Bisort.

computed by a processor different from the one that actually executes the call. Thus, the processor that computes the parameters must write them in the shared memory from which the processor assigned to the call will read them. A simple analysis shows that the total number of memory accesses for inter-processor communication is $O(P)$, so that the overhead contributes a lower order term to the total number of operations of the merging algorithm.

4.2 Sorting

Essentially using the classical sort-by-merge scheme (see Figure 4), and standard manipulations, Theorem 2 and 3 lead to the following results.

Theorem 4. Let S be the set of the $N \log N(\log N + 1)/2$ comparisons executed by Batcher's sorting network. Then, there exists a subset $S' \subset S$ of size

$$S(N) \triangleq |S'| = 2N \log N - 4N + \log N + 4 \qquad (12)$$

such that the results of the comparisons in S are uniquely determined by the results of the comparisons in S'.

Theorem 5. Adaptive Bitonic sorting can be implemented on a PRAC of P processors in time $T = O((N \log N)/P)$, for $1 \leq P \leq N/2^{\lfloor \log \log N \rfloor}$.

We observe that $|S'|$ is within a factor of two of $\lceil \log N! \rceil$, which is a lower bound on the number of comparisons needed to sort N elements [Kn73]. It is remarkable that sorting can be done in $O(\log^2 N)$ time with so little redundance. An analysis similar to the one developed for merging, shows that the total number of memory accesses for inter-processor communication is $O(P \log N)$, contributing a lower order term to the total number of operations of our algorithm.

5 Input Sequence of Arbitrary Length

In the previous sections, it has been assumed that N, the number of elements to be sorted, is a power of two. The algorithms so derived can be used for any input sequence after adding enough dummy elements to it so that the length becomes a power of two. However, this strategy leads to a constant factor increase in time complexity, which is undesirable in practical applications.

In this section we modify our algorithm to handle arbitrary values of N, with a negligible increase in complexity with respect to the case where N is a power of two. The basic idea consists in simulating the actions that the power-of-two version of the algorithm would perform on the input sequence padded with dummies, while avoiding representing and processing most of the dummies.

5.1 Padding the Input Sequence

Let $n \triangleq \lceil \log N \rceil$, that is, let 2^n be the minimum power of two non-smaller than N. Given a sequence $\underline{x} = (x_0, ..., x_{N-1})$ to be sorted, we augment it to obtain a sequence $\underline{z} = (d, ..., d, x_0, ..., x_{N-1})$ of length 2^n by inserting, at the beginning of $\underline{x}$, $D \triangleq 2^n - N$ dummy elements of value $d < x_i$, for all x_i's.

If $\underline{z}$ is stored in a tree according to the inorder traversal, the dummy elements occupy a left subforest of the tree. More precisely, let $D = \sum_{i=0}^{n-1} D_i \, 2^i$, with $D_i \in \{0, 1\}$. Let π be the path in the tree that starts at the root and whose i^{th} edge goes left when $D_{n-i} = 0$, and goes right when $D_{n-i} = 1$, for $i = 1, 2, ..., n$. Then, the D dummy elements occupy the nodes of π that are followed by a right edge and the left subtrees of such nodes. In other words, for each $D_i = 1$, there is a dummy subtree of depth i, whose root is at level $n - i$ and is the left son of a node on π, also containing a dummy.

We now analyze the behaviour of our algorithm on a padded sequence $\underline{z}$, focusing in particular on the dummy subtrees.

For a given call to procedure *bimerge*, consider the "parallel" paths p' and p'' traced in the bitonic tree by the search for the boundary between $\underline{x}'$ and $\underline{x}''$ (see Section 3.1). A given subtree T' can be modified by the call only if it is traversed by any of the two paths.

If p' and p'' originate at the root v' of T' or at a descendent of it, then only elements inside T' will be rearranged. In particular, if T' was originally a tree of dummies, it will remain such.

If p' and p'' originate at an ancestor of v', then only one path, say p', can traverse T', and v' is the first node of T' to be visited. During the traversal, v' will be compared with some node v'' , root of a subtree T''. Paths p' and p'' continue in T' and T'', respectively. If T' is a tree of dummies, its elements form a consecutive run in the sorted output. Thus, the comparison of any node of p' with the corresponding node of p'' gives the same result as the comparisons of v' and v''. Therefore, subtrees T' and T'' will either be exchanged completely, or left in their positions.

