A Taxonomy of Parallel Sorting

Dina Bitton†
David J. DeWitt††
David K. Hsiao*
Jaishankar Menon**

TR 84-601
April 1984

†Department of Computer Science
Cornell University
Ithaca, New York 14853

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

*Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

**IBM Research Center
San Jose, California 93940

A TAXONOMY OF PARALLEL SORTING

by

Dina Bitton[+], David J. DeWitt[++]

David K. Hsiao[*] and Jaishankar Menon[**]

+ Department of Computer Science, Cornell University, Ithaca, NY 14853.

++ Computer Science Department, University of Wisconsin, Madison, WI 53706.

* Department of Computer Science, Naval Postgraduate School, Monterey, CA 93940.

** IBM Research Center, San Jose, CA 93940.

ABSTRACT

In this paper, we propose a taxonomy of parallel sorting that includes a broad range of array and file sorting algorithms. We analyze the evolution of research on parallel sorting, from the earliest sorting networks to the shared memory algorithms and the VLSI sorters. In the context of sorting networks, we describe two fundamental parallel merging schemes - the odd-even and the bitonic merge. Sorting algorithms have been derived from these merging algorithms for parallel computers where processors communicate through interconnection networks such as the perfect shuffle, the mesh and a number of other sparse networks. After describing the network sorting algorithms, we show that, with a shared memory model of parallel computation, faster algorithms have been derived from parallel enumeration sorting schemes, where keys are first ranked and then rearranged according to their rank.

Parallel sorting algorithms are evaluated according to a number of criteria, related not only to their time complexity, but also to their feasibility from a computer architecture point of view. We show that in addition to their attractive communication schemes, network sorting algorithms have non-adaptive schedules that make them suitable for implementation. In particular, they are easily generalized to block-sorting algorithms, that 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 memories. Finally, the newer area of VLSI sorting is mentioned as an active and promising direction of research on parallel sorting.

1. INTRODUCTION

Sorting, in computer terminology, is defined as the process of rearranging in ascending or descending order a set of values stored in contiguous memory locations. It is often the case that computer programs such as compilers or editors choose to sort tables and lists of symbols stored in memory, in order to enhance the speed and the simplicity of algorithms used to access them (for search or insertion of additional elements, for instance). Because of its practical importance as well as its theoretical interest, the problem of sorting values stored in random access memory *(internal sorting)* has 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 research area. Many efficient serial algorithms are known that can sort an array of size n in at most O(nlogn) comparisons, the theoretic lower bound for this problem. Various properties of these *serial internal sorting* algorithms, in addition to their time complexity, have also been investigated. In particular, sorting algorithms have been evaluated with respect to *time-memory trade-offs* (that is the amount of additional memory required to run the algorithm, in addition to the memory storing the initial sequence), *stability* (that is the requirement that equal elements retain their original relative order), or their 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, for which the notion of "contiguous" memory locations and the way multiple processors access memory are defined in different ways. In order to clearly state the parallel sorting problem, one must first define what is meant by a sorted sequence in a parallel processor. When processors share a common memory, the notion of contiguous memory locations is identical to the serial processor's memory case. Thus, 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, as in the serial case. On the other hand, when processors do not share memory and communicate along the lines of an interconnection network, a convention to order the processors and, thus, the union of their local memory locations is required to define the sorting problem. When parallel processors are used, the time complexity of a sorting algorithm is expressed in terms of parallel comparisons and exchanges between processors 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. While the first results on parallel sorting were related to sorting networks [Batcher 1968], faster parallel sorting algorithms were first proposed for theoretical models of parallel processors with shared memory [Hirschberg 1978, Preparata 1978]. A chain of results in this area has led to a number of parallel sorting schemes that exhibit a O(logn) time complexity. For the most part, research on parallel sorting has been concerned with pure theoretical issues. Typically, the parallel sorting problem is stated as the problem of sorting n numbers with n or more processors, all sharing a large common memory, that they may access with various degrees of contention (e.g. parallel reads and parallel writes with arbitration). It is only recently that feasibility issues such as limited parallelism or, in the context of VLSI sorting, trade-offs between hardware complexity (expressed in terms of chip area) and time complexity, are being addressed.

Besides its use in rearranging numbers in memory, sorting is often advocated in the context of information processing. In this context, sorting consists of ordering a file of data records, initially 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. Because of memory limitations, file sorting cannot be performed in memory and *external sorting algorithms* must be used. Files are sorted either to deliver a well-organized output to a user (e.g. a telephone directory), or as an intermediate step in the execution of a complex database operation [Selinger et al. 1979, Bitton and DeWitt 1983]. External sorting schemes are usually based on iterative merging [Knuth 1973, 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]

---

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

Despite the clear 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 that explain the relatively small number of studies dealing with parallel external sorting [Even 1974, Bitton 1982] are most likely related to the necessity of adapting these sorting schemes to mass-storage device characteristics.

It may seem that advances in computer technology could eliminate, or at least significantly 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 the advent of associative memories will suppress the need of sorting. However, associative stores remain too expensive for widespread usage, especially when large volumes of data are involved. Also in the case that 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 the 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 the evolution of research on parallel sorting - from the earliest sorting networks to the shared memory model algorithms and the VLSI sorters. We attempt to classify a broad range of parallel sorting algorithms, according to various criteria that include not only their time efficiency, but also the architectural requirements that they rely upon. The goal of the present study is to provide a basic understanding as well as a unified view of the body of research on parallel sorting. It would be beyond the scope of a single paper to survey in detail the models of computation that have been proposed, or to analyze in depth the complexity of the 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 1983, Shiloah et al. 1981, Valiant 1981]), we emphasize 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 2, we show that certain fast serial sorting algorithms can be parallelized but this approach leads to simple and relatively slow parallel algorithms. Section 3 is devoted to the network sorting algorithms. In particular, we describe in detail several sorting networks that perform Batcher's bitonic sort. Section 4 surveys a chain of results that led to the development of very fast sorting algorithms: the shared memory model parallel merging [Valiant 1975, Gavril 1975], and the shared memory sorting algorithms [Hirschberg 1978, Preparata 1978]. In Section 5, we address the issue of limited parallelism and in this context, we define "block-sorting" parallel algorithms, that sort M.p elements with p processors. We then identify two methods for deriving a block-sorting algorithm. In Section 6, 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 7 contains an overview of recently proposed designs for dedicated sorting devices. Finally, Section 8 summarizes this survey and indicates possible directions for future research on parallel sorting.

2. 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, at that parallelism is exploited by comparing 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, using (n-1) processors.

Parallelism may 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 becomes 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 they require. Then, the *speedup* of a parallel sorting algorithm 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(nlogn) comparisons to sort n elements [Knuth 1973, p.183], the optimal speedup would be achieved when, using n processors, n elements are sorted in O(logn) parallel comparisons. It does not, however, seem possible to achieve this bound by simply parallelizing one of the well-known O(logn)-time serial sorting algorithms. These algorithms appear to have serial constraints that cannot be relaxed. A simple example will illustrate this problem. Consider a 2-way merge sort [Knuth 1973, p.160]. The algorithm consists of logn 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 2 sequences of n/2 elements each is a serial process that may require as many as n-1 comparisons.

On the other hand, parallelization of straight sorting methods that require $O(n^2)$ comparisons seems easier, but this approach can at most 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 for this kind of parallelization is a well-known parallel version of the common bubble-sort, called the *odd-even transposition sort* (Section 2.1).

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 the comparisons at the same level of the tree are performed in parallel. The result is a parallel tree sort that is described in Section 2.2. This parallel algorithm is used in the database Tree Machine [Bentley and Kung 1979].

2.1. The odd-even transposition sort

The serial "bubble-sort" proceeds by comparing and exchanging pairs of adjacent items. In order to sort an array $(x_1,x_2,...,x_n)$, (n-1) comparison-exchanges (x_1,x_2), (x_2,x_3),...,(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 $(x_1,x_2,...,x_{n-1})$. By iterating (n-1) times, the entire sequence is sorted.

The serial odd-even transposition sort [Knuth 1973, p. 65] is a variation of the basic bubble sort, with a total of n phases, each requiring 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 compared are (x_2,x_3), (x_4,x_5),... . To completely sort the sequence, it has been shown that 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,..., P_n. We 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 resided in P_i for i=1,2,...,n. To sort $(x_1, x_2,...,x_n)$ in parallel, let P_1, P_3, P_5, ... 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, ... be active during the even time steps, and perform the even phases in parallel.

