A Taxonomy of Parallel Sorting

DINA BITTON

Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel

DAVID J. DeWITT

Computer Science Department, University of Wisconsin, Madison, Wisconsin 53706

DAVID K. HSIAO AND JAISHANKAR MENON

Computer and Information Science Department, The Ohio State University, Columbus, Ohio 43210

We propose a taxonomy of parallel sorting that encompasses a broad range of array- and file-sorting algorithms. We analyze how research on parallel sorting has evolved, from the earliest sorting networks to shared memory algorithms and VLSI sorters.

In the context of sorting networks, we describe two fundamental parallel merging schemes: the odd–even and the bitonic merge. We discuss sorting algorithms that evolved from these merging schemes for parallel computers, whose processors communicate through interconnection networks such as the perfect shuffle, the mesh, and a number of other sparse networks. Following our discussion of network sorting algorithms, we describe how faster algorithms have been derived from parallel enumeration sorting schemes, where, with a shared memory model of parallel computation, keys are first ranked and then rearranged according to their rank.

Parallel sorting algorithms are evaluated according to several criteria related to both the time complexity of an algorithm and its feasibility from the viewpoint of computer architecture. We show that, in addition to attractive communication schemes, network sorting algorithms have nonadaptive schedules that make them suitable for implementation. In particular, they are easily generalized to block sorting algorithms, which utilize limited parallelism to solve large sorting problems. We also address the problem of sorting large mass-storage files in parallel, using modified disk devices or intelligent bubble memory. We conclude by mentioning VLSI sorting as an active and promising direction for research on parallel sorting.

Categories and Subject Descriptors: B.3 [**Hardware**]: Memory Structures; B.4 [**Hardware**]: Input/Output and Data Communications; B.7.1 [**Integrated Circuits**]: Types and Design Styles; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms

Additional Key Words and Phrases: Block sorting, bubble memory, external sorting, hardware sorters, internal sorting, limited parallelism, merging, parallel sorting, sorting networks

---

D. Bitton's present address is Department of Computer Science, Cornell University, Ithaca, New York 14853.

CONTENTS

INTRODUCTION

Sorting in computer terminology is defined as the process of rearranging a sequence of values in ascending or descending order. Computer programs such as compilers or editors often choose to sort tables and lists of symbols stored in memory in order to enhance the speed and simplicity of algorithms used to access them (for the search or insertion of additional elements, for instance). Because of both their practical importance and theoretical interest, algorithms for sorting values stored in random access memory (*internal sorting*) have been the focus of extensive research on algorithms. First, *serial sorting* algorithms were investigated. Then, with the advent of parallel processing, *parallel sorting* algorithms became a very active area of research. Many efficient serial algorithms are known which can sort n values in at most $O(n \log n)$ comparisons, the theoretic lower bound for this problem [Knuth 1973]. In addition to their time complexity, various other properties of these *serial internal sorting* algorithms have also been investigated. In particular, sorting algorithms have been evaluated with respect to time-memory trade-offs (the amount of additional memory required to run the algorithm in addition to the memory storing the initial sequence), stability (the requirement that equal elements retain their original relative order), and sensitivity to the initial distribution of the values (in particular, best case and worst case complexity have been investigated).

In the last decade, parallel processing has added a new dimension to research on internal sorting algorithms. Several models of parallel computation have been considered, each with its own idea of "contiguous" memory locations and definition of the way multiple processors access memory. In order to state the problem of parallel sorting clearly, we must first define what is meant by a sorted sequence in a parallel processor. When processors share a common memory, the idea of contiguous memory locations in a parallel processor is identical to that in a serial processor. Thus, as in the serial case, the time complexity of a sorting algorithm can be expressed in terms of number of comparisons (performed in parallel by all or some of the processors) and internal memory moves. On the other hand, when processors do not share memory and communicate along the lines of an interconnection network, definition of the sorting problem requires a convention to order the processors and thus the union of their local memory locations. When parallel processors are used, the time complexity of a sorting algorithm is expressed in terms of parallel comparisons and exchanges between processors that are adjacent in the interconnecting network.

Shared memory models of parallel computation have been instrumental in investigating the intrinsic parallelism that exists in the sorting problem. Whereas the first results on parallel sorting were related to

sorting networks [Batcher 1968], faster parallel sorting algorithms have been proposed for theoretical models of parallel processors with shared memory [Hirschberg 1978; Preparata 1978]. A chain of results in shared memory computation has led to a number of parallel sorting schemes that exhibit a $O(\log n)$ time complexity. Typically, the parallel sorting problem is expressed as that of sorting n numbers with n or more processors, all sharing a large common memory, so that they may access with various degrees of contention (e.g., parallel reads and parallel writes with arbitration). Research on parallel sorting has been largely concerned with purely theoretical issues, and it is only recently that feasibility issues such as limited parallelism or, in the context of very large scale integration (VLSI) sorting, trade-offs between hardware complexity (expressed in terms of chip area) and time complexity are being addressed.

In addition to using sorting algorithms to rearrange numbers in memory, sorting is often advocated in the context of information processing. In this context, sorting is used to order a file of data records, stored on a mass-storage device. The records are ordered with respect to the value of a key, which might be a single field or the concatenation of several fields in the record. Files are sorted either to deliver well-organized output to a user (e.g., a telephone directory), or as an intermediate step in the execution of a complex database operation [Bitton and DeWitt 1983; Selinger et al. 1979]. Because of memory limitations file sorting cannot be performed in memory and *external sorting algorithms* must be used. External sorting schemes are usually based on iterative merging [Knuth 1973, sec. 5.4]. Even when fast disk devices are used as mass-storage devices, input/output accounts for most of the execution time in external sorting.[1]

Despite the obvious need for fast sorting of large files, the availability of parallel processing has not generated much interest in research on new external sorting schemes. The reasons for the relatively small amount of research on parallel external sorting [Bitton-Friedland 1982; Even 1974] are most likely related to the necessity of adapting such schemes to mass-storage device characteristics.

It may seem that advances in computer technology, such as the advent of intelligent or associative memories, could eliminate or reduce the use of sorting as a tool for performing other operations. For example, when sorting is used in order to facilitate searching, one may advocate that associative memories will suppress the need of sorting. However, associative stores remain too expensive for widespread use, especially when large volumes of data are involved. In the case where sorting is required for the sole purpose of ordering data, the only way to reduce sorting time is to develop fast parallel sorting schemes, possibly by integrating sorting capability into mass-storage memory [Chen et al. 1978; Chung et al. 1980].

In this paper, we propose a taxonomy of parallel sorting that includes both internal and external parallel sorting algorithms. We analyze how research on parallel sorting has evolved from the earliest sorting networks to shared memory model algorithms and VLSI sorters. We attempt to classify a broad range of parallel sorting algorithms according to various criteria, including time efficiency and the architectural requirements upon which they depend. The goal of this study is to provide a basic understanding and a unified view of the body of research on parallel sorting. It would be beyond the scope of a single paper to survey the proposed models of computation in detail or to analyze in depth the complexity of the various algorithms surveyed. We have kept to a minimum the discussion on algorithm complexity, and we only describe the main upper-bound results for the number of parallel comparison steps required by the algorithms. Rather than theoretical problems related to parallel sorting (which have been treated in depth in a number of studies, e.g., Borodin and Hopcroft [1982], Shiloach and Vishkin [1981], and Valiant [1975]), we emphasize

[1] It is estimated that the OS/VS Sort/Merge program consumes as much as 25 percent of all input/output time on IBM systems [Bryant 1980].

problems related to the feasibility of parallel sorting with present or near-term technology.

The remainder of this paper is organized as follows. In Section 1, we show that certain fast serial sorting algorithms can be parallelized, but that this approach leads to simple and relatively slow parallel algorithms. Section 2 is devoted to network sorting algorithms; in particular, we describe in detail several sorting networks that perform Batcher's bitonic sort. In Section 3 we survey a chain of results that led to the development of very fast sorting algorithms: the shared memory model parallel merging [Gavril 1975; Valiant 1975] and the shared memory sorting algorithms [Hirschberg 1978; Preparata 1978]. In Section 4 we address the issue of limited parallelism and, in this context, we define "block sorting" parallel algorithms, which sort $M \cdot p$ elements with p processors. We then identify two methods for deriving a block sorting algorithm. In Section 5, we address the problem of sorting a large file in parallel. We show that previous results on parallel sorting are mostly applicable to *internal sorting* schemes, where the array to be sorted is entirely stored in memory, and propose external parallel sorting as a new research direction. Section 6 contains an overview of recently proposed designs for dedicated sorting devices. In Section 7 we summarize this survey and indicate possible directions for future research.

1. PARALLELIZING SERIAL SORTING ALGORITHMS

Parallel processing makes it possible to perform more than a single comparison during each time unit. Some models of parallel computation (the sorting networks in particular) assume that a key is compared to only one other key during a time unit, and that parallelism is exploited to compare different pairs of keys simultaneously. Another possibility is to compare a key to many other keys simultaneously. For example, in Muller and Preparata [1975], a key is compared to $(n - 1)$ other keys in a single time unit by using $(n - 1)$ processors.

Parallelism can also be exploited to move many keys simultaneously. After a parallel comparison, step, processors conditionally exchange data. The concurrency that can be achieved in the exchange steps is limited either by the interconnection scheme between the processors (if one exists) or by memory conflicts (if shared memory is used for communication).

With a parallel processor, the analog to a comparison and move step in a uniprocessor memory is a parallel comparison–exchange of pairs of keys. Thus it is natural to measure the performance of parallel sorting algorithms in terms of the number of comparison–exchanges that they require. The *speedup* of a parallel sorting algorithm then can be defined as the ratio between the number of comparison–moves required by an optimal serial sorting algorithm and the number of comparison–exchanges required by the parallel algorithm.

Since a serial algorithm that sorts by comparison requires at least $O(n \log n)$ comparisons to sort n elements [Knuth 1973, p. 183], the optimal speedup would be achieved when, by using n processors, n elements are sorted in $O(\log n)$ parallel comparisons. It does not, however, seem possible to achieve this bound by simply parallelizing one of the well-known $O(n \log n)$-time serial sorting algorithms. These algorithms appear to have serial constraints that cannot be relaxed. Consider, for example, a two-way merge sort [Knuth 1973, p. 160]. The algorithm consists of $\log n$ phases. During each phase, pairs of sorted sequences (produced in the previous phase) are merged into a longer sequence. During the first phases, a large number of processors can be used to merge different pairs in parallel. However, there is no obvious way to introduce a high degree of parallelism in later phases. In particular, the last phase that consists of merging two sequences, each of which contains $n/2$ elements, is a serial process that may require as many as $n - 1$ comparisons.

On the other hand, parallelization of straight sorting methods requiring $O(n^2)$ comparisons seems easier. However, this approach can at best produce $O(n)$-time parallel sorting algorithms when $O(n)$ processors are used, since by performing n comparisons instead of 1 in a single time

unit, the execution time can be reduced from $O(n^2)$ to $O(n)$. An example of this kind of parallelization is a well-known parallel version of the common bubble sort called the *odd–even transposition sort* (Section 1.1).

Partial parallelization of a fast serial algorithm can also lead to a parallel algorithm of order $O(n)$. For example, the serial tree selection sort can be modified so that all comparisons at the same level of the tree are performed in parallel. The result is a parallel tree sort, described in Section 1.2. This parallel algorithm is used in the database Tree Machine [Bentley and Kung 1979].

1.1 The Odd–Even Transposition Sort

The serial "bubble sort" proceeds by comparing and exchanging pairs of adjacent items. In order to sort an array $(x_1, x_2, \ldots, x_n)$, $(n-1)$ comparison–exchanges (x_1, x_2), $(x_2, x_3), \ldots, (x_{n-1}, x_n)$ are performed. This results in placing the maximum at the right end of the array. After this first step, x_n is discarded, and the same "bubble" sequence of comparison–exchanges is applied to the shorter array $(x_1, x_2, \ldots, x_{n-1})$. By iterating $(n-1)$ times the entire sequence is sorted.

The serial odd–even transposition sort [Knuth 1973] is a variation of the basic bubble sort, with a total of n phases, each of which requires $n/2$ comparisons. Odd and even phases alternate. During an odd phase, odd elements are compared with their right adjacent neighbor; thus the pairs (x_1, x_2), (x_3, x_4), ... are compared. During an even phase, even elements are compared with their right adjacent neighbor; that is, the pairs (x_2, x_3), (x_4, x_5), ... are compared. To completely sort the sequence, a total of n phases (alternately odd and even) is required [Knuth 1973, p. 65].

This algorithm calls for a straightforward parallelization [Baudet and Stevenson 1978]. Consider n linearly connected processors and label them $P_1, P_2, \ldots, P_n$. Assume that the links are bidirectional so that P_i can communicate with both P_{i-1} and P_{i+1}. Also assume that initially x_i resides in P_i for $i = 1, 2, \ldots, n$. To sort $(x_1, x_2, \ldots, x_n)$ in parallel, let $P_1, P_3, P_5, \ldots$ be active during the odd time steps, and execute the odd phases of the serial odd–even transposition sort in parallel. Similarly, let $P_2, P_4, \ldots$ be active during the even time steps, and perform the even phases in parallel.

Note that a single comparison–exchange requires two transfers. For example, during the first step, x_2 is transferred to P_1 and compared to x_1 by P_1. Then, if $x_1 > x_2$, x_1 is transferred to P_2; otherwise x_2 is transferred back to P_2. Thus the parallel odd–even transposition algorithm sorts n numbers with n processors in n comparisons and $2n$ transfers.

1.2 A Parallel Tree-Sort Algorithm

In a serial tree selection sort, a binary tree data structure with $(2n-1)$ nodes is used to sort n numbers. The tree has n leaves, and initially one number is stored in each leaf. Sorting is performed by selecting the minimum of the n numbers, then the minimum of the remaining $(n-1)$ numbers, etc.

The binary tree structure is used to find the minimum by iteratively comparing the numbers in two sibling nodes, and moving the smaller number to the parent node (see Figure 1). By simultaneously performing all the comparisons at the same level of the binary tree, a parallel tree sort is obtained [Bentley and Kung 1979].