5.2 Pruning the Tree

The above discussion shows that dummy subtrees are left intact throughout the algorithm, and that they can be processed by examining only their root. This suggests a modification to the bitonic tree whereby a dummy subtree is represented by a single node (its root) with left and right pointers set to ***nil***. We refer to the resulting data structure as the *pruned bitonic tree*.

The procedures *bisort* and *bimerge* have to be modified to work correctly on the pruned version of the tree. The necessary changes are simple, and are outlined below:

1. The call *bisort*(*root*, *spare*, *dir*) is not executed whenever root is a node with dummy value and ***nil*** pointers.

2. The call *bimerge*(*root*, *spare*, *dir*) is not executed whenever root is a node with dummy value and ***nil*** pointers.

3. Whenever a node v', with dummy value and ***nil*** pointers is compared with another node v'', and v' and v'' are exchanged, their left and right pointers are also exchanged. In any case, the current call to *bimerge* is terminated after the comparison.

5.3 Analysis

Let us consider a dummy subtree of depth i ($i \leq n-2$). The savings in comparisons coming from each of the modifications to the algorithm described above are as follows:

1. $(S(2^i) - 1)$ for replacing a call to ***bisort*** with a single comparison between the root of the subtree and a spare node.

2. $(M(2^i) - 1)(n - i)$ for replacing each of the $(n - i)$ calls to ***bimerge*** with a single comparison between root and spare node.

3. $(i - 1)v_i$ for not traversing the $(i - 1)$ levels below the root v' for each of the v_i times in which the dummy subtree should be traversed by a path originating above v'. In general, v_i is a function of the input sequence, but always satisfies the simple bound $v_i \leq (n-i)(n-i+1)/2$, the latter being the number of calls to ***bimerge*** with an ancestor of v' as the root.

We can now estimate the total number of comparisons performed by the pruned-tree sorting algorithm as

$$S(N) = S(2^n) \; - \; \textit{savings} = S(2^n) - \sum_{i=0}^{n-2} d_i[S(2^i) - 1 + (M(2^i) - 1)(m - i) + (i - 1)v_i]$$

and thus

$$S(N) \leq 2N\lceil \log N \rceil - 4N + \textit{lower order terms}. \qquad (13)$$

To obtain the last expression we have: (i) neglected the (negative) contribution of the term $(i - 1)v_i$, (ii) used expressions (11) and (12) for $M(2^i)$ and $S(2^i)$, respectively, and (iii) applied some simple algebraic manipulation. The sublinear terms turn out to be of $O(\log^3 N)$. Comparing (13) with (12) we see that the complexity of the sorting algorithm for arbitrary N differs from the complexity for N a power of two only in sublinear terms. Summarizing the above discussion, we have:

Theorem 6. The pruned-tree version of adaptive bitonic sorting executes a number of comparisons

$$S(N) = 2N\lceil \log N \rceil - 4N + O(\log^3 N). \qquad (14)$$

6 Conclusions

We have presented a parallel sorting algorithm with optimal $TP = O(N \log N)$ complexity for $\Omega(\log^2 N) \leq T \leq O(N \log N)$. To explore the practical potential of our algorithm we must examine the constant factors. The small comparison count ($< 2N \log N$) is encouraging, but we need to consider the total number of operations. To this end, we observe that, when running on P processors, adaptive bitonic sorting executes all the operations that it would execute on one processor plus a small number ($O(P \log N)$) of memory references due to interprocessor communication. Therefore, an accurate estimate of the operation count can be obtained by considering the performance of uni-processor implementations.

We have coded the sequential pruned-tree version of our sorting algorithm, in C under Berkeley Unix 4.2 on a VAX 780, and a Gould 9080. The only optimization we have performed is the straightforward removal of recursion. As a term of comparison, we have chosen "quicker-sort", the Unix system sort, which is a carefully tuned version of quicksort. On sequences of length up to 2^{19} the running time of our algorithm has consistently been below 2.5 times the running time of quicker-sort. This performance is remarkable for an algorithm that, with a small synchronization overhead, can run in $O(\log^2 N)$ parallel time.