Note that a single comparison-exchange requires 2 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   3   4   5   6   7   8
  X       X       X       X
      X               X

              X

           STEP 1
```

```
X   2   X   4   X   6   X   8
  1       3       5       7
      X               X

              X

           STEP 2
```

```
X   2   X   4   X   6   X   8
  X       3       X       7
      1               5

              X

           STEP 3
```

```
X   2   X   4   X   6   X   8
  X       3       X       7
      X               5

              1

           STEP 4
```

FIGURE 1: PARALLEL TREE SELECTION SORT

2.2. A parallel tree-sort algorithm

In a serial tree selection sort, n numbers are sorted using a binary tree data structure. 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 a processor at each of the n leaf nodes in addition to every interior node of the tree. 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 non empty children and to select the minimum if they receive 2 elements. At every other step, the next element in increasing order reaches the root. Thus sorting is completed in time O(n).

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 will show that faster parallel sorting algorithms have been developed by exploiting the intrinsic parallelism in sorting, rather than parallelizing serial sorting algorithms.

3. NETWORK SORTING ALGORITHMS

It is somehow surprising that a simple hardware problem, namely designing a multiple-input multiple-output switching network, has motivated the development and the proliferation of parallel sorting algorithms. The earliest results in the parallel sorting area are found in the literature on sorting networks [Van Voorhis 1971, Batcher 1968]. 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 3.1, we will describe in detail the odd-even and the bitonic merging networks. In Section 3.2, 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 [Thompson and Kung 1977, Nassimi and Sahni 1979].

Several other networks are of major interest. In particular, the Cube [Pease 1977] and the Cube-Connected-Cycles [Preparata and Vuillemin 1979] have been shown to be 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 in detail these networks. We will 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.

3.1. 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,...,n), it can be used as a multiple-input multiple-output switching network [Batcher 1968]. To design a fast sorting network, it is necessary to exploit the possibility of having a number of comparator modules perform comparisons in parallel. 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.

One of the first results in parallel sorting is due to Batcher [Batcher 1968], who presented two methods to sort n keys with $O(n\log^2 n)$ 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, where several bits are compared in parallel at each step, 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. Starting with an initial sequence of 2^k numbers, a specific iterative rule is applied to create sorted runs of length 2, 4, 8, ..., 2^k during successive stages of the algorithm.

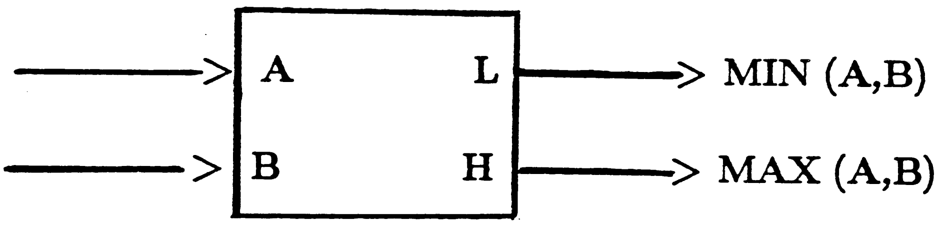


FIGURE 2 A COMPARISON-EXCHANGE MODULE

3.1.1. 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, ..., a_n) and (b_1, b_2, ..., b_n), two new sequences (the "odd" and "even" sequences) are created: one consists of the odd numbered terms and the other of the even numbered terms from both sequences. The odd sequence (c_1, c_2, ...) is obtained by merging the odd

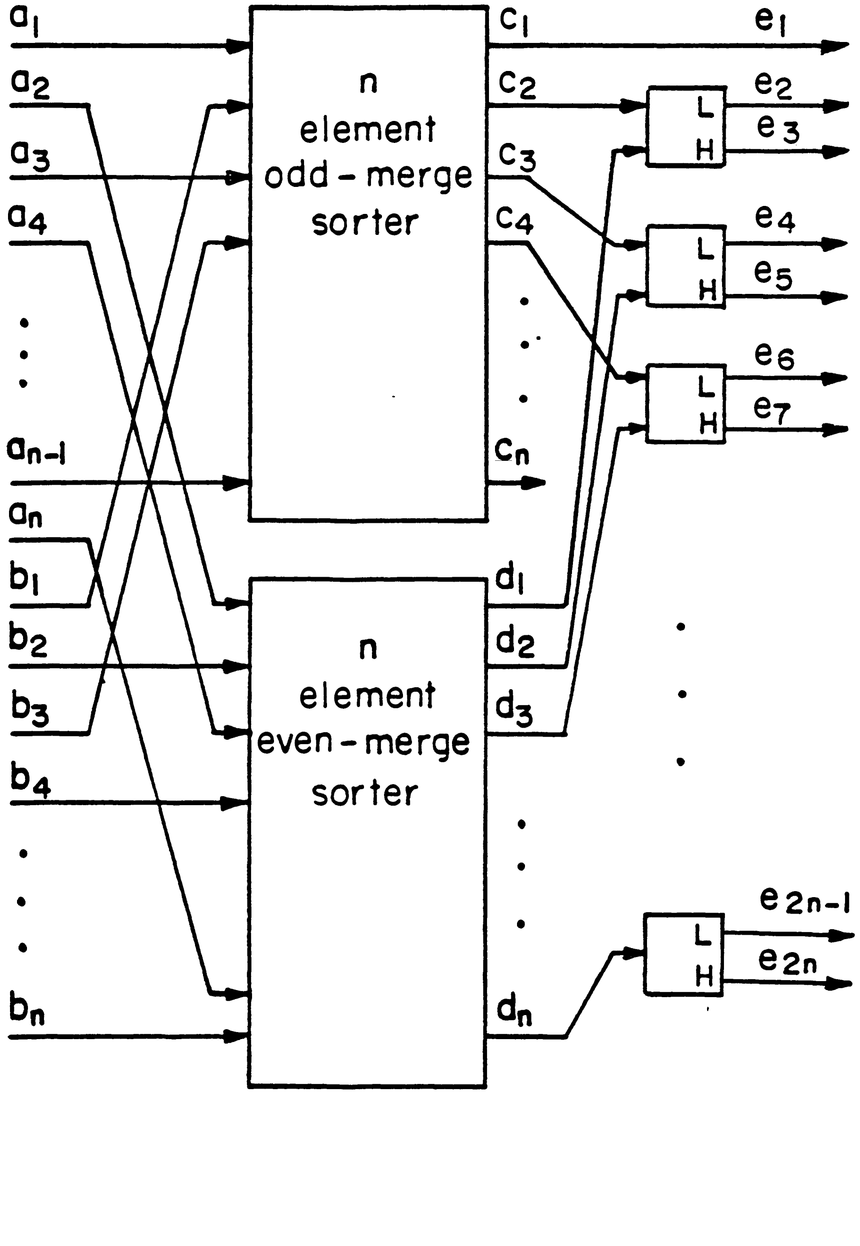


FIGURE 3: THE ITERATIVE RULE FOR THE ODD-EVEN MERGE

terms (a_1, a_3, ...) with the odd terms (b_1, b_3, ...). Similarly, the even sequence (d_1, d_2, ...) is obtained by merging (a_2, a_4, ...) with (b_2, b_4, ...). Finally, the sequences (c_1, c_2, ...) and (d_1, d_2, ...) are merged into (e_1, e_2, ..., e_{2n}) by performing the following comparison-exchanges:

$$e_1 = c_1$$
$$e_{2i} = \min(c_{i+1}, d_i)$$
$$e_{2i+1} = \max(c_{i+1}, d_i), \text{ for } i=1,2,...$$

The resulting sequence will be sorted (for a proof, the reader is referred to [Knuth 1973, p. 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, etc. 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 $1+2+3+...+k = k(k+1)/2$ 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].

3.1.2. 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, ..., a_{2n}) be a bitonic sequence such that $a_1 \leq a_2 \leq ... \leq a_n$ and $a_{n+1} \geq a_{n+2} \geq ... \geq a_{2n}$.[2] Then the sequences

$$\min(a_1, a_{n+1}), \min(a_2, a_{n+2}), ...$$
$$\text{and } \max(a_1, a_{n+1}), \max(a_2, a_{n+2}), ...$$