Consider a set of $(2n-1)$ processors interconnected to form a binary tree with one processor at each of n leaf nodes and at each interior node of the tree. By starting with one number at each leaf processor, the minimum can be transferred to the root processor in $\log_2(n)$ parallel comparison and transfer steps. At each step, a parent receives an element from each of its two children, performs a comparison, retains the smaller element, and returns the larger one. After the minimum has reached the root, it is written out. From then on, empty processors are instructed to accept data from nonempty children and select the minimum if they receive two elements. At every other step, the next elements in increasing order reaches the root. Thus sorting is completed in time $O(n)$.

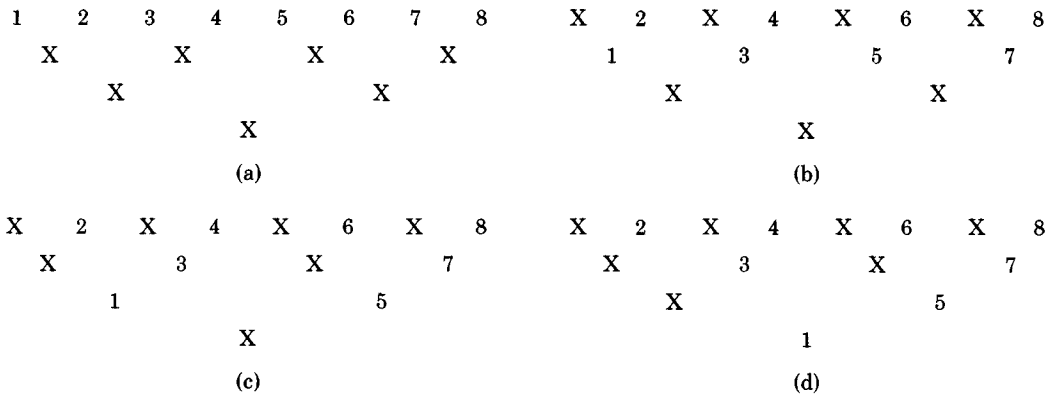


Figure 1. Parallel tree selection sort. (a) Step 1. (b) Step 2. (c) Step 3. (d) Step 4.

Both the odd–even transposition sort and the parallel tree sort constitute two simple parallel sorting algorithms, derived by performing in parallel the sequence of comparisons required at different stages of the bubble sort and the tree selection sort. Both algorithms use $O(n)$ processors to sort an arbitrary sequence of n elements in $O(n)$ comparisons. We shall show that parallel sorting algorithms developed by exploiting the intrinsic parallelism in sorting are faster than those developed by parallelizing serial sorting algorithms.

2. NETWORK SORTING ALGORITHMS

It is somehow surprising that initially the simple hardware problem of designing a multiple-input multiple-output switching network has been a prime motivation in developing parallel sorting algorithms. The earliest results in parallel sorting are found in the literature on sorting networks [Batcher 1968; Van Voorhis 1971]. Since then, a wide range of network topologies have been proposed, and their ability to support fast sorting algorithms has been extensively investigated. In Section 2.1, we describe in detail the odd–even and the bitonic merging networks. In Section 2.2, we show that parallel sorting algorithms for SIMD (single instruction multiple data stream) machines are derived from the bitonic network sort. In particular, we describe two bitonic sort algorithms for a mesh-connected processor [Nassimi and Sahni 1979; Thompson and Kung 1977].

Several other networks are of major interest, particularly the cube [Pease 1977] and the cube-connected cycles [Preparata and Vuillemin 1979], which are suitable for sorting as well as for a number of numerical problems. It has been shown that sorting based on the bitonic merge can be implemented as a routing strategy on these networks. It is beyond the scope of this paper to investigate these networks in detail; we shall concentrate on explaining the basic merge patterns that determine the routing strategies on all these networks and deriving the $O(\log^2 n)$ lower bound for sorting time on Batcher's networks.

2.1 Sorting Networks

Sorting networks originated as fast and economical switching networks. Since a sorting network with n input lines can order any permutation of $(1, 2, \ldots, n)$, it can be used as a multiple-input multiple-output switching network [Batcher 1968]. Implementing a serial sorting algorithm on a network of comparators [Knuth 1973, p. 220] results in a serialization of the comparators and consequently increases the network delay. To make a sorting network fast, it is necessary to have number of comparator modules perform comparisons in parallel. Parallel sorting algorithms are therefore necessary to design efficient sorting networks.

One of the first results in parallel sorting is due to Batcher [1968], who presented two methods to sort n keys with $O(n \log^2 n)$

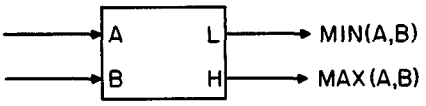


Figure 2. A comparison–exchange module.

comparators in time $O(\log^2 n)$. As shown in Figure 2, a comparator is a module that receives two numbers on its two input lines A, B, and outputs the minimum on its output line L and the maximum on its output line H. A serial comparator receives A and B with their most significant bit first, and can be realized with a small number of gates. Parallel comparators compare several bits in parallel at each step; they are faster but obviously more complex. Both of Batcher's algorithms, the "odd–even sort" and the "bitonic sort," are based on the principle of iterated merging. A specific iterative rule is applied to an initial sequence of 2^k numbers in order to create sorted runs of length 2, 4, 8, . . . , 2^k during successive stages of the algorithm.

2.1.1 The Odd–Even Merge Rule

The iterative rule for the odd–even merge is illustrated in Figure 3. Given two sorted sequences $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$, two new sequences are created: The odd sequence consists of the odd-numbered terms and the even sequence consists of the even-numbered terms. The odd sequence $(c_1, c_2, \ldots)$ is obtained by merging the odd terms $(a_1, a_3, \ldots)$ with the odd terms $(b_1, b_3, \ldots)$. Similarly, the even sequence $(d_1, d_2, \ldots)$ is obtained by merging $(a_2, a_4, \ldots)$ with $(b_2, b_4, \ldots)$. Finally, the sequence $(c_1, c_2, \ldots)$ and $(d_1, d_2, \ldots)$ are merged into $(e_1, e_2, \ldots, e_{2n})$ by performing the following comparison–exchanges:

$$\begin{aligned} e_1 &= c_1, \\ e_{2i} &= \min(c_{i+1}, d_i), \\ e_{2i+1} &= \max(c_{i+1}, d_i) \qquad \text{for } i = 1, 2, \ldots, \\ e_{2n} &= d_n. \end{aligned}$$

The resulting sequence will be sorted (for a proof, see Knuth [1973, pp. 224, 225]). To sort 2^k numbers using the odd–even iterative merge rule requires 2^{k-1} (1 by 1) merging networks (i.e., comparison–exchange modules), followed by 2^{k-2} (2 by 2) merging networks, followed by 2^{k-3} (4 by 4) merging networks, and so on. Since a 2^{i+1} by 2^{i+1} merging network requires one more step of comparison–exchange than a 2^i by 2^i merging network, it follows that an input number goes through at most $1 + 2 + 3 + \cdots + k = k(k + 1)/2$ comparators. This means that 2^k numbers are sorted by performing $k(k + 1)/2$ parallel comparison–exchanges. However, the number of comparators required by this type of sorting network is $(k^2 - k + 4)2^{k-2} - 1$ [Batcher 1968].

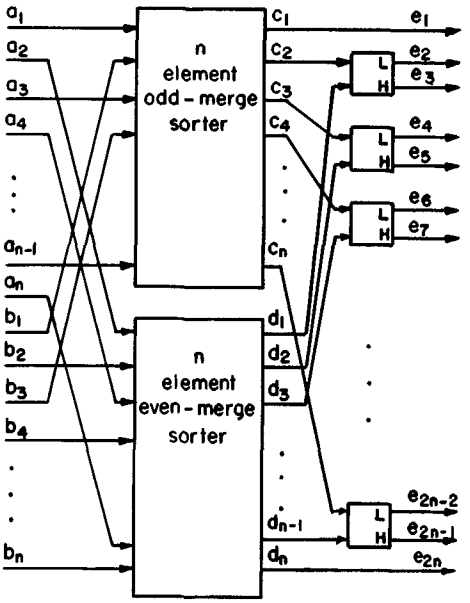


Figure 3. The iterative rule for the odd–even merge.

2.1.2 The Bitonic Merge Rule

For the bitonic sort, a different iterative rule is used (Figure 4). A bitonic sequence is obtained by concatenating two monotonic sequences, one ascending and the other descending. A cyclic shift of this concatenated sequence is also a bitonic sequence. The bitonic iterative rule is based on the observation that a bitonic sequence can be split into two bitonic sequences by performing a single step of comparison–exchanges. Let $(a_1, a_2, \ldots, a_{2n})$ be a bitonic sequence such that $a_1 \leq a_2 \leq \cdots \leq a_n$ and

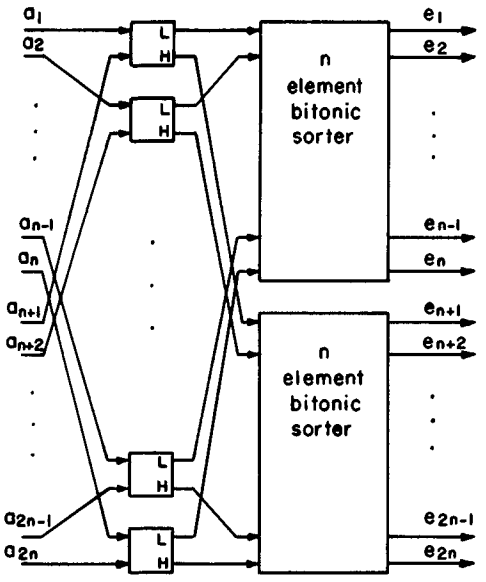


Figure 4. The iterative rule for the bitonic merge.

$a_{n+1} \geq a_{n+2} \geq \cdots \geq a_{2n}$.[2] Then the sequences

$$\min(a_1, a_{n+1}), \min(a_2, a_{n+2}), \ldots$$

and

$$\max(a_1, a_{n+1}), \max(a_2, a_{n+2}), \ldots$$

are both bitonic. Furthermore, the first sequence contains the n lower elements of the original sequence, whereas the second contains the n higher ones. It follows that a bitonic sequence can be sorted by separately sorting two bitonic sequences each half as long as the original sequence.

To sort 2^k numbers by using the bitonic iterative rule, we can iteratively sort and merge sequences into larger sequences until a bitonic sequence of 2^k is obtained. This bitonic sequence can be split into "lower" and "higher" bitonic subsequences. Note that the iterative building procedure of a bitonic sequence requires that some comparators invert their output lines and output a pair of numbers in decreasing order (to produce the decreasing part of a bitonic sequence). Figure 5 illustrates a bitonic sort network for eight input lines. In general, the bitonic sort of 2^k numbers requires $k(k + 1)/2$ steps, with each step using 2^{k-1} comparators.

After the first bitonic sorter was presented, it was shown that the same sorting scheme could be realized with only $n/2$ comparators, interconnected as a *perfect shuffle* [Stone 1971]. Stone noticed that if the input values were labeled with a binary index, then the indices of every pair of keys entering a comparator at any step of the bitonic sort would differ by a single bit in their binary representations. Stone also made the following observations: The network has $\log n$ stages. The ith stage consists of i steps and at step i, inputs that differ in their least significant bit are compared. This regularity in the bitonic sorter suggests that a similar interconnection scheme could be used between the comparators of any two adjacent columns of the network.

Stone concluded that the *perfect shuffle* interconnection could be used throughout all the stages of the network. "Shuffling" in input lines (in a manner similar to shuffling a deck of cards) is equivalent to shifting their binary representation to the left. Shuffling twice shifts the binary representation of each index twice. Thus the input lines can be ordered before each step of comparison–exchanges by shuffling them as many times as required by the bitonic sort algorithm. The network that realizes this idea is shown in Figure 6 for eight input lines. In general, for $n = 2^k$ input lines, this type of bitonic sorter requires a total of $(n/2)(\log n)^2$ comparators, arranged in $(\log n)^2$ ranks of $(n/2)$ comparators each. The network has $\log n$ stages, with each stage consisting of $\log n$ steps. At each step, the output lines are shuffled before they enter the next rank of comparators. The comparators in the first $(\log n) - i$ steps of the ith stage do not exchange their inputs; their only use is to shuffle their input lines.

As an alternative to a multistage network, the bitonic sort can also be implemented as a recirculating network, which requires a much smaller number of comparators. For example, an alternative bi-

[2] As pointed out by one of the referees, the restriction to equal-length ascending and descending parts is not necessary. However, we have made this assumption for the sake of clarity in explaining the bitonic merge rule.

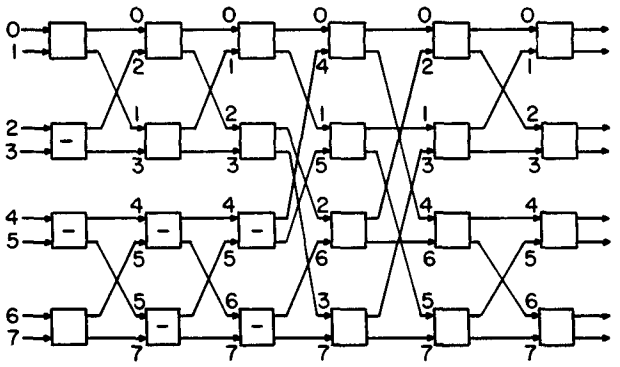


Figure 5. Batcher's bitonic sort for eight numbers. The boxes containing minus signs indicate comparators that invert their output lines.