The combination of relative simplicity, optimal operation count, and small overhead makes adaptive bitonic sorting appealing for practical implementation.

References

[AKS83] M. Ajtai, J. Komlos and E. Szemeredi, "An $O(\log N)$ sorting network", *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, Boston, MA., pp 1-9, April 1983.

[AS85] S.G. Akl, N. Santoro, "Optimal parallel merging without memory conflicts", *Proceedings of the 1st International Conference on Supercomputing Systems*, St.Petersburg Florida, pp.205-208, December 1985.

[Ba68] K.E. Batcher, "Sorting networks and their applications", *Proceedings of AFIPS Spring joint Computer Conference*, Vol. 32, pp.307-314, April 1968.

[BH85] A. Borodin and J.E. Hopcroft, "Routing, merging, and sorting on parallel models of computation", *Journal of Computer and System Sciences*, Vol.30, No.1, pp.130-145, February 1985.

[Bl85] G. Bilardi, "A family of merging networks derived from the bitonic merger", *Proceedings of the 23rd Annual Allerton Conference on Communication, Control and Computing*, Monticello Illinois, pp.261-267, October 1985.

[BP84] G. Bilardi and F.P. Preparata, "An architecture for bitonic sorting with optimal VLSI performance", *IEEE Transactions on Computers*, Vol.C-33, No.7, pp.646-651, July 1984.

[C86] R. Cole, "Parallel Merge Sort," *Extended Abstract*, 1986.

[HH81] R. Haggkvist and P. Hell, "Parallel sorting with constant time for comparisons", *SIAM Journal on Computing*, Vol.10, No.3, 1981.

[HS82] Z. Hong and R. Sedgewick, "Notes on merging networks", *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pp.296-302, San Francisco California, May 1982.

[Kn73] D.E. Knuth, "The art of computer programming", *Vol.3: Sorting and Searching, Addison Wesley*, Reading Massachusetts, 1973.

[Kr83] C.P. Kruskal, "Searching, merging, and sorting in parallel computation", *IEEE Transactions on Computers*, Vol.C-32, No.10, pp.942-946, October 1983.

[L85] T.F. Leighton, "Theory of Parallel Computation and VLSI", *Class Notes*, MIT, 1985.

[LPV81] G. Lev, N. Pippinger and L.G. Valiant, "A fast parallel algorithm for routing in permutation networks", *IEEE Transactions on Computers*, Vol.C-30, No.2, pp.93-100, February 1981.

[NS79] D. Nassimi and S.Sahni, "Bitonic sort on a mesh-connected parallel computer", *IEEE Transactions on Computers*, Vol.C-28, No.1, pp.2-7, January 1979.

[P78] F.P. Preparata, "New parallel-sorting schemes, *IEEE Transactions on Computers*, Vol.C-27, No.7, pp. 669-673, July 1978.

[Pe77] M.C. Pease, "The indirect binary n-cube microprocessor array", *IEEE Transactions on Computers*, Vol.C-26, No.5, pp.458-473, May 1977.

[Pr83] Y. Perl, "The bitonic and odd-even networks are more than merging", *Department of Computer Science technical report* DCS-TR-123, Rutgers University, February 1983.

[PV81] F.P. Preparata and J. Vuillemin, "The cube-connected-cycles: A versatile network for parallel computation", *Communications of the ACM*, Vol.24, No.5, pp.300-309, May 1981.

[Sn85] M. Snir, "On Parallel Searching", *SIAM journal on Computing*, Vol.14,No.3, pp.688-708, August 1985.

[St71] H.S. Stone, "Parallel processing with the perfect shuffle", *IEEE Transactions on Computers*, Vol.C-20, No.2, pp.153-161, February 1971.

[St78] H.S. Stone, "Sorting on STAR", *IEEE Transactions on Software Engineering*, Vol.SE-4, No.2, March 1978.

[SV81] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model", *Journal of Algorithms*, Vol.2, No.1, pp.88-102, March 1981.

[TK77] C.D. Thompson and H.T. Kung, "Sorting on a mesh-connected computer", *Communications of the ACM*, Vol.20, No.4, pp.263-271, April 1977.

[V75] L.G. Valiant, "Parallelism in comparison problems", *SIAM Journal on Computing*, Vol.4, No.3, pp. 348-355, September 1975.