are both bitonic. Furthermore, the first sequence contains the n lower elements of the original sequence, while the second contains the n higher ones. It follows that a bitonic sequence can be

[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.

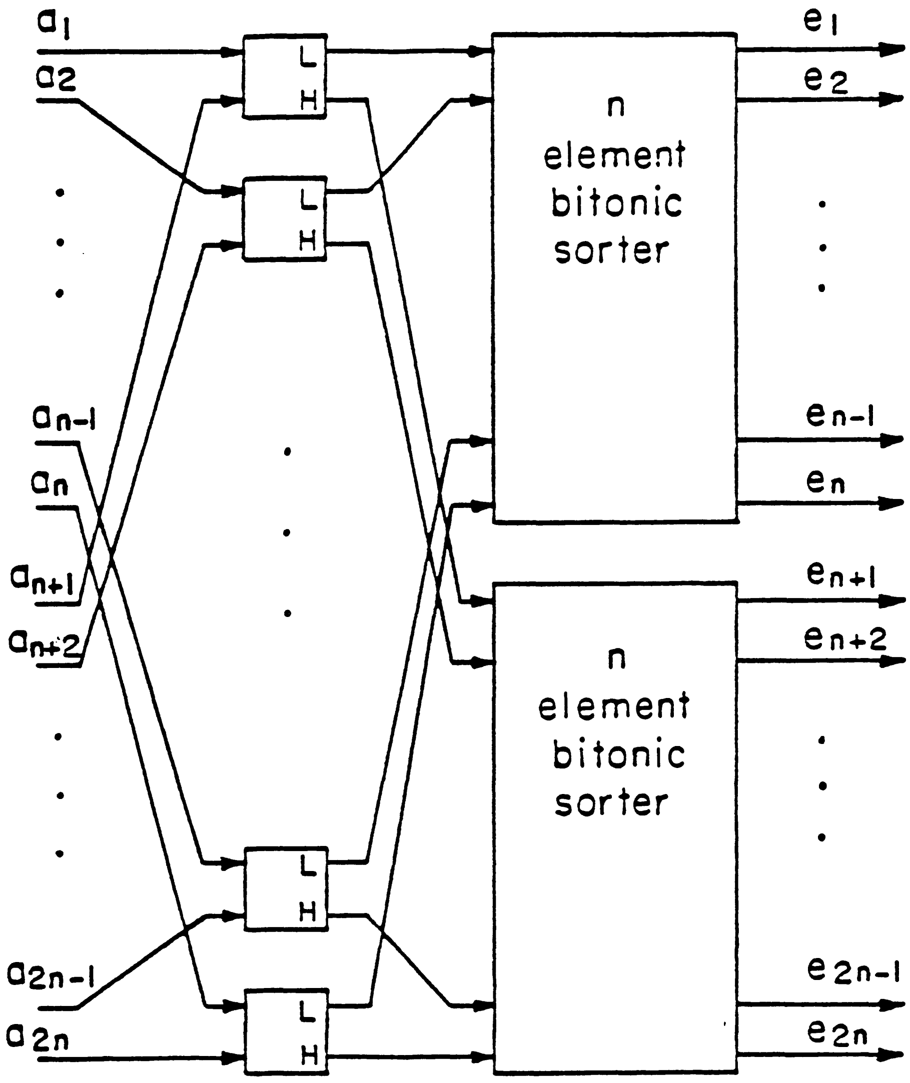
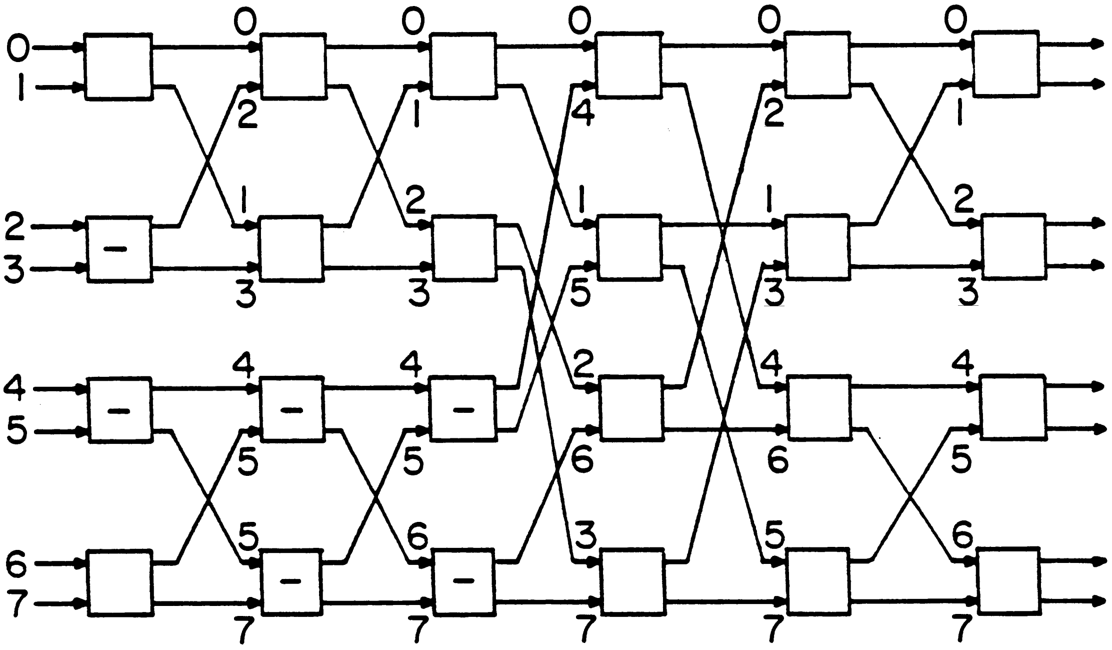


FIGURE 4: THE ITERATIVE RULE FOR THE BITONIC MERGE



$\boxed{-}$: Indicates a comparator which inverts its output lines.

FIGURE 5: BATCHER'S BITONIC SORT FOR 8 NUMBERS

sorted by sorting separately two bitonic sequences each half as long as the original sequence.

To sort 2^k numbers 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 must use some comparators that 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 that bitonic sort network for 8 input lines. In general, the bitonic sort of 2^k numbers requires $k(k+1)/2$ steps, each 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, with *perfect shuffle* interconnection [Stone 1971]. Stone noticed that if the inputs were labeled by a binary index, then the indices of every pair of keys that enter a comparator at any step of the bitonic sorting network, would differ by a single bit in their binary representations. Stone also made the following observations: The network has logn 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" the 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 illustrated in Figure 6 for 16 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 logn stages, with each stage consisting of logn steps. At each step, the output lines are shuffled before they enter the next rank of comparators. The comparators in the first (logn)-i steps of the ith stage do not exchange their inputs. Their only use is to shuffle their input lines.

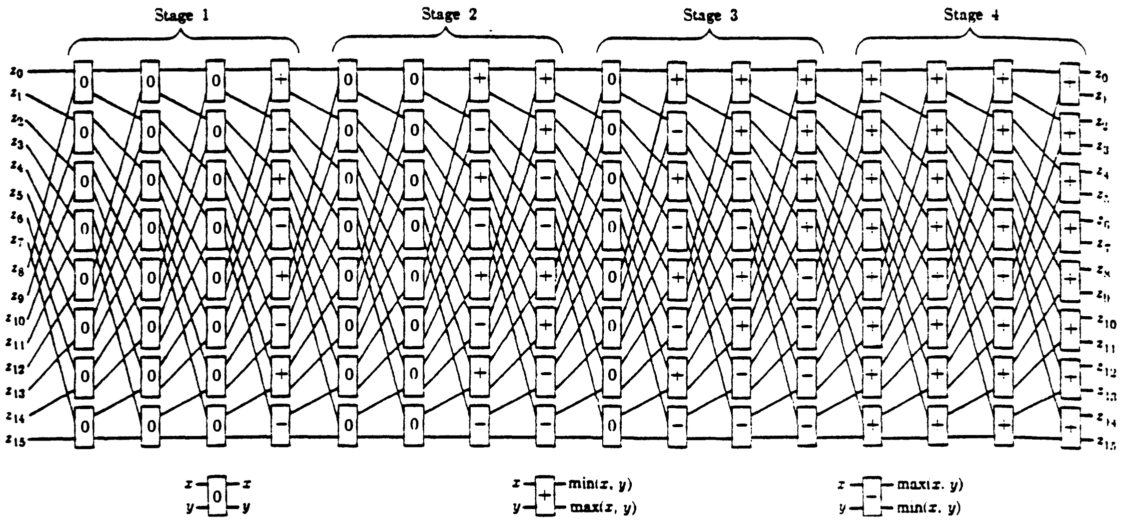


FIGURE 6: STONE'S MODIFIED BITONIC SORT

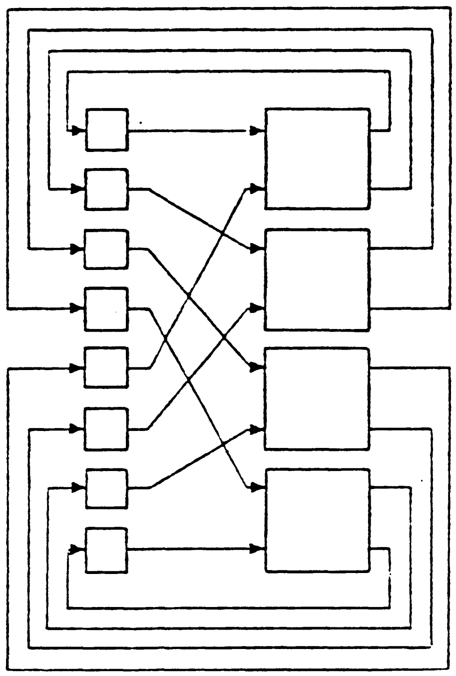


FIGURE 7: STONE'S ARCHITECTURE FOR THE BITONIC SORT

Instead of 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 bitonic 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

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

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, it improves significantly the previous known bound of $O(n)$ for the time 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]. Among the networks he considered, are the Cube and the Plus-Minus 2^i networks. Essentially, the data exchanges required by the bitonic sort scheme can be realized on these networks too (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, that sorting can also be performed within this time limit. Finally, one should also mention the versatile *Cube-Connected-Cycles (CCC)*, a network that efficiently emulates the Cube and the shuffle, and yet requires only 3 communication ports per processor [Preparata and Vuillemin 1979]. A bitonic or an odd-even sort can also be performed on the CCC in time $O(\log^2 n)$.

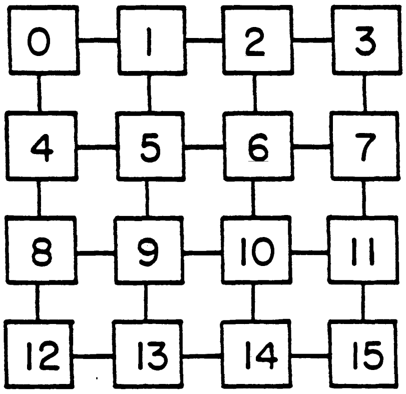
3.2. Sorting on an SIMD machine

Sorting networks are characterized by their "non adaptivity" property. They perform the same sequence of comparisons regardless of the result of intermediate comparisons. In other words, whenever two keys R_i and R_j are compared, the subsequent comparison for R_j in the case that $R_i < R_j$ are the same as the comparison that R_j would have entered in the case $R_j < R_i$.

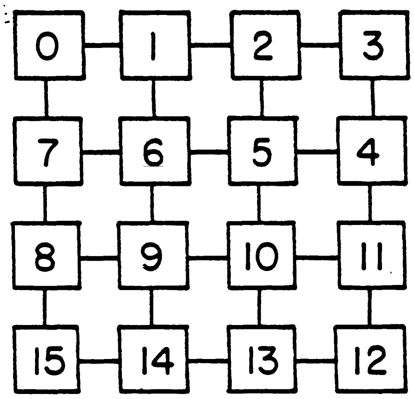
The non-adaptivity property makes the implementation of an algorithm very convenient for an SIMD machine. An SIMD (single instruction stream, multiple data stream) machine is a system consisting of a control unit and a set of processors with local memory and interconnected by an interconnection network. The processors are highly synchronized. The control unit broadcasts instructions which all the 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.

3.2.1. Sorting on an array processor

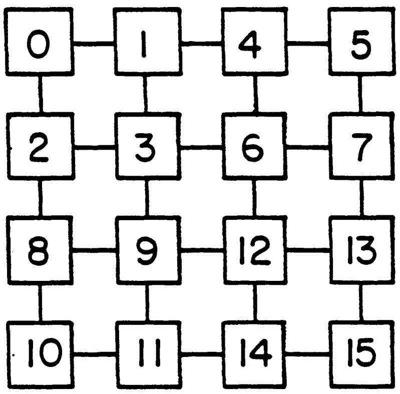
The sorting problem can be also defined as the problem of permuting numbers between the local memories associated with processors of an SIMD machine. In particular, assuming that one number is stored in the local memory of each processor of a mesh-connected machine, sorting may be viewed as the process of that permutes 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 may be indexed according to a prespecified rule. The row-major or column-major indexing are commonly accepted ways to order an array. Thompson and Kung adapted the bitonic sorting scheme to a mesh-connected processor, with three alternative indexing rules: the row-major, the snake-like row-major, and the shuffled row-major rules [Thompson and Kung 1977]. These rules are shown in Figure 8. Assuming that n^2 keys with arbitrary numbers 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 the sorting networks, parallelism is used to simultaneously compare pairs of numbers, 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