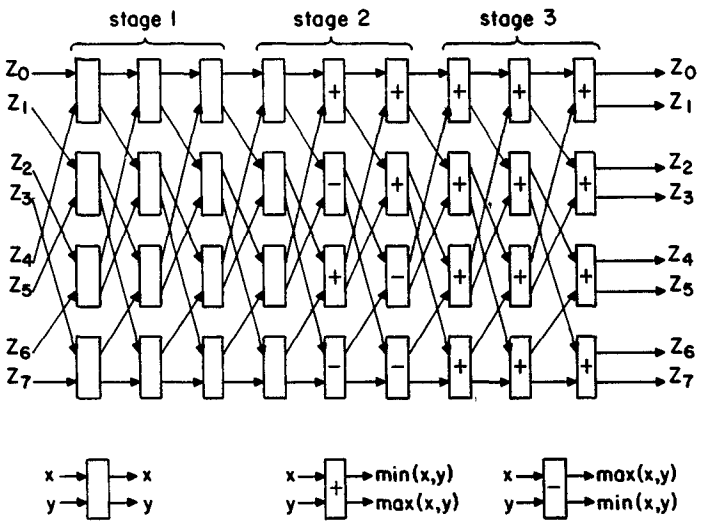


Figure 6. Stone's modified bitonic sort. The boxes containing minus signs indicate comparators that invert their output lines.

tonic sorter can be built with a single rank of comparators connected by a set of shift registers and shuffle links, as shown in Figure 7. Since the ith stage of the bitonic sort algorithm requires i comparison–exchanges, Batcher's sort requires

$$1 + 2 + 3 + \cdots + \log n$$
$$= \log n(\log n + 1)/2$$

parallel comparison–exchanges. Stone's bitonic sorter, on the other hand, requires a total of $(\log n)^2$ steps because additional steps are needed for shuffling the input lines (without performing a comparison). In both cases, the asymptotic complexity is $O(\log^2 n)$ comparison–exchanges. This provides a speedup of $O(n/\log n)$ over the $O(n \log n)$ complexity of serial sorting. Therefore, this algorithm significantly improves the previous known bound of $O(n)$ for the time required to sort n elements with n processing elements.

Siegel has shown that the bitonic sort can be also performed by other types of networks in time $O(\log^2 n)$ [Siegel 1977]. The cube and the plus–minus 2^i networks are among the networks that he considered. Essentially, the data exchanges required by the bitonic sort scheme can be realized on these networks as well (in fact, the perfect shuffle may be seen as an emulator of the cube). Siegel proves that simulating the shuffle on a large class of interconnection networks takes $O(\log^2 n)$ time, and thus sorting can also be performed within this time limit. Finally, we should also mention the versatile cube-connected cycle (CCC), a

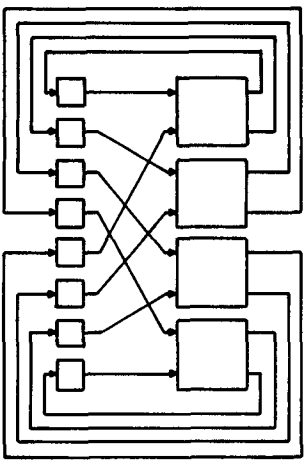


Figure 7. Stone's architecture for the bitonic sort.

network that efficiently emulates the cube and the shuffle, and yet requires only three communication ports per processor [Preparata and Vuillemin 1979]. A bitonic or an odd–even sort can also be performed on the CCC in time $O(\log^2 n)$.

2.2 Sorting on an SIMD Machine

Sorting networks are characterized by their property of nonadaptivity. They perform the same sequence of comparisons, regardless of the results of intermediate comparisons. In other words, whenever two keys R_i and R_j are compared, the subsequent comparison for R_j is the same in the case where $R_i < R_j$ as it would have been in the case where $R_j < R_i$. As a result of the nonadaptivity property, a network sorting algorithm is conveniently implemented on an SIMD machine. An SIMD (single instruction stream, multiple data stream) machine is a system consisting of a control unit and a set of processors with local memory interconnected by an interconnection network. The processors are highly synchronized. The control unit broadcasts instructions that all active processors execute simultaneously (a mask specifies a subset of processors that are idle during an instruction cycle). Since the sequence of comparisons and transfers required in a network sorting algorithm is determined when the sorting operation is initialized, a central controller can supervise the execution of the algorithm by broadcasting at each time step the appropriate compare–exchange instruction to the processors.

2.2.1 Sorting on an Array Processor

The sorting problem can be also defined as the problem of permuting numbers among the local memories associated with the processors of an SIMD machine. In particular, by assuming that one number is stored in the local memory of each processor of a mesh-connected machine, sorting can be seen as the process of permuting the numbers stored in neighboring processors until they conform to some ordering of the mesh.

The processors of an n by n mesh-connected parallel processor can be indexed according to a specified rule, such as the row-major or column-major indexing, which are commonly accepted ways to order an array. Thompson and Kung [1977] adapted the bitonic sorting scheme to a mesh-connected processor, with three alternative indexing rules: the row-major rules, the snakelike row-major rules, and the shuffled row-major rules. These rules are shown in Figure 8.

By assuming that n^2 keys with arbitrary values are initially distributed so that exactly one number resides in each processor, the sorting problem consists of moving the ith smallest number to the processor indexed by i, for $i = 1, \ldots, n^2$. As with sorting networks, parallelism is used to compare pairs of numbers simultaneously, and a number is compared to only one other number at any given unit of time. Concurrent data movement is allowed but only in the same direction; that is, all processors can simultaneously transfer the content of their transfer register to their right, left, above, or below neighbor. This computation model is SIMD since at each time unit a single instruction (compare or move) can be broadcast for concurrent execution by the set of processors specified in the instruction. The complexity of a method that solves the sorting problem for this model can be measured in terms of the number of

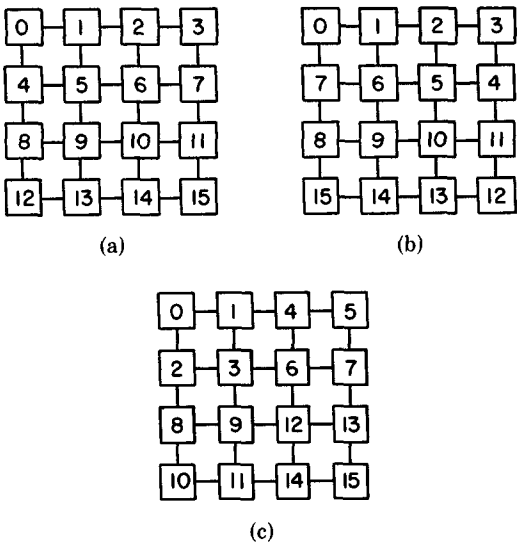


Figure 8. Array processor, indexing schemes. (a) Row-major indexing. (b) Snakelike row-major indexing. (c) Shuffled row-major indexing.

comparison and *unit-distance routing* steps. For the rest of this section we refer to the unit-distance routing step as *a move*. Any algorithm that is able to perform a permutation for this model will require at least $4(n-1)$ moves, since it may have to interchange the elements from two opposite corners of the array processor (this is true for any indexing scheme). In this sense a sorting algorithm that requires $O(n)$ moves is optimal.

The odd–even and the bitonic network sorting algorithms were adapted to this parallel computation model, leading to two algorithms that perform the mesh sort in $O(n)$ comparisons and moves [Thompson and Kung 1977]. The first algorithm uses an odd–even merge of two-dimensional arrays and orders the keys with snakelike row-major indexing. The second uses a bitonic sort and orders the keys with shuffled row-major indexing. A third algorithm that sorts in row-major order with similar performance was later obtained [Nassimi and Sahni 1979]. This algorithm is also an adaptation of the bitonic sort in which the iterative rule is a merge of two-dimensional arrays. Finally, an improved version of the two-dimensional odd–even merge was recently proposed [Kumar and Hirschberg 1983]. On the basis of this merge pattern, a two-dimensional array can be sorted in row-major order, in time $O(n)$, and with a smaller proportionality constant than the previous algorithms.

2.3 Summary

In this section we have examined two well-known sorting networks, the odd–even and bitonic networks, and shown that the concept of a sorting network has been extended to various schemes of synchronous parallel sorting. Although some consideration was given to the hardware complexity, the complexity of sorting on these networks has mainly been characterized in terms of execution time and number of processing elements utilized. Thus, our baseline for evaluating the various sorting schemes employed by these networks was the number of comparison–exchanges required, and we did not systematically account for the degree of network interconnection as a complexity measure of the network sorting algorithms. It is beyond the scope of this study to provide a comprehensive analysis of interconnection networks. Extensive literature exists on this topic, and we have listed some references for the interested reader [Feng 1981; Nassimi and Sahni 1982; Pease 1977; Preparata and Vuillemin 1979; Siegel 1977, 1979; Thompson 1980].

Until very recently, the best-known performance for sorting networks was an $O(\log^2 n)$ sorting time with $O(n \log^2 n)$ comparators. We have shown how the bitonic network sort can be interpreted as a sorting algorithm that sorts n numbers in time $O(\log^2 n)$ with $n/2$ processors. In Section 3, we show that, in an attempt to develop faster parallel sorting algorithm, a more flexible parallel computation model than the network comparators—the shared memory model—has been successfully investigated. However, a recent theoretical result may renew the interest in network sorting algorithms [Ajtai et al. 1983], showing a network of $O(n \log n)$ comparators that can sort n numbers in $O(\log n)$ comparisons. Unfortunately, unlike the odd–even or the bitonic sort, this algorithm is not suitable for implementation. It is based

on a complex graph construction that may make the proportionality constant (in the lower bound for the number of comparisons) unacceptably high.

3. SHARED MEMORY PARALLEL SORTING ALGORITHMS

After the time bound of $O(\log^2 n)$ was achieved with the network sorting algorithms, researchers attempted to improve this to the theoretical lower bound of $O(\log n)$. In this section, we describe several parallel algorithms that sort n elements with $O(\log n)$ comparisons. These algorithms assume a shared memory model of parallel computation.

Although the sorting network algorithms are based on comparison–exchanges of pairs, shared memory algorithms generally use *enumeration* to compute the rank of each element. Sorting is performed by computing in parallel the rank of each element, and routing the elements to the location specified by their rank. Thus, in the network sorting algorithms, individual processors decide *locally* about their next routing step by communicating with their nearest neighbors, whereas in the shared memory algorithms, any processor may access any location of the global memory at every step of the computation. As shown in Section 2, the network algorithms assume a sparse interconnection scheme and differ only by the network interconnection topology. The shared memory sorting algorithms rely on parallel computation models that differ in whether or not they allow read and write conflicts and how they resolve these conflicts [Borodin and Hopcroft 1982]. Clearly, the shared memory models are more powerful. However, at present, they are largely of theoretical interest, whereas the network models are suitable to implemention with current or near-term technology.

In the remainder of this section, we first describe a modified sorting network scheme that sorts by enumeration, using $O(n^2)$ processing elements [Muller and Preparata 1975]. We then survey two parallel merge algorithms that are faster than the nonadaptive network merge algorithms (the odd–even and the bitonic merge described in Section 2) and sorting algorithms that combine enumeration with parallel merge procedures [Preparata 1978]. In addition to these enumeration sorting algorithms, we also describe a parallel bucket sorting algorithm [Hirschberg 1978].

3.1 A Modified Sorting Network

In an attempt to reduce the number of comparisons required for sorting by increasing the degree of parallelism beyond $O(n)$, Muller and Preparata [1975] first proposed a modified sorting network based on a different type of comparators (Figure 9). These comparators have two input lines and one output line. The two numbers to compare are received on the A and B lines. The output bit x is 0 if $A < B$ and 1 if $A > B$. To sort a sequence of n elements, each element is simultaneously compared to all the others in one unit of time by using a total of $n(n-1)$ comparators. The output bits from the comparators are then fed into a parallel counter that computes in $\log n$ steps the rank of an element by counting the number of bits set to 1 as a result of comparing this element with all the other $(n-1)$. Finally, a switching network, consisting of a binary tree with $(\log n) + 1$ levels of single-pole double-throw switches, routes the element of rank i to the ith terminal of the tree. There is one such tree for each element, and each tree uses $(2n-1)$ switches. Routing an element through this tree requires $\log n$ time units, which determines the algorithm's complexity. At the cost of additional hardware complexity, this algorithm sorts n elements in $O(\log n)$ time with $O(n^2)$ processing elements. Muller and Preparata's algorithm was the first to use an *enumeration scheme* for parallel sorting.

The idea of sorting by enumeration was exploited to develop other very fast parallel sorting algorithms [Hirschberg 1978; Preparata 1978], which improve Muller and Preparata's result by reducing the number of processing elements. Even from a theoretical point of view, the requirement of n^2 processors for achieving a speed of $O(\log n)$ is not satisfactory. A parallel sort-

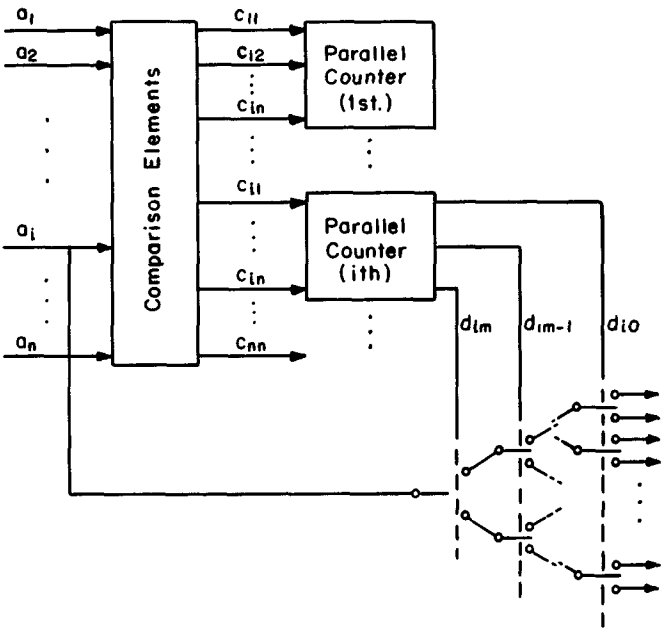


Figure 9. Muller and Preparata's [1975] sorting network.

ing algorithm could theoretically achieve the same speed with only $O(n)$ processors if it had a parallel speedup of order n.