Row-major indexing



Snake-like row-major indexing



Shuffled row-major indexing

FIGURE 8: ARRAY PROCESSOR, INDEXING SCHEMES

execution by the set of processors specified in the instruction. The complexity of a method which solves the sorting problem for this model can be measured in terms of the number of 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 such a permutation will require at least 4(n-1) moves, since it may have to interchange the elements from two opposite corners of the array processor (and this is true for any indexing scheme). In this sense a sorting algorithm which requires O(n) moves is optimal.

The odd-even and the bitonic network sorting algorithms have been 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 snake-like 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]. The latter algorithm is also an adaptation of the bitonic sort where 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]. Based on this merge pattern, a two- dimensional array can be sorted in row-major order, in time O(n) and a smaller proportionality constant than the previous algorithms.

3.3. Summary

In this section we have examined two well-known sorting networks, and shown that the sorting network concept 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 sorting network algorithms. It is beyond the scope of this study to provide a comprehensive analysis of interconnection networks. However, an extensive literature exists on this topic, and we have listed some references for the interested reader [Pease 1977, Siegel 1977, Thompson 1980, Preparata and Vuillemin 1979, Siegel 1979, Feng 1981, Nassimi and Sahni 1982].

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 that 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 4, we will 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 now renew the interest in network sorting algorithms [Ajtai et al. 1983]. A network of $O(n\log n)$ comparators is shown 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.

4. SHARED MEMORY PARALLEL SORTING ALGORITHMS

After the time bound of $O(\log^2 n)$ was achieved with the sorting networks algorithms, attention was directed towards improving this bound to the theoretical lower bound of $O(\log n)$. In this section, several parallel algorithms are described that sort n elements with $O(\log n)$ comparisons. These algorithms assume a shared memory model of parallel computation. While the sorting network algorithms are based on comparison-exchanges of pairs, the shared-memory algorithms, for the most part, 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, while in the network sorting algorithms, individual processors decide *locally* about their next routing step, by communicating with their nearest neighbors, in the shared memory algorithms, any processor may access any location of the global memory, at every computation step. As shown in Section 3, 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, they are mostly of theoretical interest, while the network models are more suitable to implemention with current or near-term technology.

In the remainder of this section, we will 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 non-adaptive network merge algorithms (the odd-even and the bitonic merge described in Section 3), and sorting algorithms that combine enumeration with parallel merge procedures [Preparata 1978]. In addition to these enumeration sorting algorithms, we will describe a parallel bucket sorting algorithm [Hirschberg 1978].

4.1. A modified sorting network

In a first attempt to reduce the number of comparisons required for sorting, by increasing the degree of parallelism beyond O(n), Muller and Preparata first proposed a modified sorting network, based on a different type of comparators (Figure 9). These comparators have 2 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 logn 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 (logn)+ 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 logn time units, and this determines the algorithm complexity. A diagram for this type of network is presented in Figure 9.

At the cost of additional hardware complexity, the above algorithm sorts n elements in O(logn) time, with $O(n^2)$ processing elements. This algorithm was the first to use an *enumeration*

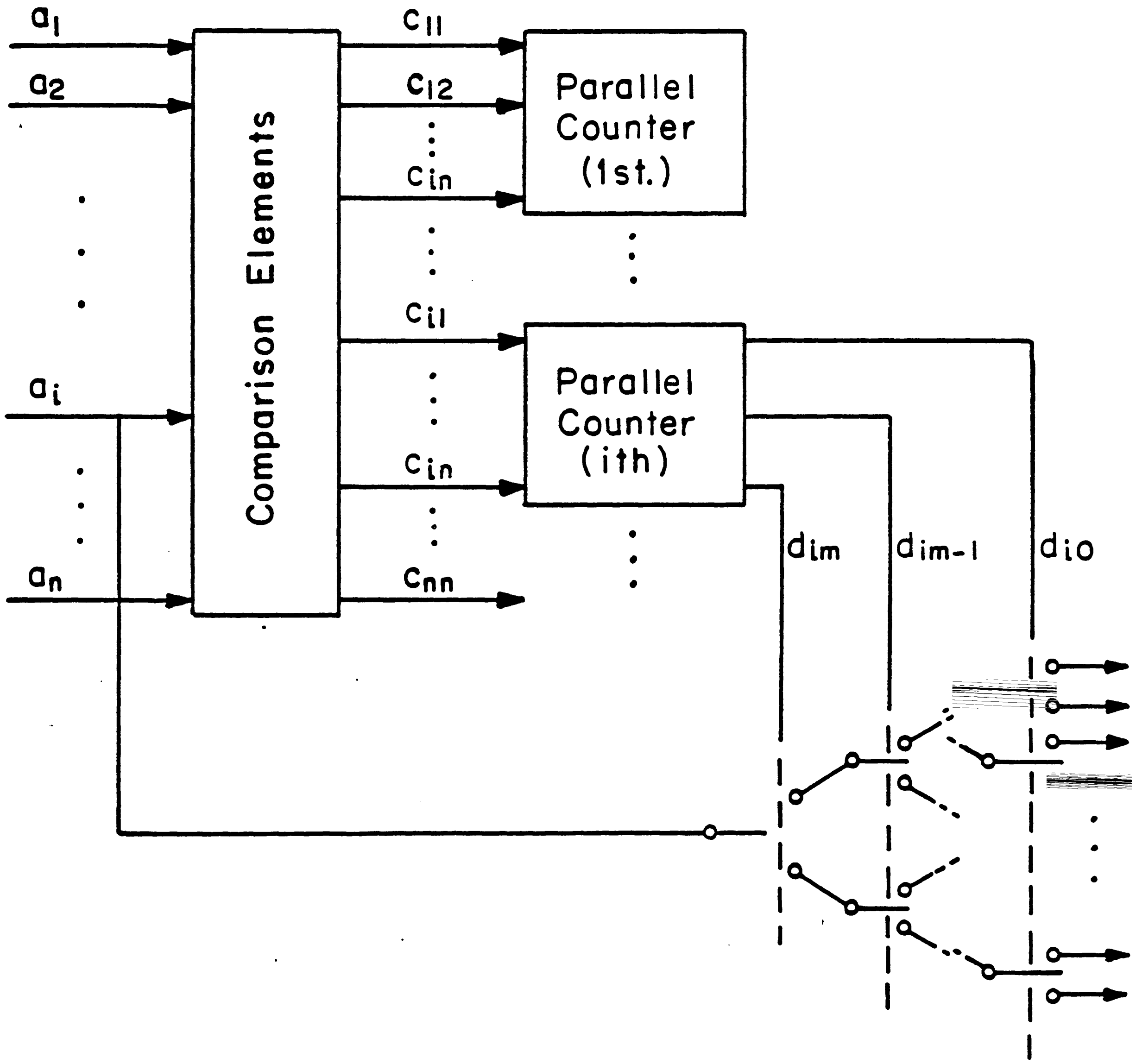


FIGURE 9: MULLER AND PREPARATA'S SORTING NETWORK

scheme for parallel sorting. The idea of sorting by enumeration was exploited to develop other very fast parallel sorting algorithms [Hirschberg 1978, Preparata 1978], that 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(logn) is not satisfactory. A parallel sorting algorithm could theoretically achieve the same speed with only O(n) processors, if it had a parallel speedup of order n.