3.2 Faster Parallel Merging Algorithms

Optimal parallel sorting algorithms may use fast merging procedures in addition to enumeration. In a study of parallelism in comparison problems, Valiant [1975] presents a recursive algorithm that merges two sorted sequences of n and m elements ($n \leq m$) with mn processors in $2 \log(\log n) + O(1)$ comparison steps (compared to $\log n$ for the bitonic merge). On the other hand, Gavril [1975] proposes a fast merging algorithm that merges two sorted sequences of length n and m with a smaller number of processors $p \leq n \leq m$. This algorithm is based on binary insertion and requires only $2 \log(n + 1) + 4(n/p)$ comparisons when $n = m$.

Both Valiant's and Gavril's merging algorithms assume a shared memory model of computation. All the processors utilized can access elements of the initial data simultaneously, or intermediate computation results.

3.3 Bucket Sorting

Hirschberg's [1978] "bucket sort" algorithm sorts n numbers with n processors in time $O(\log n)$, provided that the numbers to be sorted are in the range $\{0, 1, \ldots, m - 1\}$. A side effect of this algorithm is that duplicate numbers that may appear in the initial sequence are eliminated in the sorting process. If memory conflicts could be ignored, there would be a straightforward way to parallelize a bucket sort: It would be sufficient to have m buckets and to assign one number to each processor. The processor that gets the ith number is labeled P_i, and it is responsible for placing the value i in the appropriate bucket. For example, if P_3 had the number 5, it would place the value 3 in bucket 5. The problem with this simplistic solution is that a memory conflict may result when several processors simultaneously attempt to store different values of i in the same bucket.

The memory contention problems can be solved by substantially increasing the memory requirements. Suppose that there is enough memory available for m arrays, each of size n. Each processor then can

write in a bucket without any fear of memory conflict. To complete the bucket sort, the m arrays must be merged. The processors perform this merge operation by searching, in a binary tree search method, for the marks of "buddy" active processors. If P_i and P_j discover each other's marks and $i < j$, then P_i continues and P_j deactivates (hence dropping a duplicate value).

Hirschberg also generalizes this algorithm so that duplicate numbers remain in the sorted array, but this generalization degrades the performance of the sorting algorithm. The result is a method that sorts n numbers with $n^{1+1/k}$ processors in time $O(k \log n)$ (where k is an arbitrary integer).

In addition to the lack of feasibility of the shared memory model, another major drawback of the parallel bucket sort is its $O(mn)$ space requirement. Even when the range of values is not very large, it would be desirable to reduce the space requirement; in the case of a wide range of values (e.g., when the sort keys are arbitrary character strings rather than integer numbers), the algorithm cannot be utilized.

3.4 Sorting by Enumeration

Parallel enumeration sorting algorithms, which do not restrict the range of the sort values and yet run in time $O(\log n)$, are described by Preparata [1978]. The keys are partitioned into subsets, and a partial count is computed for each key in its respective subset. Then, for each key the sum of these partial counts is computed in parallel, giving the rank of that key in the sorted sequence. Preparata's first algorithm uses Valiant's [1975] merging procedure, and sorts n numbers with $n \log n$ processors in time $O(\log n)$. The second algorithm uses Batcher's odd–even merge, and sorts n numbers with $n^{1+1/k}$ processors in time $O(k \log n)$. The performance of the latter algorithm is similar to Hirschberg's (Section 2.3), but has the advantage of being free of memory contention. Recall that Hirschberg's model required simultaneous fetches from the shared memory, whereas Preparata's method does not (since each key participates in only one comparison at any given unit of time).

3.5 Summary

Despite the improvement achieved by eliminating memory conflicts, the more recent shared memory algorithms are still far from being suitable for implementation. Any model requiring at least as many processors as the number of keys to be sorted, all sharing a very large common memory, is not feasible with present or near-term technology. These models also ignore significant computation overheads such as, for instance, the time associated with the reallocation of processors during various stages of the sort algorithm (although a first attempt at introducing this factor in a computation model is made by Vishkin [1981]).

However, the results achieved are of major theoretical importance, and the methods used demonstrate the intrinsic parallel nature of certain sorting procedures. It may also happen that future research will succeed in refining the shared memory model for parallel computation and make it more reasonable from a computer architecture point of view. An attempt to classify the various types of assumptions underlying recent research on shared memory models of parallel computation is made by Borodin and Hopcroft [1982]. Of particular interest is the class of algorithms that allow simultaneous reads, but allow simultaneous writes only if all processors try to write the same value [Shiloach and Vishkin 1981].

4. BLOCK SORTING ALGORITHMS

For all the parallel sorting algorithms described in previous sections, the number of records or keys to be sorted is limited by the number of processors available. Typically, $O(n)$ or more processors are utilized to sort n records. Thus these algorithms implicitly assume that the number of processors is very large.

This type of assumption was initially justified when parallel sorting algorithms were developed for implementing fast switching networks. In this context, there are two reasons that justify the n (or $n/2$) processors requirement to sort n numbers. First, in a switching network, the processors are

simple hardware boxes that compare and exchange their two inputs. Second, since the number of processors is proportional to the number of input lines to the network, it can never be prohibitively high.

However, for a general-purpose sorting algorithm, it is desirable to set a limit on the number of processors available so that the number of records than can be sorted will not be bounded by the number of processors. Furthermore, it must be possible to sort a large array with a relatively small number of processors. In general, research on parallel algorithms (for sorting, searching, and various numerical problems) is based on the assumption of unlimited parallelism. It is only recently that technology constraints, on one hand, and a better understanding of parallel algorithms, on the other, are motivating the development of algorithms for computers with a relatively small number of processors. An excellent illustration of this trend is a systematic study of *quotient networks* by Fishburn and Finkel [1982] for networks such as the perfect shuffle and the hypercube. Quotient networks are architectures that exploit limited parallelism in a very efficient way. The idea is that, given a network of p processing units, a problem of size n (for arbitrarily large n) can be solved by having each processing unit emulate a network of size $O(n/p)$ with the same topology. Together, the p processing units will emulate a network of size $O(n)$.

In the area of parallel sorting, the problem of limited parallelism has not been systematically addressed until recently. We propose some basic ideas for further research in this direction in the following paragraphs.

When p processors are available and n records are to be sorted, one possibility is to distribute the n records among the p processors so that a block of $M = \lceil n/p \rceil$ records is stored in each processor's local memory (a few dummy records may have to be added to constitute the last block). The processors are labeled $P_1, P_2, \ldots, P_p$, according to an indexing rule that is usually dictated by the topology of the interconnecting network. Then, the processors cooperate to redistribute the records so that

(1) the block residing in each processor's memory constitutes a sorted sequence S_i of length M, and
(2) the concatenation of these local sequences, $S_1, S_2, \ldots, S_p$, constitutes a sorted sequence of length n.

For example, for three processors, the distribution of the sort keys before and after sorting could be the following:

	Before	After
P_1	2, 7, 3	1, 2, 3
P_2	4, 9, 1	4. 5, 6
P_3	6, 5, 8	7, 8, 9

Thus we now have a convention for ordering the total address space of a multiprocessor, and we have defined parallel sorting of an array of size n, where n may be much larger than p.

Algorithms to sort large arrays of files that are initially distributed across the processors' local memories can be constructed as a sequence of block *merge–split steps*. During a merge–split step, a processor merges two sorted blocks of equal length (which were produced by a previous step), and splits the resulting block into a "higher" and a "lower" block, which are sent to two destination processors (like the high and low outputs in a comparison–exchange step).

A *block sorting algorithm* is obtained by replacing every comparison–exchange step (in a sorting algorithm that conists of comparison–exchange steps) by a merge–split step. It is easy to verify that this procedure produces a sequence that is sorted according to the above definition.

There are two ways to perform a merge–split step. One is based on a two-way merge [Baudet and Stevenson 1978]; the other is based on a bitonic merge [Hsaio and Menon 1980]. In Sections 4.1 and 4.2, we describe both methods and illustrate them by investigating the block sorting algorithms that they generate on the basis of the odd–even transposition sort (Section 1.1) and the bitonic sort (Section 2.1.2). An important property of the parallel block sorting algorithms generated by both methods is that, like the network sorting algorithms, they can be executed in SIMD mode (see Section 2.2).

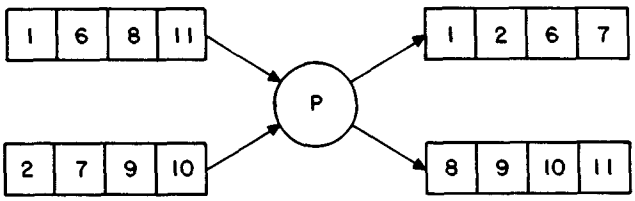


Figure 10. Merge–split based on two-way merge.

4.1 Two-Way Merge–Split

A two-way merge–split step is defined as a two-way merge of two sorted blocks of size M, followed by splitting the result block of size $2M$ into two halves. Both operations are executed within a processor's local memory. The contents of processor's memory before and after a two-way merge-split are shown in Figure 10. After two sorted sequences of length M have been stored in each processor's local memory, the processors execute in parallel a merge procedure and fill up an output buffer $O[1..2M]$ (thus a two-way merge–split step uses a local memory of size at least $4M$). After all processors have completed the parallel execution of the merge procedure, they split their output buffer and send each half to a destination processor. The destination processors' addresses are determined by the comparison–exchange algorithm on which the block sorting algorithm is based.

4.1.1 Block Odd–Even Sort Based on Two-Way Merge–Split

Initially, each of the p processors' local memory contains a sequence of length M. The algorithm consists of a preprocessing step (Step 0), during which each processor independently sorts the sequence residing in its local memory, and p additional steps (Steps 1 to p), during which the processors cooperate to merge the p sequences generated by Step 0. During Step 0, the processors perform a local sort by using any fast serial sorting algorithm. For example, a local two-way merge or a quick sort can be used. Steps 1 to p are similar to Steps 1 to p of the odd–even transposition sort (see Section 1.1). During the odd (even) steps, the odd- (even-) numbered processors receive from their right neighbor a sorted block, perform a two-way merge, and send back the higher M records. The algorithm can be executed synchronously by p processors, odd and even processors being alternately idle.

4.1.2 Block Bitonic Sort Based on Two-Way Merge–Split

By using Batcher's bitonic, p records can be sorted with $p/2$ processors in $\log^2 p$ shuffle steps and $1/2((\log p) + 1)(\log p)$ comparison–exchange steps. Suppose that each processor has a local memory large enough to store $4M$ records. In this case, a processor can perform a two-way merge split on two blocks of size M. By replacing each comparison–exchange step by a two-way merge–split step, we obtain a block bitonic sort algorithm that can sort $M \cdot p$ records with $p/2$ processors in $\log^2 p$ shuffle steps, and $1/2((\log p) + 1)(\log p)$ merge–split steps. During a shuffle step, each processor sends to each of its neighbors a sorted sequence of length M. During a merge–split step, each processor performs a two-way merge of the two sequences of length M (which it has received during the previous shuffle step) and splits the resulting sequence into two sequences of length M. The algorithm is illustrated in Figure 11, for two processors, where $M = 2$.

In the general case, the algorithm requires $p/2$ processors, where p is a power of 2, each with a local memory of size $4(M \cdot p)$, to sort $M \cdot p$ records.

4.1.3 Processor Synchronization

When M is large, or when the individual records are long, transferring blocks of $M \cdot p$ records between the processors introduces time delays that are higher by several orders of magnitude than the instruction rate of the individual processors. In addition, depending on the data distribution,

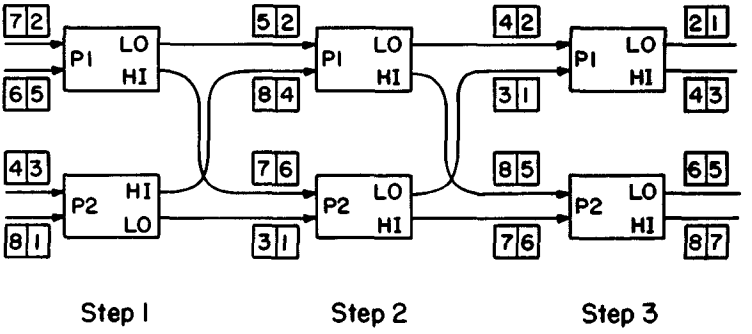


Figure 11. Block-bitonic sort based on two-way merge.

the number of comparisons required to merge two blocks of M records may vary. Thus, for the execution of block sorting algorithms based on two-way merge–split, a coarser granularity for processor synchronization might be more adequate than the SIMD mode, where processors are synchronized at the machine instruction level. A multiprocessor model for those algorithms in which processors operate independently of each other, but can be synchronized by exchanging messages among themselves or with a controlling processor at intervals of several thousand instructions, is more appropriate for these algorithms. At the initiation time of a block sorting algorithm, the controller assigns a number of processors to its execution. Because other operations may already be executing, the controller maintains a free list and assigns processors from this list. In addition to the availability of processors, the size of the sorting problem is also considered by the controller to determine optimal processor allocation.

4.2 Bitonic Merge–Exchange

Consider the situation in which two processors P_i and P_j each contain a sorted block of length M, and we want to compare and exchange records between the processors so that the lower M records reside in P_i and the higher M in P_j. One way to obtain this result is to execute the following three steps:

(1) P_j sends its block to P_i.
(2) P_i performs a two-way merge–split.
(3) P_i sends the high half-block to P_j.

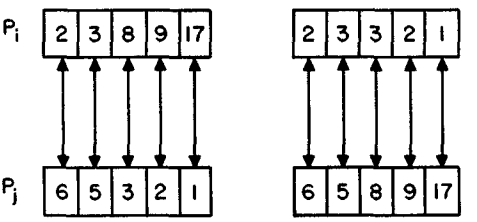


Figure 12. Bitonic merge–exchange step.

However, as indicated in the previous section, the two-way merge–split requires a processor's local memory size to be at least $4M$. Another alternative is that P_j send one of its records at a time and wait for a return record from P_i before sending the next record. Suppose that M records $(x_1, x_2, \ldots, x_M)$ are stored in increasing order in P_i's memory and the M records $(y_1, y_2, \ldots, y_M)$ are stored in decreasing order in P_j's memory. Let P_j send y_1 to P_i. P_i then compares x_1 and y_1, keeps the lower of the two, and sends back to P_j the higher record. This procedure is then repeated for $(x_2, y_2), \ldots, (x_M, y_M)$. This sequence of comparison–exchanges constitutes the "bitonic merge" and results in having the highest M records in P_j and the lowest M in P_i [Alekseyev 1969; Knuth 1973]. Thus the merge–split operation can now be completed by having P_i and P_j each perform a local sort of their M records. Figure 12 illustrates the bitonic merge–exchange operation for $M = 5$. It is important to notice that the data exchanges are synchronous (unlike in the two-way merge–split operation). Thus the block sorting algorithms based on the bitonic

merge–exchange are more suitable for implementation on parallel computers that require a high degree of synchronization between their processors.

The bitonic merge–exchange also requires substantially less buffer space than the two-way merge–split. Because the two-way merge–split merges two blocks of size M within a processor's local memory, it uses $(4 \cdot M)$ space. The bitonic merge–exchange requires space for only $M + 1$ records. Finally, the comparisons (of pairs of records) and the transfers are interleaved in every bitonic merge–exchange step. The two-way merge–split requires that an entire block of data be transferred to a processor's memory before the merge operation is initiated, whereas the bitonic merge–exchange can overlap each record's transfer time with processing time.

However, a major disadvantage of the bitonic merge–exchange is the necessity for performing a local sort of M records in each processor after the exchange step is completed. To perform the local sort, a serial sorting algorithm that permutes the records in place (such as heap sort) should be used. Otherwise the local sort might require more memory than the exchange. Note that the sequences generated by the bitonic exchange are bitonic. Thus sorting these sequences requires at most $(M/2) \log M$ comparisons and local moves.

4.2.1 Block Odd–Even Sort Based on Bitonic Merge–Exchange

As with the block odd–even merge based on two-way merge (Section 4.1.1), we start with M records in each processor's memory and perform an initial phase where each processor independently sorts the sequence in its memory. However, Steps $1 \cdots p$ are different. During odd (even) steps, odd-(even-) numbered processors perform a bitonic merge–exchange with their right neighbor. Figure 13 illustrates this algorithm for $p = 4$, where $M = 5$.

4.2.2 Block Bitonic Sort Based on Bitonic Merge–Exchange

A fast and space-efficient block sorting algorithm can be derived from Stone's version of the bitonic sort, which was described in Section 2.1.2. Consider a network of p identical processors, where p is a power of 2, interconnected by two types of links (Figure 14):

(1) two-way links, between pairs of adjacent processors: $P_0P_1, P_2P_3, \ldots$;
(2) one-way shuffle links, connecting each P_i to its shuffle processor.

If each processor has a local memory of size $M + 1$, then $M \cdot p$ records can be sorted by alternating local-sort, block-bitonic exchanges between neighbor processors and shuffle procedures. During a shuffle procedure, each processor sends the records that were in its memory, in order, to the corresponding location of the shuffle processor's memory and receives the records that were in the memory of the reverse shuffle processor. Figure 15 illustrates this algorithm for $p = 4$, where $M = 5$.

5. EXTERNAL PARALLEL SORTING

In this section, we address the problem of sorting a large file in parallel. Serial file sorting algorithms are often referred to as "external sorting algorithms," as opposed to array sorting algorithms that are "internal." For a conventional computer system the need for an external sorting algorithm arises when the file to be sorted is too large to fit in main memory.

Thus for a single processor the distinction between internal sorting and external sorting methods is well known, and there are accepted criteria for measuring their respective performances. However, the topic of external parallel sorting has not yet received adequate consideration.

In Section 4, we presented a number of parallel algorithms that can sort an array that is initially distributed across the processors' memories. The size of the array was limited only by the total memory of the system (considered as the concatenation of the processors' local memories). By analogy to the definition of serial internal sorting, these algorithms may be called "parallel internal sorting algorithms."

A parallel sorting algorithm is defined as a *parallel external sorting* algorithm if it can

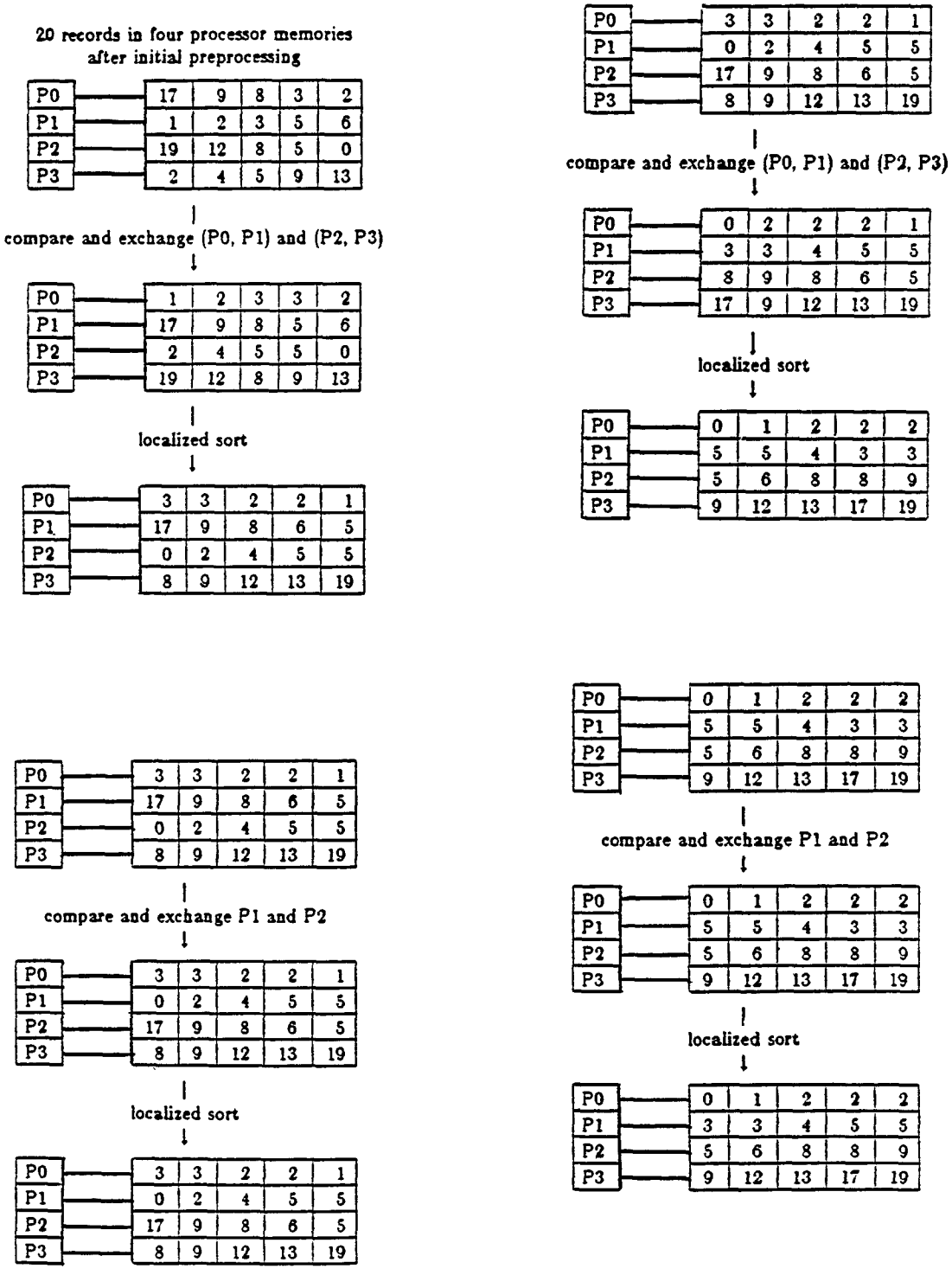


Figure 13. Block odd–even sort (20 records).

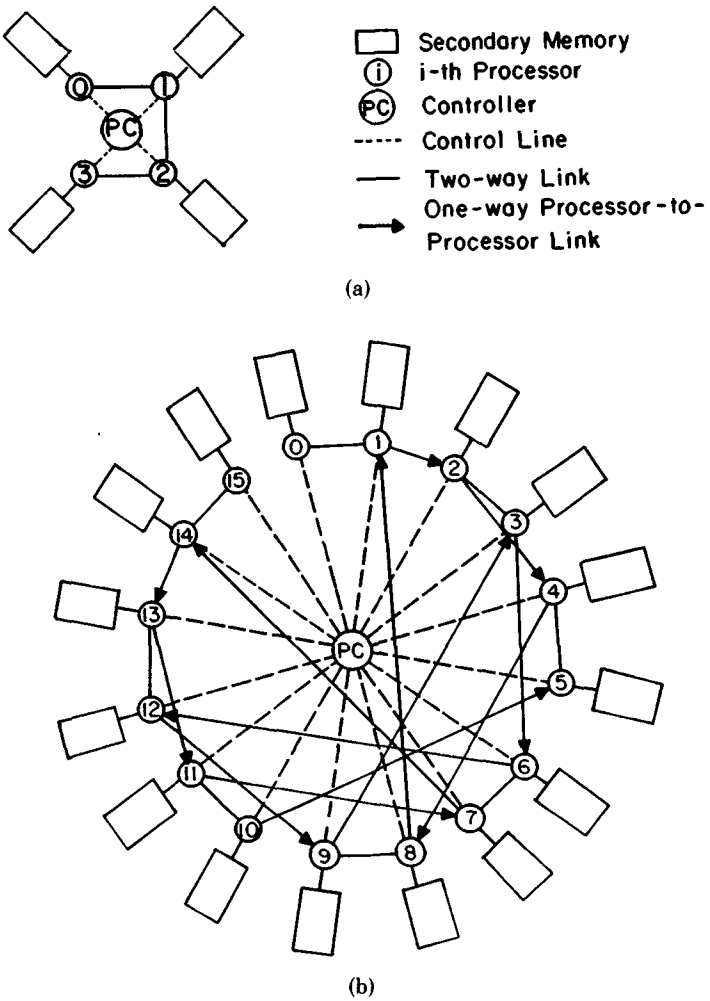


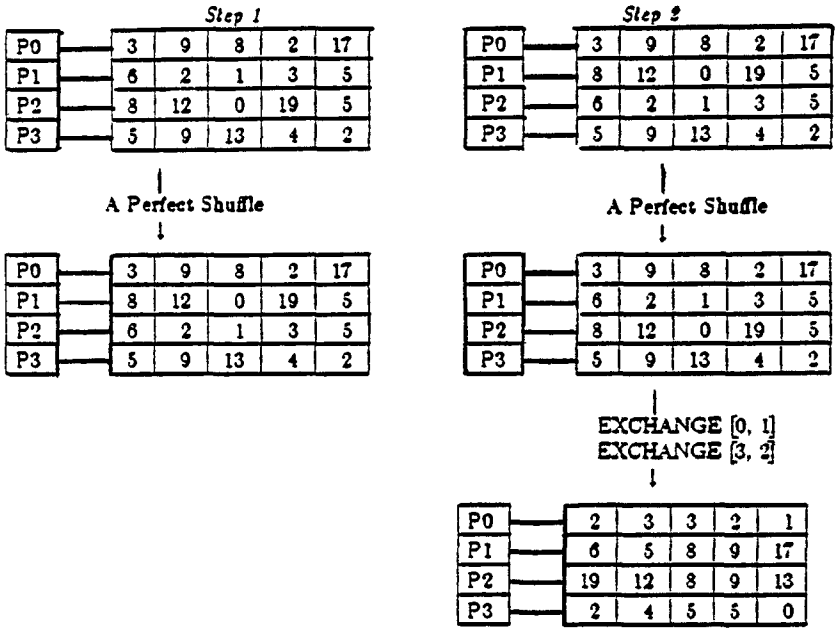
Figure 14. Processors' interconnection for block-bitonic sort. (a) $p = 4$. (b) $p = 16$.

sort a collection of elements that is too large to fit in the total memory available in the multiprocessor. This definition is general enough to apply to both categories of parallel architectures: the *shared memory* multiprocessors and the *loosely coupled* multiprocessors (also called "multicomputers").

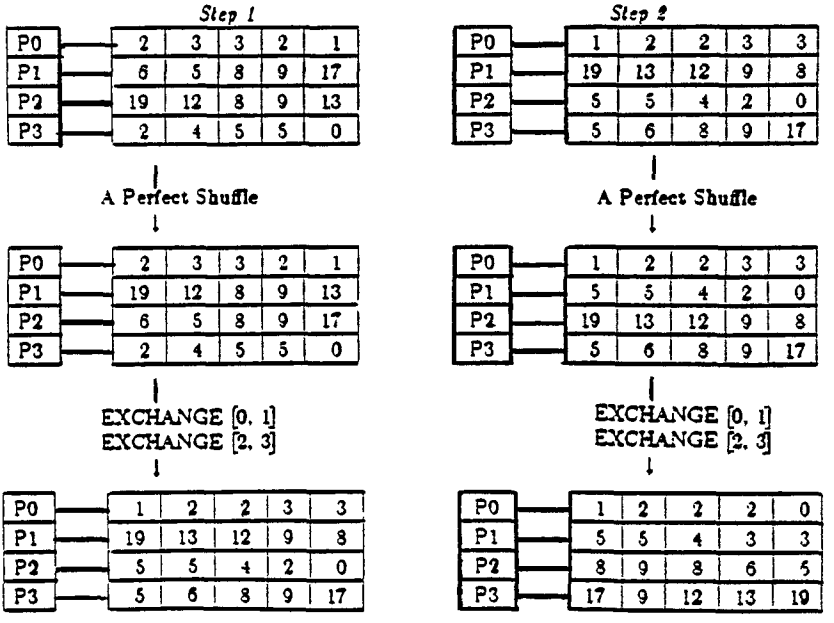
For shared memory multiprocessors, an external sorting algorithm is required when the shared memory is not large enough to hold all the elements (and some work space to execute the sort program). On the other hand, for loosely coupled multiprocessors, the assumption is that the source records cannot be distributed across the processors' local memories. That is, the multicomputer has p identical processors and each processor's memory is large enough to hold k records, but the source file has more than $p \cdot k$ records. In both cases, the processor can access a mass-storage device on which the file resides. At termination of the algorithm, the file must be written back to the mass-storage device in sorted order.[3]

An early result on tape parallel sorting appeared in Even [1974]. Recently in Bitton [1982], several parallel sorting algorithms have been proposed for files residing on a modified moving-head disk.[3]

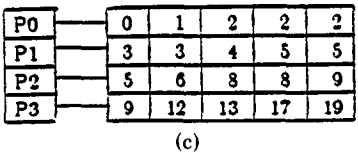
[3] Physical order on the mass storage device must be defined according to the physical characteristics of the storage device. For example, for a magnetic disk, a track-numbering convention must be agreed upon.