4.2. Faster parallel merging algorithms

In addition to the idea of using enumeration, optimal parallel sorting algorithms may use fast merging procedures. In a study of parallelism in comparison problems, Valiant presents an recursive algorithm that merges two sorted sequences of n and m elements ($n \leq m$) with mn processors in 2log(log n) + O(1) comparison steps (compared to logn for the bitonic merge [Valiant 1975]. On the other hand, Gavril proposes a fast merging algorithm that solves the problem of merging two sorted sequences of length n and m with a smaller number of processors $p \leq n \leq m$ [Gavril 1975]. This algorithm is based on binary insertion, and requires only 2log(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 simultaneously access elements of the initial data, or intermediate computation results.

4.3. Bucket sorting

Hirschberg's algorithm is a "bucket sort" that sorts n numbers with n processors in time O(logn), provided that the numbers to be sorted are in the range {0,1,...,m-1} [Hirschberg 1978]. A side effect of this algorithm is that duplicate numbers -if they appear in the initial sequence- are eliminated in the sorting process. If memory conflicts were 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 may be solved by increasing substantially the memory requirements. Suppose there is enough memory available for m arrays, each of size n. Each processor can then 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 mark 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 which sorts n numbers with $n^{1+1/k}$ processors in time O(klogn) (where k is an arbitrary integer).

A major drawback of the parallel bucket sort (in addition to the lack of feasibility of the shared memory model) is its (m.n) space requirement. Even when the range of values is not very large, it would be desirable to reduce this requirement. In the case of a wide range of values (for example, when the sort keys are arbitrary character strings rather than integer numbers), the algorithm cannot be utilized.

4.4. Sorting by enumeration

Parallel enumeration sorting algorithms, that do not restrict the range of the sort values and yet run in time O(logn), are described in [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 use Valiant's merging procedure [Valiant 1975], and sorts n numbers with nlogn processors in time O(logn). The second algorithm uses Batcher's odd-even merge, and sorts n numbers with $n^{1+1/k}$ processors in time O(klogn). The performance of the latter algorithm is similar to Hirschberg's (Section 3.3), but it has the additional advantage of being free of memory contention. Recall that Hirschberg's model required simultaneous fetches from the shared memory, while Preparata's method does not (since each key participates in only one comparison at any given unit of time).

4.5. 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. In addition, these models usually 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 in [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 by that, 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 in [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 [Shiloah and Vishkin 1981].

5. BLOCK SORTING ALGORITHMS

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

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 explain and 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 that 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 hand, are motivating a new trend of research: algorithms for computers with a relatively small number of processors. An excellent illustration of this trend is a systematic study of *quotient networks,* in [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. Then, together the p processing units emulate a network of size O(n).

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

(2) the concatenation of these local sequences, $S_1,S_2...S_p$, constitutes a sorted sequence of length n.

For example, for 3 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 (that were produced by a previous step), and splits the resulting block into a "higher" and a "lower" block, that 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 consists of comparison-exchange steps) by a merge-split step. It is easy to verify that this procedure produces a sequence which is sorted according to the above definition.

There are two ways to perform a merge-split step. One is based on a 2-way merge [Baudet and Stevenson 1978]; the other on a bitonic merge [Hsiao and Menon 1980]. In Sections 5.1 and 5.2, we describe both methods, and illustrate them by investigating the block sorting algorithms that they generate, based on the odd-even transposition sort (Section 2.1) and the bitonic sort (Section 3.1.2). 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 3.2).