(a)



(b)



(c)

Figure 15. Block-bitonic sort. (a) Stage 1. (b) Stage 2. (c) Stage 3.

5.1 Parallel Tape Sorting

The sorting problem addressed by Even [1974] is to sort a file of n records with p processors (where n is much larger than p) and $4p$ magnetic tapes. The only internal memory requirement is that three records could fit simultaneously in each processor's local memory. Under those assumptions, Even proposes two methods for parallelizing the serial two-way external merge sort algorithm. In the first method, all the processors start together and work independently of each other on separate partitions of the file. In the second, processors are added one at a time to perform sorting in a pipelinelike algorithm. Both methods can be described briefly:

Method 1. Each processor is assigned n/p records and four tapes and performs a (serial) external merge sort on this subset. After p sorted runs have been produced by this parallel phase, during a second phase a single processor merge sorts these runs serially.

Method 2. The basic idea is that each processor performs a different phase of the serial merge procedure. The ith processor merges pairs of runs of size 2^{i-1} into runs of size 2^i. Ideally, n is a power of 2 and log n processors are available. A high degree of parallelism is achieved by using the output tapes of a processor as input tapes for the next processor, so that, as soon as a processor has written two runs, these runs can be read and merged by another processor. In order to overlap the output time of a processor with the input time of its successor, each processor writes alternately on four tapes (one output run on each tape).

These methods show that, from the algorithmic point of view, it is possible to introduce a high degree of parallelism in the conventional two-way external merge–sort. However, the assumptions about the mass storage device do not consider constraints imposed by technology. Like the shared memory model for array sorting, a parallel file sorting model that assumes a shared mass storage device with unlimited I/O bandwidth (e.g., a model with p processors and $4p$ magnetic tape drives) provides very limited insight into practical aspects of implementation.

5.2 Parallel Disk Sorting

The notion of a sorted file stored on a magnetic disk requires that physical order be defined since disks are not sequential storage media. Within a disk track records are stored sequentially, but then a convention is needed for numbering tracks. For example, adjacent tracks could be defined as consecutive tracks on one disk surface. This convention is adequate if a separate processor is associated with each disk surface. Another way to model the mass-storage device is to consider a modified moving-head disk that provides for parallel read/write of tracks on the same cylinder (Figure 17). Disks that provide this capability have been proposed [Banerjee et al. 1978] and, in some cases, already built. The idea was pioneered by database machine designers, and prototypes have been built in the framework of database research projects (see, e.g., Leilich et al. [1978]). Commercial parallel readout disks have recently been made available for high-performance computers (e.g., a 600-Mbyte drive with a four-track parallel readout capability and a data transfer rate of 4.84 Mbytes/second is now available for the Cray-1 computer). Thus parallel readout disks appear to constitute a viable compromise between the cost-effective, conventional moving-head disk and the obsolete fixed-head disk.

In order to minimize seek time, two disk drives can be used concurrently. During execution of a single phase of a sorting algorithm, one drive can be utilized for reading and the other for writing.

In Bitton-Friedland [1982] a number of parallel external sorting algorithms and architectures are examined and analyzed. The mass-storage device is modeled as a parallel read/write disk. The algorithm with the best performance is a parallel two-way external merge–sort, termed the *parallel binary merge* algorithm. It is an improved variation of Method 1 in Section 5.1, achieved by parallelizing the second phase of this method.

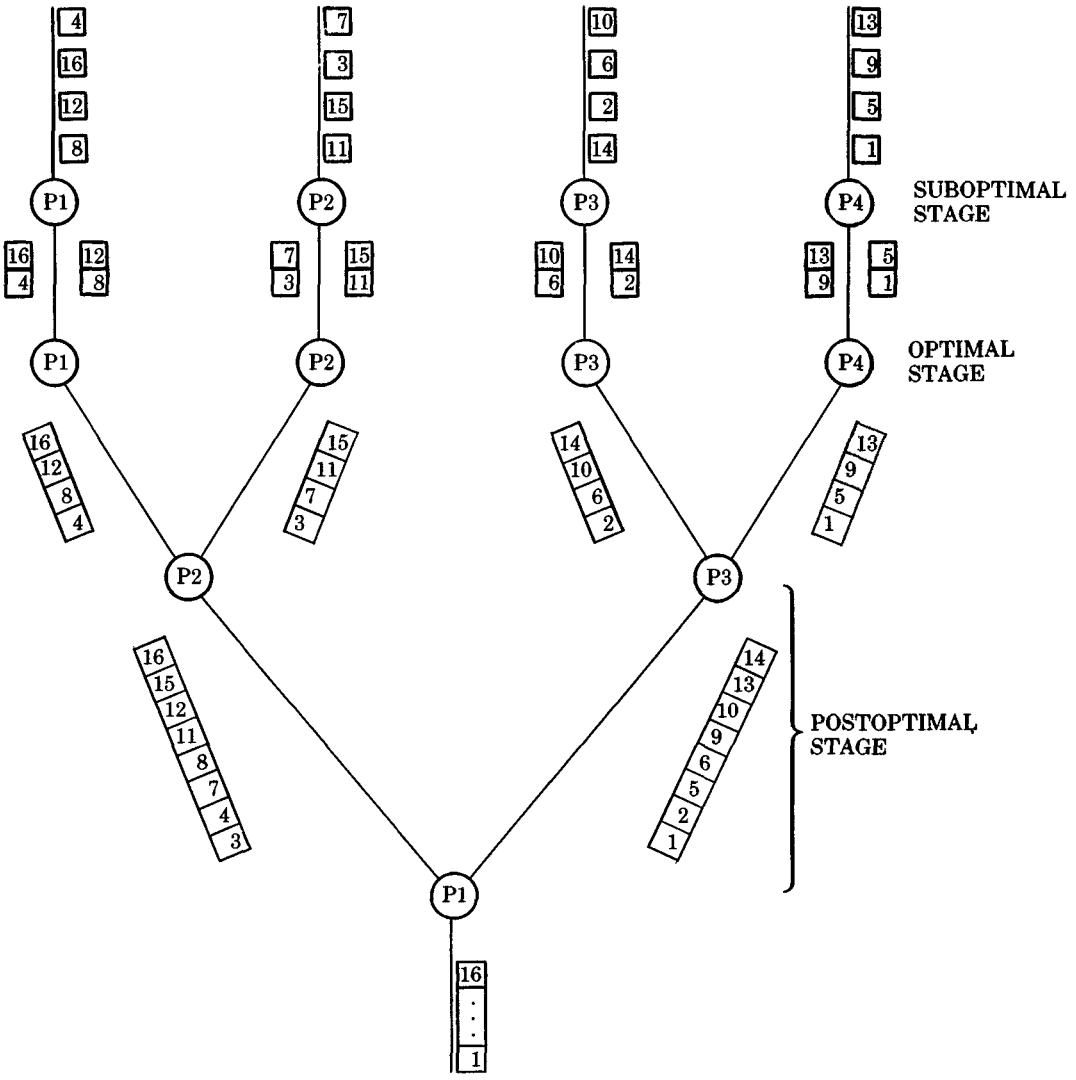


Figure 16. Parallel binary merge sort.

When the number of output runs is 2^k, and $k > 1$, 2^{k-1} processors can be used to perform the next step of the merge sort concurrently. Thus execution of the parallel binary merge algorithm can be divided into three stages, as shown in Figure 16. The algorithm begins execution in a suboptimal stage (similar to Phase 1 in Method 1), in which sorting is done by successively merging pairs of longer runs until the number of runs is equal to twice the number of processors. During the suboptimal stage, the processors operate in parallel, but on separate data. Parallel I/O is made possible by associating each processor with a surface of the read disk and a surface of the write disk.

When the number of runs equals $2 \cdot p$, each processor will merge exactly two runs of length $N/2p$. We term this stage the optimal stage. During the postoptimal stage, parallelism is employed in two ways. First, 2^{k-1} processors are utilized to concurrently merge 2^{k-1} pairs of runs (this occurs

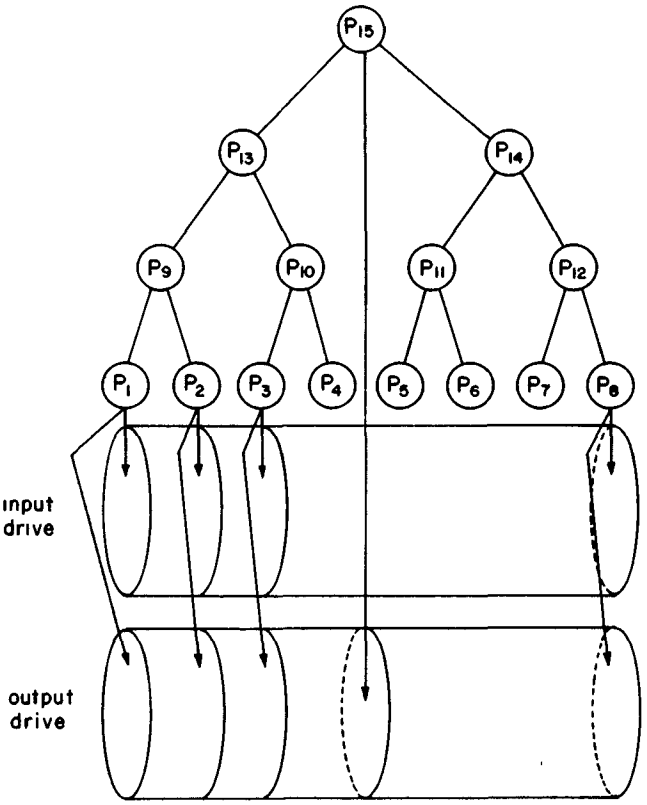


Figure 17. Architecture for the parallel binary merge sort.

after $\log(m/k)$ merge steps). Second, pipelining is used between merge steps. That is, the ith merge step starts as soon as Step $(i-1)$ has produced one unit of each of two output runs (where a unit can be a single record or an entire disk page).

The ideal architecture for the execution of this algorithm is a binary tree of processors, as shown in Figure 17. The mass-storage device consists of two drives, and each leaf processor is associated with a surface on both drives. In addition to the leaf processors, the disk is also accessed by the root processor to write the output file. This organization permits leaf processors to do input/output in parallel, while reducing almost by half the number of processors that must actually do input/output.

5.3 Analysis of the Parallel External Sorting Algorithm

For serial external sorting, numerous empirical studies have been done on real computers and real data in order to evaluate the performance of external sorting algorithms. The results of these studies have complemented analytical results when modeling analytically the effect of access time and the impact of data distribution was too complex. In a parallel environment, the analytical performance evaluation of an external sorting scheme is made even more difficult by the complexity of the input/output device.

We can get some indication of the parallel speedup that can be achieved by performing an external sort in parallel by assuming that the available input/output bandwidth is limited only by the number of processors. However, a satisfactory analysis of parallel external sorting algorithms must also consider the constraints imposed by mass-storage technology. For example, if the modified disk described in Section 5.2 is used for storage, the suboptimal stage for the parallel binary merge algorithm can either be highly parallel, or almost sequen-

tial, depending on whether or not the processors request data from several tracks on the same cylinder.

6. HARDWARE SORTERS

The high cost and frequent need of sorting are motivating the design of "sort engines," which could eventually off-load the sorting function from general-purpose central processing units (CPUs). By implementing the sequence of comparison and move steps required by an efficient sorting algorithm in hardware, one could realize a low-cost, fast, hardware device that would significantly lighten the burden on the CPU. Several alternative designs of hardware sorters recently have been proposed [Chen et al. 1978; Chung et al. 1980; Dohi et al. 1982; Lee et al. 1981; Thompson 1983; Yasuura et al. 1982], and preliminary evaluations seem to indicate that a VLSI implementation of sorting devices could soon become feasible. The relatively simple logic required for sorting constitutes a strong argument in favor of this approach. In addition, the advent of new and inexpensive shift-register technologies, such as charge-coupled devices and bubble memories, is stimulating new designs of hardware sorters based on these technologies [Chung et al. 1980; Lee et al. 1981].

A future outcome of improvement in technology might be that bubble chips could provide storage for large files with on-chip sorting capabilities. In this case, the sorting function could be provided by the mass-storage devices without requiring the transfer of files to a dedicated sorting machine or the main memory of a general-purpose computer. However, it is premature at this point to determine whether or not advances in technology will be able to provide for intelligent mass-storage devices with sorting capabilities.

Hardware sorters, in particular VLSI sorting circuits, are at present the focus of active research. Theoretical problems related to area–time complexity are also drawing considerable attention to VLSI sorting [Leiserson 1981; Thompson 1980, 1983]. It is beyond the scope of this paper to present a proper survey of the theoretical bounds obtained for chip area and time complexity of VLSI sorters. These results pertain to an area of research in complexity theory that is being defined, and they cannot be presented without properly defining VLSI circuit areas and introducing the theoretical area * time^2 trade-off.[4]

In the remainder of this section, we present an overview of designs that have been proposed for hardware sorters. We have chosen to concentrate on sorters that were originally conceived for the magnetic bubble technology because they illustrate well how technology constraints define trade-offs between sorting speed and parameters related to input/output bandwidth, memory, and number of control lines required by on-chip sorters. In particular, we shall describe the *rebound sorter* and the *up–down sorter* (which are clever pipeline versions of the odd–even transposition sort (Section 1.1)) and a number of magnetic bubble sorters that integrate a sorting capability in bubble memory.