5.1. Two-way merge-split

A two-way merge-split step is defined as a two-way merge of 2 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 a processor's memory before and after a two-way merge-split is shown in Figure 10.

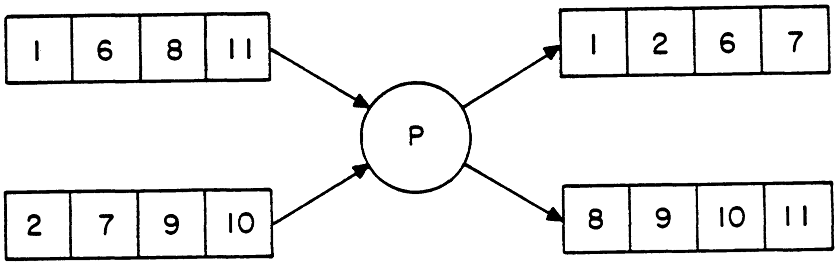


FIGURE 10: MERGE-SPLIT BASED ON 2-WAY MERGE

After 2 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 requires 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.

5.1.1. Block odd-even sort based on 2-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 using any fast serial sorting algorithm. For example, a local 2-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 2.1). During the odd (even) steps, the odd (even) numbered processors receive from their right neighbor a sorted block, perform a 2-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.

5.1.2. Block bitonic sort based on 2-way merge-split

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 2-way merge split on 2 blocks of size M. By replacing each comparison-exchange step by a 2-way merge-split step, we obtain a block bitonic sort algorithm, that can sort M.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 2-way merge of the 2 sequences of length M (that 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 2 processors and 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.p), to sort M.p records.

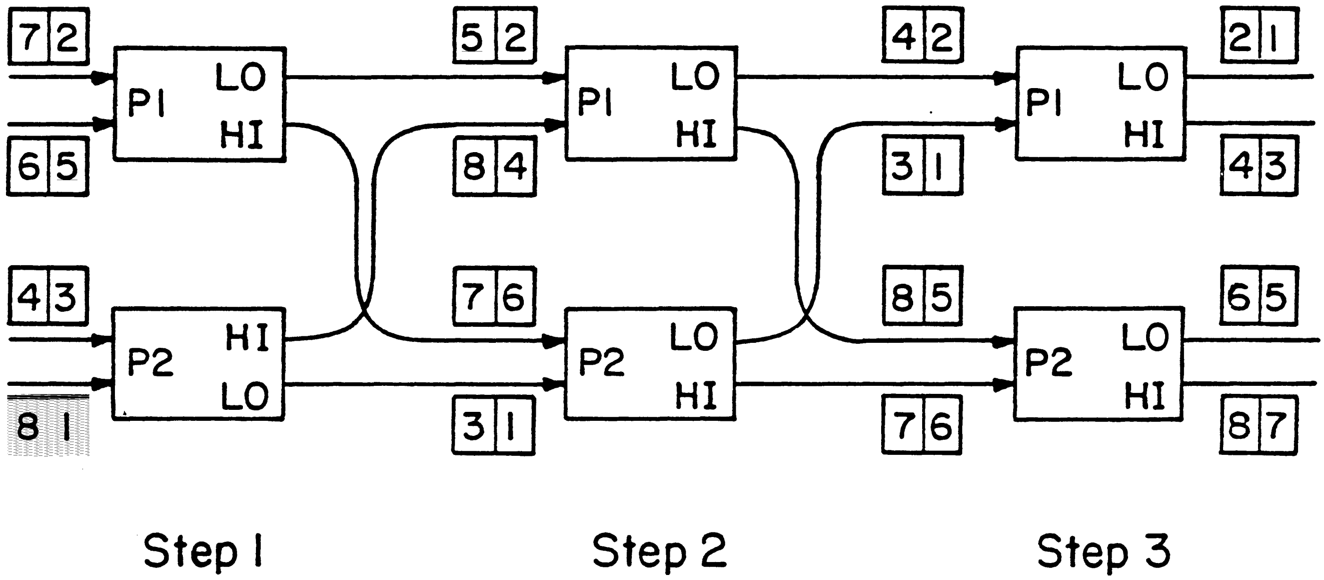


FIGURE 11: BLOCK-BITONIC SORT BASED ON 2-WAY MERGE

5.1.3. Processor synchronization

When M is large, or when the individual records are long, transferring blocks of M.p records between the processors introduce time delays that are by several order of magnitudes higher than the instruction rate of the individual processors. In addition, depending on the data distribution, the number of comparisons required to merge 2 blocks of M records may vary Thus, for the execution of block sorting algorithms based on 2-way merge-split, a coarser granularity for processor synchronization might be more adequate than the SIMD mode of execution, where processors are synchronized at the machine instruction level. A more adequate multiprocessor model for these algorithms is one where 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. At initiation time of a block sorting algorithm, the controller assigns a number of processors to its execution. Because other operations may be already in the process of being executed, 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 taken into consideration by the controller to determine the optimal processor allocation.

5.2. Bitonic merge-exchange

Consider the situation where 2 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:

P_j sends its block to P_i

P_i performs a 2-way merge-split

P_i sends high half block to P_j

However, as indicated in the previous section the 2-way merge-split requires a processor's local memory of size 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 $(x_1, x_2, ..., x_M)$ are stored in increasing order in P_i's memory, and the M records $(y_1, y_2, ..., y_M)$ 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 2 and sends back to P_j the higher record. This procedure is then repeated for $(x_2,y_2),...,(x_M,y_M)$. It is known that 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.

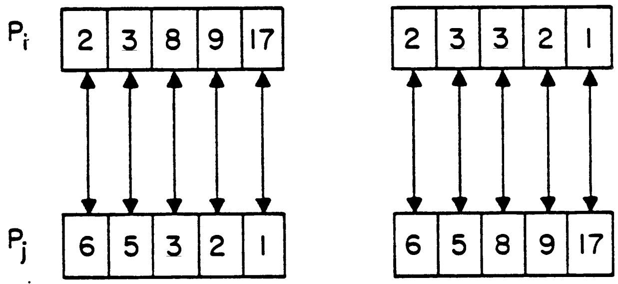


FIGURE 12: BITONIC MERGE-EXCHANGE STEP

It is important to notice that the data exchanges are synchronous (unlike in the 2-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 2-way merge-split. Because the 2-way merge-split merges 2 blocks of size M within a processor's local memory, it requires (4.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. While for the 2-way merge-split, an entire block of data must be transferred to a processor's memory before the merge operation is initiated, for the bitonic merge-exchange, it is possible to overlap each record's transfer time with processing time.

However, a major disadvantage of the bitonic merge-exchange is the necessity to perform 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).logM comparisons and local moves.

5.2.1. Block odd-even sort based on bitonic merge-exchange

As with the block odd-even merge based on two-way merge (Section 5.1.1), 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...p are different. During odd (even) steps, odd (respectively even) numbered processors perform a bitonic merge-exchange with their right neighbor. Figure 13 illustrates this algorithm for p=4 and M=5.

5.2.2. 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, that was described in Section 3.1.2. Consider a network of p identical processors, where p is a power of 2, interconnected by two types of links (Figure 14):