6.1 The Rebound Sorter

The *uniform ladder* [Chen 1976] is an N-loop shift-register structure capable of storing N records, one record to a loop. Since records stored in adjacent loops can be exchanged, this storage structure is very suitable for a hardware implementation of the odd–even transposition sort (see Section 1.1). If the time for a bit to circulate within a loop is called a *period*, then N records (which have been previously stored in the ladder) can be sorted in $(N + 1)/2$ periods using $(N - 1)$ comparators.

Further investigation of the ladder structure led to the design of a new sorting scheme, where input/output of the records can be completely overlapped with the sorting time. This scheme is the *rebound sort*

[4] In a recent work (published after this paper was written), Thompson has surveyed 13 different VLSI circuits that implement a range of sorting schemes: heap sort, pipelined merge-sort, bitonic sort, bubble sort, and sort by enumeration. For each of these schemes one or several circuit topologies (linear array, mesh, binary tree, shuffle-exchange and cube-connected cycles, mesh of trees) are considered, and the resulting sorter is evaluated with respect to its area * time^2 complexity.

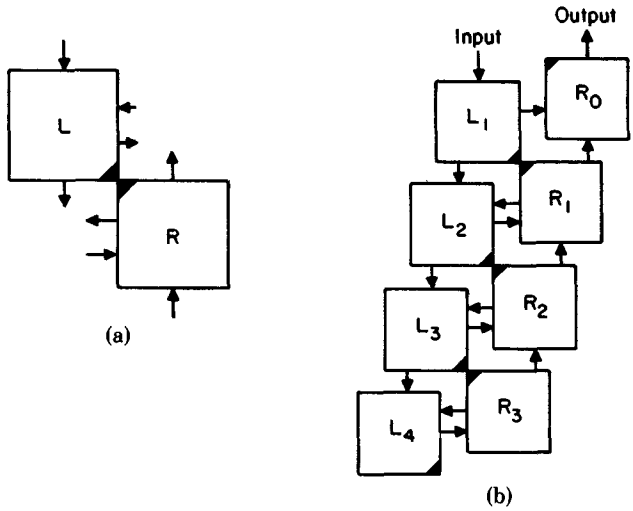


Figure 18. The rebound sorter. (a) The steering unit. (b) Stacking steering units. [From Chen et al. 1978. © IEEE 1978.]

[Chen et al. 1978]. The basic building block of the rebound sorter is the steering unit (Figure 18a), which has an upper-left cell L and a lower-right cell R. A sorter for N records is assembled by stacking $(N - 1)$ steering units, plus a bottom cell and a top cell (Figure 18b). Associated with the two cells in a steering unit is a comparator K, which can compare the two values stored in the upper-left and lower-right cells. A record may be stored across adjacent cells in two steering units, but the sorting key (assumed to be at the head of the record) must fit entirely in one cell, so that two keys can be compared in a single steering unit. Sorting is performed by pipelining input records through the stack of steering units. Records enter the sorter through the upper-left cell of the top steering unit and emerge in sorted order from the top cell (at the upper-right corner of the stack) after $2N$ steps. The sorting scheme is illustrated in Figure 19 for $N = 4$. The sorter alternates between a *decision state* and a *continuing state*. In the decision state, each steering unit compares the keys stored in its upper-left and lower-right cells and emits the keys either horizontally (upper-left key to the right, lower-right key to the left) or vertically (upper-left key to the upper unit, lower-right key to the lower unit), depending on the outcome of the comparison. In the continuing state, each steering unit continues to emit its contents in the direction determined in the previous decision step. The continuing steps are required to append the body of the records to their key. It is readily seen that the first key emerges from the sorter after N steps ($t_0 + 8$ in Figure 9) and the complete sorted sequence is produced in the next N steps.

6.2 The Up–Down Sorter

A significant improvement of the ladder sorter can be achieved by incorporating the comparison function in the basic steering unit and using an "up–down" sorting scheme instead of the rebound sort. To sort $2N$ keys, the "compare/steer bubble sorter" [Lee et al. 1981] requires N compare/steer units stacked on top of each other. It is assumed that the entire record fits in a cell (thus two records are stored in a compare/steer unit). The up–down sort is illustrated in Figure 20 for $N = 3$ (three compare/steer units sorting six records). The up–down sorter operates in two phases. During the downward input phase, $2N$ keys are loaded in $2N$ periods. During each period of the input phase, a key enters the sorter and all units push down the larger of their two keys (to the unit beneath them). During the upward output phase, each unit pops

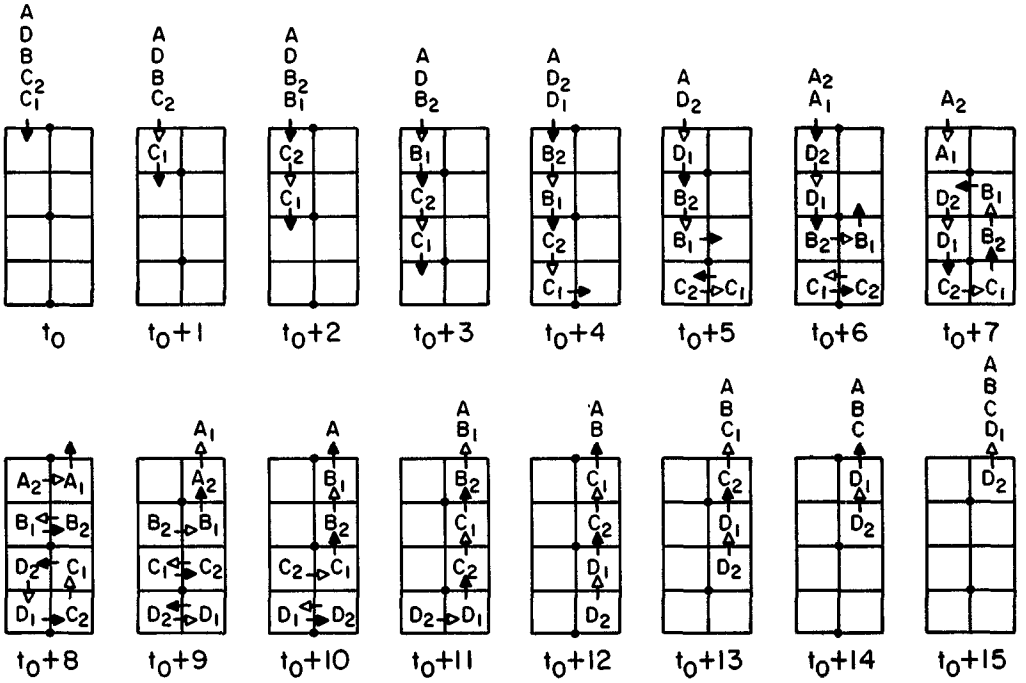


Figure 19. The rebound sort for four records. [From Chen et al. 1978. © IEEE 1978.]

up the smaller of its two keys (to the unit above it), and a key is output in every period.

The up–down sorter eliminates the large number of control lines required by the rebound sorter. Whereas the rebound sorter needs multiple control lines to individually activate switches of the bubble ladder, the up–down sorter can be implemented with a single control line for resetting all the compare/steer units at the beginning of each phase. Thus on-chip compare/steer units have a better chance to provide a chip implementation of large files sorters.

Both the rebound sorter and the up–down sorter have the very desirable property of completely overlapping input/output time with sorting time. Thus, assuming that only serial input/output of data records is available, they provide an optimal hardware implementation of file sorting.

6.3 Sorting within Bubble Memory

The sorter described in the previous section incorporates the comparison function in the design of the bubble chip. Thus this type of chip constitutes an intelligent memory capable of performing the logical operations required by sorting. The sorters proposed by Chung et al. [1980] also attach comparators to the bubble memory, but in addition they eliminate the input/output function that is an intrinsic part of the previous sorting algorithms. The motivation for designing bubble elements that sort *in situ* stems from the assumption that magnetic bubble memory may soon provide cost-effective mass-storage systems. If advances in technology make this assumption realistic, then sorting a file will only require rearranging records in mass storage according to the result of comparisons performed within the memory, without input/output operations or CPU intervention.

Four models of intelligent bubbles are considered by Chung et al. [1980], and for each model an alternative sorting scheme is proposed. The models differ in the size of the bubble loops and the number of switches required between the loops (to perform comparisons). The first two

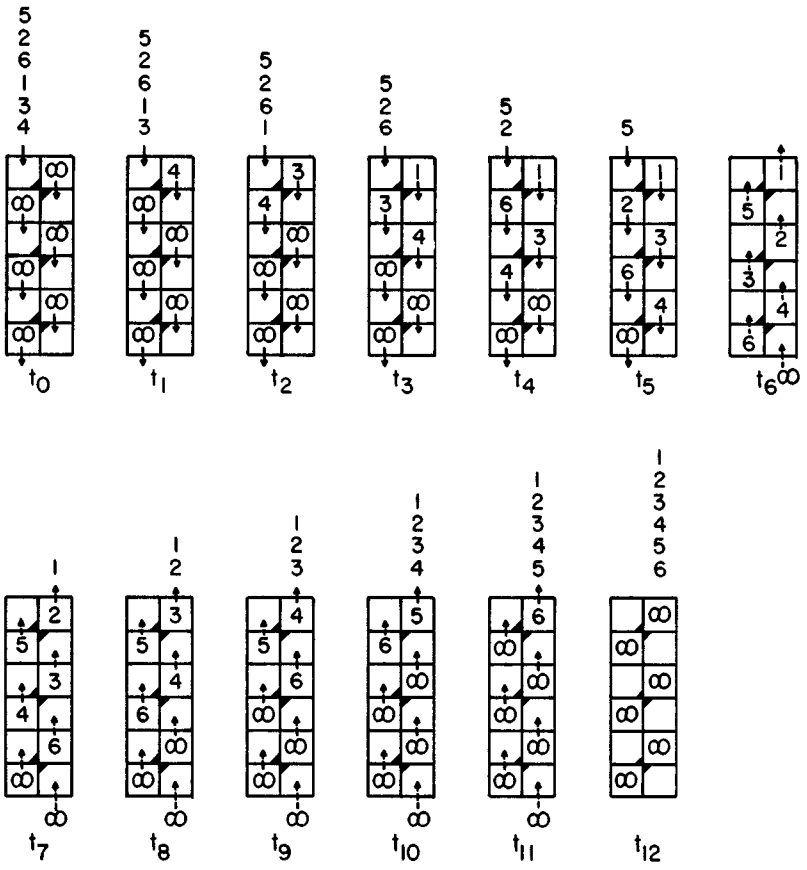


Figure 20. Operation of the up-down sorter for six records (only the keys are shown). [From Lee et al. 1981 © IEEE 1981]

models implement a bubble sort and an odd–even transposition sort, respectively, whereas the other two implement a bitonic sort. The first model has two loops, one of size $(n - 1)$ and the other of size 1, and a single switch between them (Figure 21a) is used to perform the bubble sort. The second model is a linear array of loops, all of size 1, with a switch between every pair of adjacent loops (Figure 21b). The $(n - 1)$ switches perform comparisons in parallel, according to the odd–even transposition scheme (Section 1.1). For the other two models (Figure 21c, d), the basic idea is to have the option to open a switch between adjacent loops (that are of the same size in Model 3, but of different sizes in Model 4) so that the two loops are collapsed into a larger loop. At every step of the bitonic sort, larger loops are formed that contain bitonic sequences. Because they implement a faster algorithm, these sorters are faster than the first sorter. However, the trade-off is a higher complexity of hardware (more switches and more control states per switch), which may be beyond the present limits of chip density. For example, the bubble-sort sorter sorts in $O(n^2)$ comparison steps, but it requires only one simple switch (with three control states). On the other hand, a bitonic sorter sorts in time $O(n \log^2 n)$, but requires $\log n$ complex switches (each with $3 \log n$ control states). Thus these detailed designs of bubble sorters provide an excellent illustration of cost–performance trade-offs in sorting.

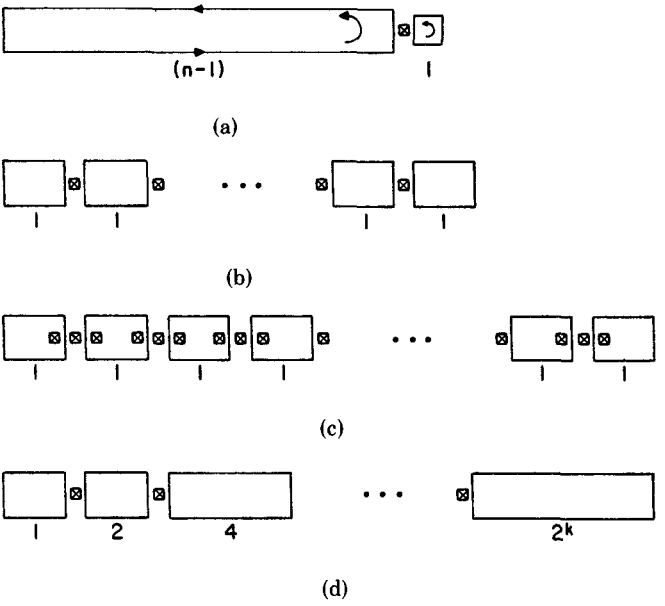


Figure 21. Bubble loop structures for sorting. (a) Model 1. (b) Model 2. (c) Model 3. (d) Model 4. [From Chung et al. 1980. © IEEE 1980.]

6.4 Summary and Recent Results

Several designs of hardware sorters have been proposed recently, and preliminary evaluations of their feasibility are being performed. One of the most promising approaches appears to be the implementation of a simple pipelined sorting scheme on bubble chips.

However, a number of alternative designs are being investigated. More recently, two detailed layouts of VLSI sorters have been proposed. A high-capacity cellular array that sorts by enumeration is investigated by Yasuura et al. [1982]. In Dohi et al. [1982], cells that are able to sort-merge data in compressed form are connected in a binary tree topology to constitute a powerful sorter. In both cases, the design of the basic cell is simple enough to allow very high density packaging with current or near-future technology.