(i) 2-way links, between pairs of adjacent processors: P_0P_1, P_2P_3,...;

(ii) 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.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

20 records in four processor memories after initial preprocessing

P0	17	9	8	3	2
P1	1	2	3	5	6
P2	19	12	8	5	0
P3	2	4	5	9	13

compare and exchange (P0, P1) and (P2, P3)

P0	1	2	3	3	2
P1	17	9	8	5	6
P2	2	4	5	5	0
P3	19	12	8	9	13

localized sort

P0	3	3	2	2	1
P1	17	9	8	6	5
P2	0	2	4	5	5
P3	8	9	12	13	19

P0	3	3	2	2	1
P1	17	9	8	6	5
P2	0	2	4	5	5
P3	8	9	12	13	19

compare and exchange P1 and P2

P0	3	3	2	2	1
P1	0	2	4	5	5
P2	17	9	8	6	5
P3	8	9	12	13	19

localized sort

P0	3	3	2	2	1
P1	0	2	4	5	5
P2	17	9	8	6	5
P3	8	9	12	13	19

P0	3	3	2	2	1
P1	0	2	4	5	5
P2	17	9	8	6	5
P3	8	9	12	13	19

compare and exchange (P0, P1) and (P2, P3)

P0	0	2	2	2	1
P1	3	3	4	5	5
P2	8	9	8	6	5
P3	17	9	12	13	19

localized sort

P0	0	1	2	2	2
P1	5	5	4	3	3
P2	5	6	8	8	9
P3	9	12	13	17	19

P0	0	1	2	2	2
P1	5	5	4	3	3
P2	5	6	8	8	9
P3	9	12	13	17	19

compare and exchange P1 and P2

P0	0	1	2	2	2
P1	5	5	4	3	3
P2	5	6	8	8	9
P3	9	12	13	17	19

localized sort

P0	0	1	2	2	2
P1	3	3	4	5	5
P2	5	6	8	8	9
P3	9	12	13	17	19

FIGURE 13: BLOCK ODD-EVEN SORT (20 records)

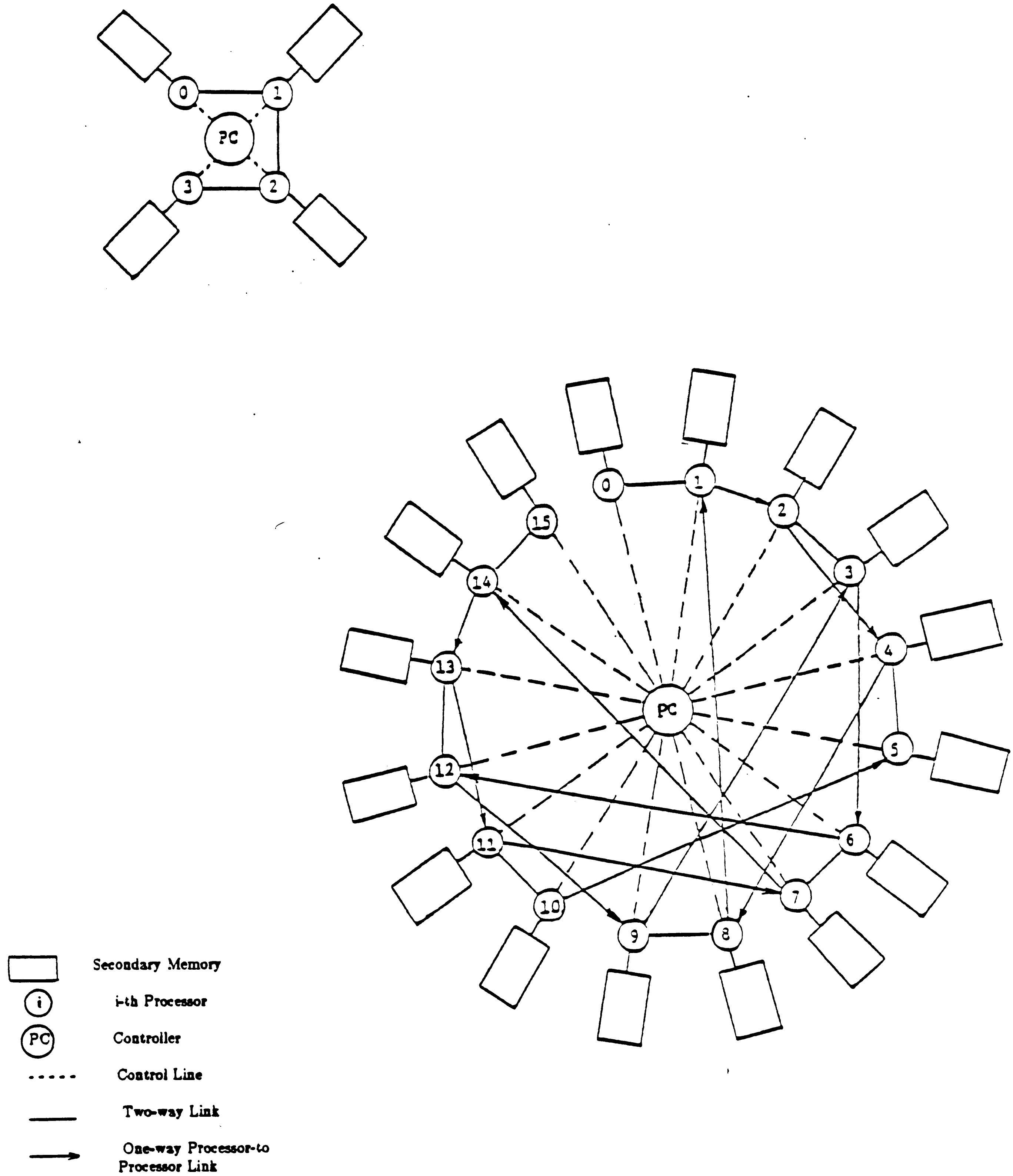


FIGURE 14: PROCESSORS' INTERCONNECTION FOR BLOCK-BITONIC SORT

(P=4 AND P=16)

memory of the reverse shuffle processor. Figure 15 illustrates this algorithm for p=4 and M=5.

6. 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 well accepted criteria for measuring their respective performance. However, the topic of external parallel sorting has not yet received adequate consideration.

In Section 5, we presented a number of parallel algorithms that can sort an array 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 with 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 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.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

STAGE 1

Step 1

P0	3	9	8	2	17
P1	6	2	1	3	5
P2	8	12	0	19	5
P3	5	9	13	4	2

A Perfect Shuffle

P0	3	9	8	2	17
P1	8	12	0	19	5
P2	6	2	1	3	5
P3	5	9	13	4	2

Step 2

P0	3	9	8	2	17
P1	8	12	0	19	5
P2	6	2	1	3	5
P3	5	9	13	4	2

A Perfect Shuffle

P0	3	9	8	2	17
P1	6	2	1	3	5
P2	8	12	0	19	5
P3	5	9	13	4	2

EXCHANGE [0, 1]
EXCHANGE [3, 2]

P0	2	3	3	2	1
P1	6	5	8	9	17
P2	19	12	8	9	13
P3	2	4	5	5	0

STAGE 2

Step 1

P0	2	3	3	2	1
P1	6	5	8	9	17
P2	19	12	8	9	13
P3	2	4	5	5	0

A Perfect Shuffle

P0	2	3	3	2	1
P1	19	12	8	9	13
P2	6	5	8	9	17
P3	2	4	5	5	0

EXCHANGE [0, 1]
EXCHANGE [2, 3]

P0	1	2	2	3	3
P1	19	13	12	9	8
P2	5	5	4	2	0
P3	5	6	8	9	17

Step 2

P0	1	2	2	3	3
P1	19	13	12	9	8
P2	5	5	4	2	0
P3	5	6	8	9	17

A Perfect Shuffle

P0	1	2	2	3	3
P1	5	5	4	2	0
P2	19	13	12	9	8
P3	5	6	8	9	17

EXCHANGE [0, 1]
EXCHANGE [2, 3]

P0	1	2	2	2	0
P1	5	5	4	3	3
P2	8	9	8	6	5
P3	17	9	12	13	19

STAGE 3 (parallel localized sort)

P0	0	1	2	2	2
P1	3	3	4	5	5
P2	5	6	8	8	9
P3	9	12	13	17	19

FIGURE 15: BLOCK BITONIC SORT

back to the mass storage device in sorted order.[3]

An early result on tape parallel sorting appeared in [Even 1974]. Recently in [Bitton 1981], several parallel sorting algorithms have been proposed for files residing on a modified moving-head disk.[3]

6.1. Parallel tape sorting

The sorting problem addressed in [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 2 methods for parallelizing the serial 2-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 pipelined-like algorithm. Both methods can be described briefly:

Method 1: each processor is assigned n/p records and 4 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 logn 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 2 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 4 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 2-way external merge-sort. However, the assumptions about the mass storage device do not take into consideration 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 implementation aspects.

---

[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.

6.2. Parallel disk sorting

The notion of 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 very 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 and Hsiao 1978], and in some cases, already built. The idea was pioneered by database machine designers, and prototypes were built in the framework of database research projects (see for example [Leilich et al. 1978]). Recently, commercial parallel readout disk are also made available for high-performance computers (for example, a 600-Mbyte drive with a 4-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 concurrently used. During execution of a single phase of a sorting algorithm, one drive can be utilized for reading and the other for writing.