The parallel sorting algorithms used by currently proposed hardware sorters are simple and slow compared to either the sorting networks or the shared memory model algorithms. Although theoretical complexity bounds are being investigated for faster VLSI sorters and important results have been achieved [Bilardi and Preparata 1983; Thompson 1983], the feasibility of fast high-capacity VLSI sorters is still an open question. However, this direction of research on parallel sorting appears to be very promising. A well-defined VLSI complexity model that combines some measures of hardware complexity with time efficiency provides a systematic approach to the analysis of parallel sorting algorithms. For bubble memory devices, when assuming that records are read and written serially, it has been shown that input/output time can be effectively overlapped with sorting time. Thus advances in technology may soon make a well-designed, dedicated sorting device a cost-effective addition to many computer systems.

7. CONCLUSIONS AND OPEN PROBLEMS

Over the last decade, parallel sorting has been the focus of active research. There are many parallel sorting algorithms currently known, and new algorithms are being developed—ranging from network sorting algorithms to algorithms for hypothetical shared memory parallel computers or VLSI chips. Research on parallel sorting has offered many challenges to both theoreticians and systems designers. From a theoretical point of view, the main research problem has been to design algorithms that, by sys-

tematically exploiting the intrinsic parallelism in sorting and merging, would reach the time theoretical lower bound, that is, algorithms that would sort n numbers in time $O(\log n)$ on a hypothetical $O(n)$-processor parallel machine. From a practical point of view, systems designers have investigated feasibility with current or near-term technology, and integration of input/output time in the cost of parallel sorting.

Despite the apparent disparity among the numerous parallel sorting algorithms that have been proposed, we have shown how these algorithms may be broadly classified into three categories: network sorting algorithms, shared memory sorting algorithms, and parallel file sorting algorithms. The first category includes algorithms that are based on nonadaptive, iterative merging rules. Although first proposed in the context of sorting networks, the two fundamental parallel merging algorithms (the odd–even merge and the bitonic merge described in Section 2.1) were subsequently embedded in a more general model of parallel computation, where processors exchange data synchronously along the lines of a sparse interconnection network. In particular, the bitonic sort has been adapted for mesh-connected processors (Section 2.2.1) and for a number of networks such as the shuffle, the cube, and the cube-connected cycles.

Algorithms in the second category require a more flexible pattern of memory accesses than the network sorting algorithms. They assume shared memory models of computation, where processors share read and write access to a very large memory pool with various degrees of contention and different policies of conflict resolution. For the most part, shared memory parallel sorting algorithms are faster than the network sorting algorithms, but they are far less feasible from a hardware point of view. In Table 1, we briefly summarize the asymptotic bounds of the main algorithms in both the network and the shared memory categories in terms of the number of processors utilized and execution time (the latter being estimated as the number of parallel comparison steps required by the algorithms).

Table 1. Number of Processors and Execution Time Required by Parallel Sorting Algorithms

Algorithm	Processors	Time
Odd–even transposition	n	$O(n)$
Batcher's bitonic	$O(n \log^2 n)$	$O(\log^2 n)$
Stone's bitonic	$n/2$	$O(\log^2 n)$
Mesh–bitonic	n	$O(\sqrt{n})$
Muller–Preparata	n^2	$O(\log n)$
Hirschberg (1)	n	$O(\log n)$
Hirschberg (2)	$n^{1+1/k}$	$O(k \log n)$
Preparata (1)	$n \log n$	$O(\log n)$
Preparata (2)	$n^{1+1/k}$	$O(k \log n)$
Ajtai et al	$n \log n$	$O(\log n)$

In the third category of parallel sorting algorithms, we include both internal and external parallel sorting algorithms that utilize limited parallelism to solve a large-sized problem. First, we dealt with block sorting algorithms, which can sort a number of elements proportional to the number of available processors (the proportionality constant being dependent on the size of the processors' memory). Then we introduced parallel external sorting algorithms, which address the problem of sorting in parallel an arbitrarily large mass-storage file.

In addition to these three classes of parallel sorting algorithms, we described (in Section 6) a number of hardware sorter designs. Hardware sorters that have been proposed assume a fixed, sparse interconnection scheme between the processing elements. The parallel sorting algorithms utilized by these sorters are highly synchronous and, for the most part, are derived from algorithms that we have classified as network sorting algorithms (in particular, the bitonic sort algorithm). Although the hardware sorters did not introduce innovative parallel sorting algorithms, new and important directions of research on parallel sorting are being explored in their design process. One such direction is the exploration of algorithms that exploit the characteristics of new storage technologies such as magnetic bubbles. Another is the integration of VLSI hardware complexity in cost models by which parallel sorting algorithms are being evaluated.

One conclusion emerges clearly from this survey. Most research in the area of parallel sorting has concentrated on finding new

ways to speed up the theoretical computation time of sorting algorithms, whereas other aspects (such as technology constraints or data dependency) have received less consideration. Typically, algorithms have been developed for hypothetical computers that utilize unlimited parallelism and space to solve the sorting problem in asymptotically minimal time. It seems that today the complexity of sorting is fairly understood, whether on networks or on shared memory parallel processors. Open questions that remain on the complexity of parallel sorting are mostly related to newer models of VLSI complexity that combine chip area with time [Thompson 1983].

It might be the case that after a decade of research devoted mainly to the theoretical complexity of parallel sorting, aspects related to the feasibility of parallel sorting in the context of current or near-term technology will now be more systematically explored. To appreciate the practical importance of parallel sorting, one should remember that the first parallel sorting algorithms were intended to solve a hardware problem: building a switching network that could provide all permutations of n input lines, with a delay shorter than the time required by serial sorting. It would be interesting, now that many fast parallel sorting algorithms are known, to investigate whether these algorithms can be adapted to realistic models of parallel computation. In particular, further research is needed to address issues related to limited parallelism (to remove the constraint relating the number of processors to the problem size), partial broadcast (to replace simultaneous reads to the same memory location), and resolution of memory contention. Another important problem is related to the validity of the performance criteria by which parallel sorting algorithms have been previously evaluated. It is clear that communication, input/output costs, and hardware complexity must be integrated in a comprehensive cost model that is general enough to include a wide range of parallel processor architectures. In particular, the initial cost of reading the source data into the processors' memories has been largely ignored by previous research on parallel sorting. Although one is justified in ignoring this issue when considering a serial, internal sorting algorithm, the situation is quite different with parallel processing. On a single processor the source data are read sequentially into memory. For a parallel processor there is the possibility that several processors can simultaneously read or write. On the ILLIAC-IV computer, for example, a fixed-head disk was used for concurrent input/output by all 64 processors. However, when a significantly larger number of processors is involved, only part of them may be able to perform input/output operations concurrently. Thus, for parallel internal sorting, the cost of reading and writing the data should be incorporated when an algorithm is evaluated. In particular, there may be no point in using a parallel sorting algorithm that requires only $O(\log n)$ time, if the start-up cost to get the data in memory were $O(n)$. Modeling the cost of input/output is even more crucial when the problem of sorting a large data file in parallel is addressed. The importance of file sorting in database systems will undoubtedly motivate further research in this direction.

REFERENCES

AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. 1983. An $O(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (Boston, Apr. 25–27). ACM, New York, pp. 1–9.

ALEKSEYEV, V. E. 1969. On certain algorithms for sorting with minimal memory. *Kibernetica 5*, 5.

BANERJEE, J., BAUM, R. I., AND HSIAO, D. K. 1978. Concepts and capabilities of a database computer. *ACM Trans. Database Syst. 3*, 4 (Dec.), 347–384.

BATCHER, K. E. 1968. Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computer Conference* (Atlantic City, N.J., Apr. 30–May 2), vol. 32. AFIPS Press, Reston, Va., pp. 307–314.

BAUDET, G., AND STEVENSON, D. 1978. Optimal sorting algorithms for parallel computers. *IEEE Trans. Comput. C-27*, 1 (Jan.).

BENTLEY, J. L., AND KUNG, H. T. 1979. A tree machine for searching problems. In *Proceedings of the 1979 International Conference on Parallel Processing* (Aug.).

BILARDI, G., AND PREPARATA, F. P. 1983. A minimum area VLSI architecture for $O(\log n)$ time sorting. TR-1006, Computer Science Department, Univ. of Illinois at Urbana-Champaign (Nov.).

BITTON, D., AND DEWITT, D. J. 1983. Duplicate record elimination in large data files. *ACM Trans. Database Syst. 8*, 2 (June), 255–265.

BITTON-FRIEDLAND, D. 1982. Design, analysis and implementation of parallel external sorting algorithms. Ph.D. dissertation, TR464, Computer Science Department, Univ. of Wisconsin, Madison (Jan.).

BORODIN, A., AND HOPCROFT, J. E. 1982. Routing, merging and sorting on parallel models of computation. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing* (San Francisco, Calif., May 5–7). ACM, New York, pp. 338–344.

BRYANT, R. 1980. External sorting in a layered storage architecture. Lecture. IBM Research Center, Yorktown Heights, N.Y.

CHEN, T C., LUM, V. Y., AND TUNG, C. 1978. The rebound sorter: An efficient sort engine for large files. In *Proceedings of the 4th International Conference on Very Large Data Bases* (West Berlin, FRG, Sept. 13–15). IEEE, New York, 312–318.

CHUNG, K., LUCCIO, F., AND WONG, C. K. 1980. On the complexity of sorting in magnetic bubble memory systems. *IEEE Trans. Comput. C-29* (July).

DOHI, Y., SUZUKI, A., AND MATSUI, N. 1982. Hardware sorter and its application to data base machine. In *Proceedings of the 9th Conference on Computer Architecture* (Austin, Tex., Apr. 26–29). IEEE, New York, pp. 218–225.

EVEN, S. 1974. Parallelism in tape-sorting. *Commun. ACM 17*, 4 (Apr.), 202–204.

FENG, T.-Y. 1981. A survey of interconnection networks. *Computer 14*, 12 (Dec.).

FISHBURN, J. P., AND FINKEL, R. A. 1982. Quotient networks. *IEEE Trans Comput C-31*, 4 (Apr.).

GAVRIL, F. 1975. Merging with parallel processors. *Commun ACM 18*, 10 (Oct.), 588–591.

HIRSCHBERG, D. S. 1978. Fast parallel sorting algorithms. *Commun. ACM 21*, 8 (Aug.), 657–666.

HSIAO, D. C., AND MENON, M. J. 1980. Parallel record-sorting methods for hardware realization. Tech. Rep. OSU-CISRC-TR-80-7, Computer and Science Information Dept., Ohio State Univ., Columbus, Ohio (July).

KNUTH, D. E. 1973. Sorting and searching. In *The Art of Computer Programming*, vol. 3 Addison-Wesley, Reading, Mass

KUMAR, M., AND HIRSCHBERG, D. S. 1983. An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Trans Comput C-32* (Mar.).

LEE, D. T., CHANG, H., AND WONG, K. 1981. An on-chip compare/steer bubble sorter. *IEEE Trans Comput. C-30* (June).

LEILICH, H. O., STIEGE, G , AND ZEIDLER, H. C. 1978. A search processor for database management systems. In *Proceedings of the 4th Conference on Very Large Databases* (West Berlin, FRG, Sept. 13–15) IEEE, New York, pp. 280–287.

LEISERSON, C. E. 1981. Area-efficient VLSI computation. Ph.D. dissertation. Tech. Rep. CMU-CS-82-108, Computer Science Dept.; Carnegie-Mellon Univ., Pittsburgh, Pa. (Oct.).

MULLER, D. E., AND PREPARATA, F. P. 1975. Bounds to complexities of networks for sorting and for switching. *J ACM 22*, 2 (Apr.), 195–201.

NASSIMI, D., AND SAHNI, S. 1979. Bitonic sort on a mesh connected parallel computer. *IEEE Trans. Comput C-27*, 1 (Jan.).

NASSIMI, D., AND SAHNI, S. 1982. Parallel algorithms to set up the Benes permutation network. *IEEE Trans Comput. C-31*, 2 (Feb.).

PEASE, M. C. 1977. The indirect binary n-cube microprocessor array. *IEEE Trans Comput C-26*, 5 (May).

PREPARATA, F. P. 1978. New parallel sorting schemes. *IEEE Trans. Comput. C-27*, 7 (July).

PREPARATA, F. P., AND VUILLEMIN, J. 1979. The cube-connected-cycles. In *Proceedings of the 20th Symposium on Foundations of Computer Science.*

SELINGER, P. G., ASTRAHAN, M. M , CHAMBERLIN, D D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, Mass., May 30–June 1). ACM, New York, pp. 23–34.

SHILOACH, Y , AND VISHKIN, U. 1981. Finding the maximum, merging and sorting in a parallel computation model. *J Algorithms 2*, 1 (Mar.).

SIEGEL, H. J. 1977. The universality of various types of SIMD machine interconnection networks. In *Proceedings of the 4th Annual Symposium on Computer Architecture* (Silver Spring, Md., Mar. 23–25). ACM SIGARCH/IEEE-CS, New York.

SIEGEL, H. J. 1979. Interconnection networks for SIMD machines. *IEEE Comput 12*, 6 (June).

STONE, H. S. 1971 Parallel processing with the perfect shuffle *IEEE Trans Comput C-20*, 2 (Feb.).

THOMPSON, C. D. 1980. A complexity theory for VLSI. Ph.D. dissertation, Tech. Rep. CMU-CS-80-140, Computer Science Dept , Carnegie-Mellon Univ , (Aug.).

THOMPSON, C. D. 1983. The VLSI complexity of sorting. *IEEE Trans Comput C-32*, 12 (Dec.).

THOMPSON, C. D., AND KUNG, H T 1977 Sorting on a mesh-connected parallel computer. *Commun. ACM 20*, 4 (Apr.), 263–271.

VALIANT, L G. 1975 Parallelism in comparison problems. *SIAM J Comput 3*, 4 (Sept.).

VAN VOORHIS, D. C. 1971. On sorting networks. Ph.D. dissertation, Computer Science Dept., Stanford Univ , Stanford, Calif.

VISHKIN U. 1981 Synchronized parallel computation. Ph.D. dissertation, Computer Science Dept., Technion—Israel Institute of Technology, Haifa, Israel.

YASUURA, H., TAKAGI, N., AND YAJIMA, S 1982. The parallel enumeration sorting scheme for VLSI. *IEEE Trans Comput C-31*, 12 (Dec.).

Received August 1982, final revision accepted August 1984