In [Bitton 1981] a number of parallel external sorting algorithms and architectures are examined and analyzed. The mass storage device is modelled as a parallel read/write disk. The algorithm that displays the best performance is a parallel 2-way external merge-sort, termed the parallel binary merge algorithm. It is an improved variation of Method 1 in Section 6.1, achieved by parallelizing the second phase of this method.

When the number of output runs is 2^k, and $k>1$, 2^{k-1} processors can be used to perform concurrently the next step of the merge sort. 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

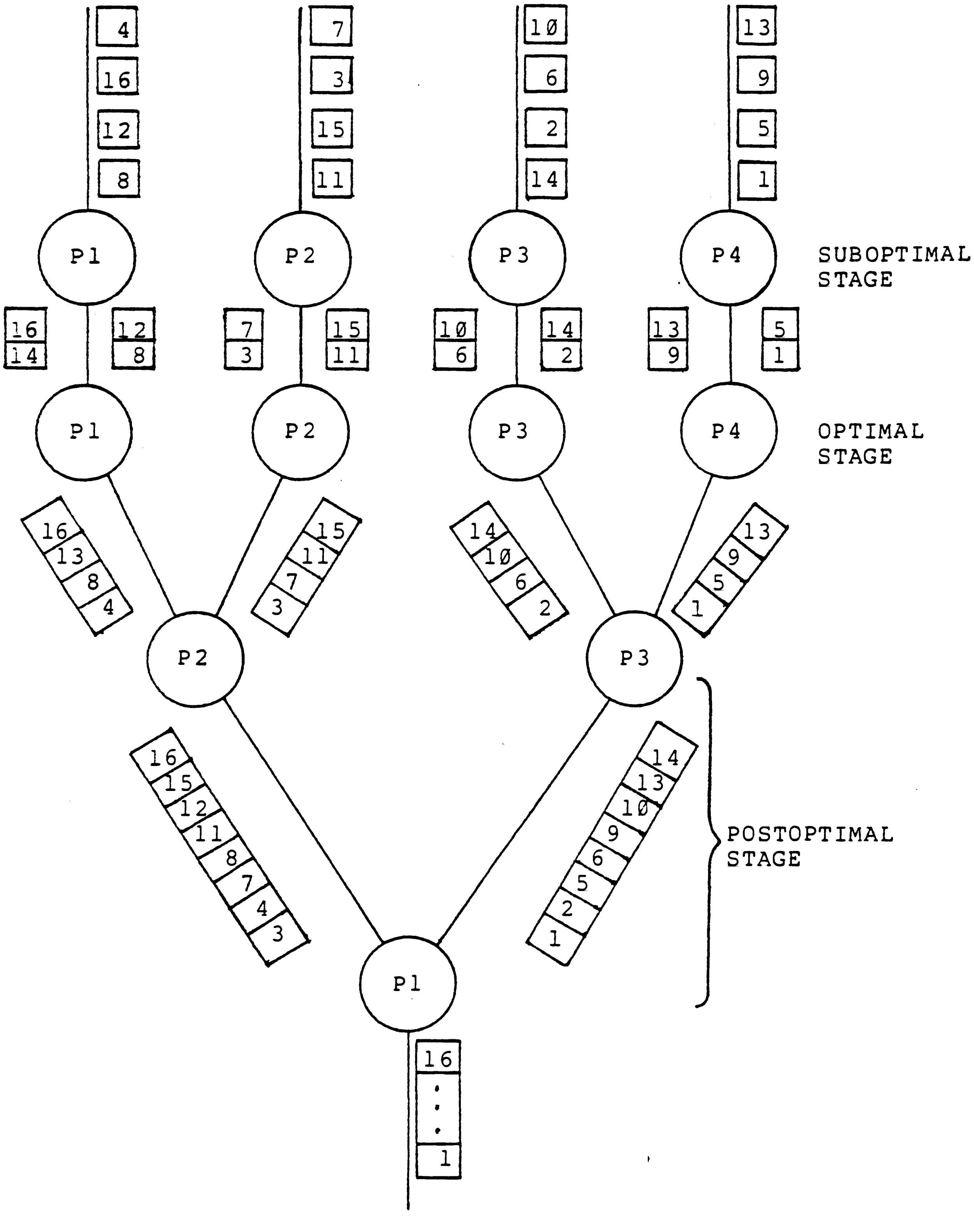


FIGURE 16: PARALLEL BINARY MERGE SORT

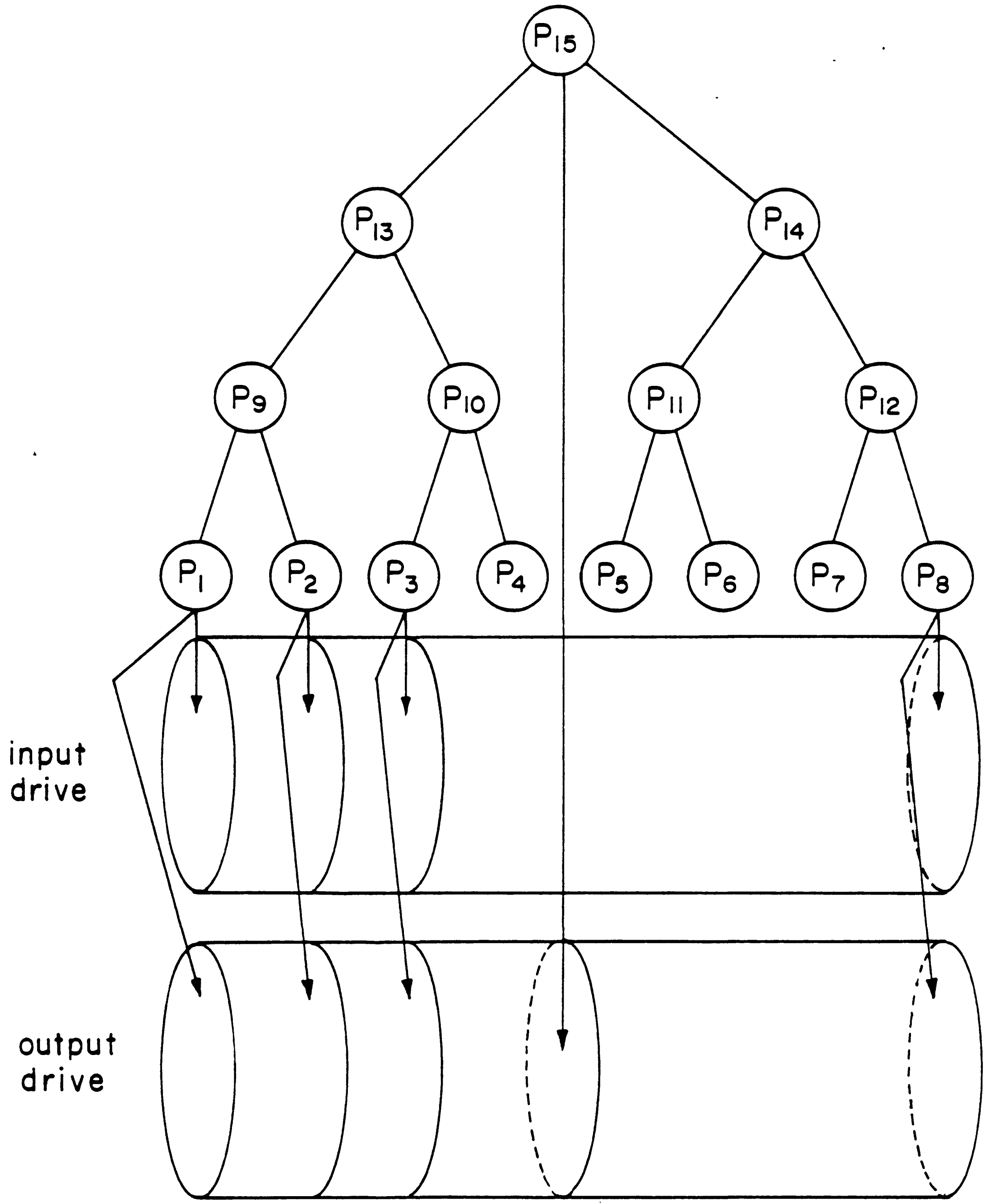


FIGURE 17: ARCHITECTURE FOR THE PARALLEL BINARY MERGE SORT

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.p, each processor will merge exactly 2 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 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 I/O in parallel, while reducing almost by half the number of processors that must actually do input/output.

6.3. Analysis of 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 modelling 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 because of the complexity of the I/O device.

Some indication of the parallel speedup that can be achieved by performing an external sort in parallel may be derived by assuming that the available I/O bandwidth is limited only by the number of processors. However, a more satisfactory analysis of parallel external sorting algorithms must take into consideration the constraints imposed by mass-storage technology. For example, for the parallel binary merge algorithm if the modified disk described in Section 6.2 is used for storage, the suboptimal stage can either be highly parallel, or almost sequential, depending whether or not the processors request data from several tracks on the same cylinder.

7. HARDWARE SORTERS

The high cost of sorting and the frequent need for it are motivating the design of "sort engines", that could eventually off-load the sorting function from general purpose CPU's. By implementing in hardware the sequence of comparison and move steps required by an efficient sorting algorithm, one could realize a low-cost, fast hardware device that would significantly lighten the burden on the CPU. Several alternative designs of hardware sorters have recently been proposed [Chen et al. 1978, Chung et al. 1980, Lee et al. 1981, Dohi et al. 1982, Yasuura et al. 1982, Thompson 1983], and preliminary evaluations seem to indicate that a VLSI implementation of sorting circuits 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 [Lee et al. 1981, Chung et al. 1980].

Another outcome of technology improvement might be that in the future, bubble chips will provide storage for large files, with on-chip sorting capabilities. In this case, the sorting function will 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, at this point, it is premature to determine whether or not advances in technology will be able to provide for intelligent mass-storage devices with sorting capabilities.

Hardware sorters, and in particular VLSI sorting circuits, are presently the focus of active research. Theoretical problems related to area-time complexity are also drawing considerable attention to VLSI sorting [Thompson 1980, Leiserson 1981, Thompson 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 a new research area 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 tradeoff.[4]

In the remainder of this section, we present an overview of alternative 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 tradeoffs between sorting speed and parameters related to I/O bandwidth, memory, and number of control lines required by on-chip sorters. In particular, we will describe the *rebound sorter* and the *up-down sorter,* that are clever pipeline versions of the odd-even transposition sort (Section 2.1), and a number of magnetic bubble sorters that integrate a sorting capability in bubble memory.

7.1. The rebound sorter

The *uniform ladder* [Chen76] 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 2.1). If the time for a bit to circulate within a loop is called a *period,* then N records (that 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* [Chen78]. The basic building block of the rebound sorter is the steering unit (Figure 18-a), 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 18-b). 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

[4] In a recent work (published after this paper was written), Thompson has surveyed thirteen 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.

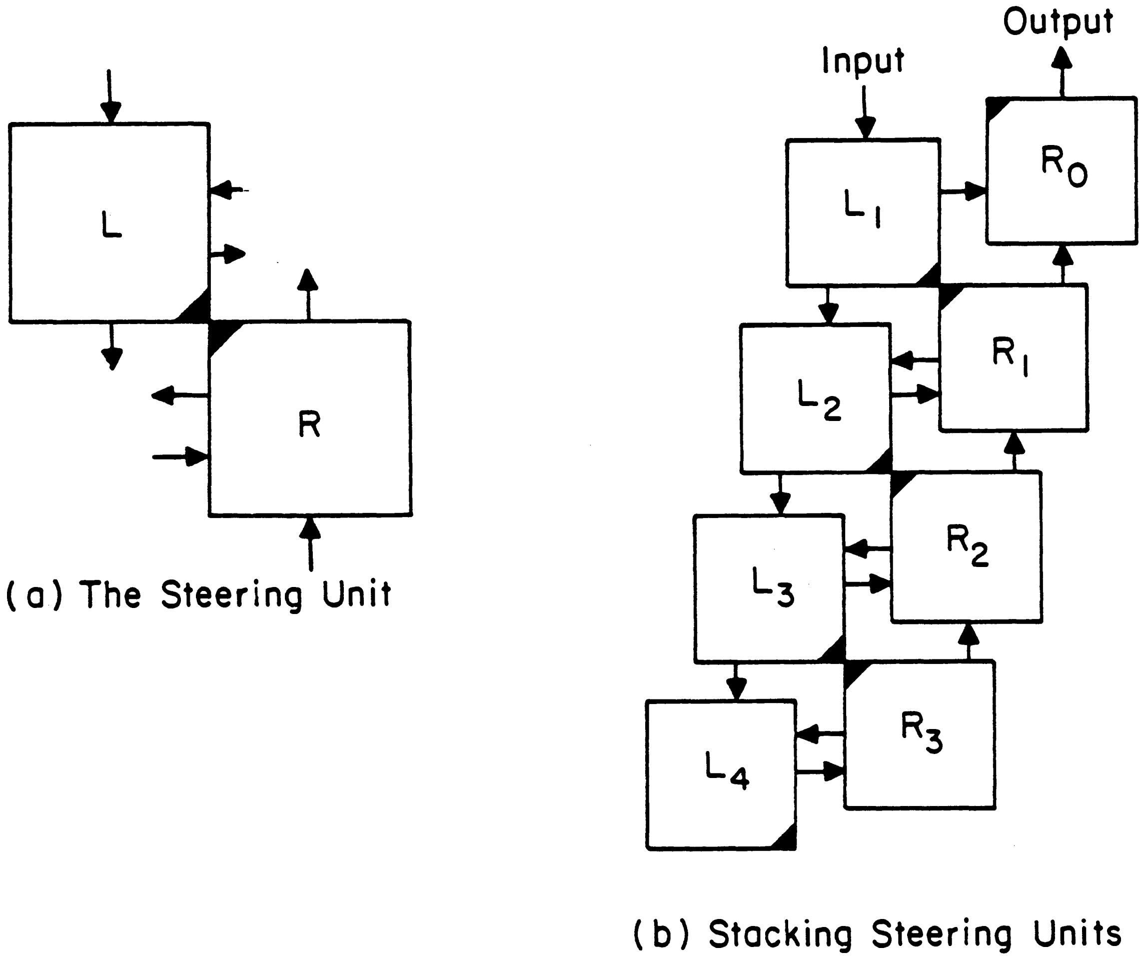


FIGURE 18: THE REBOUND SORTER [Chen et al. 1978]

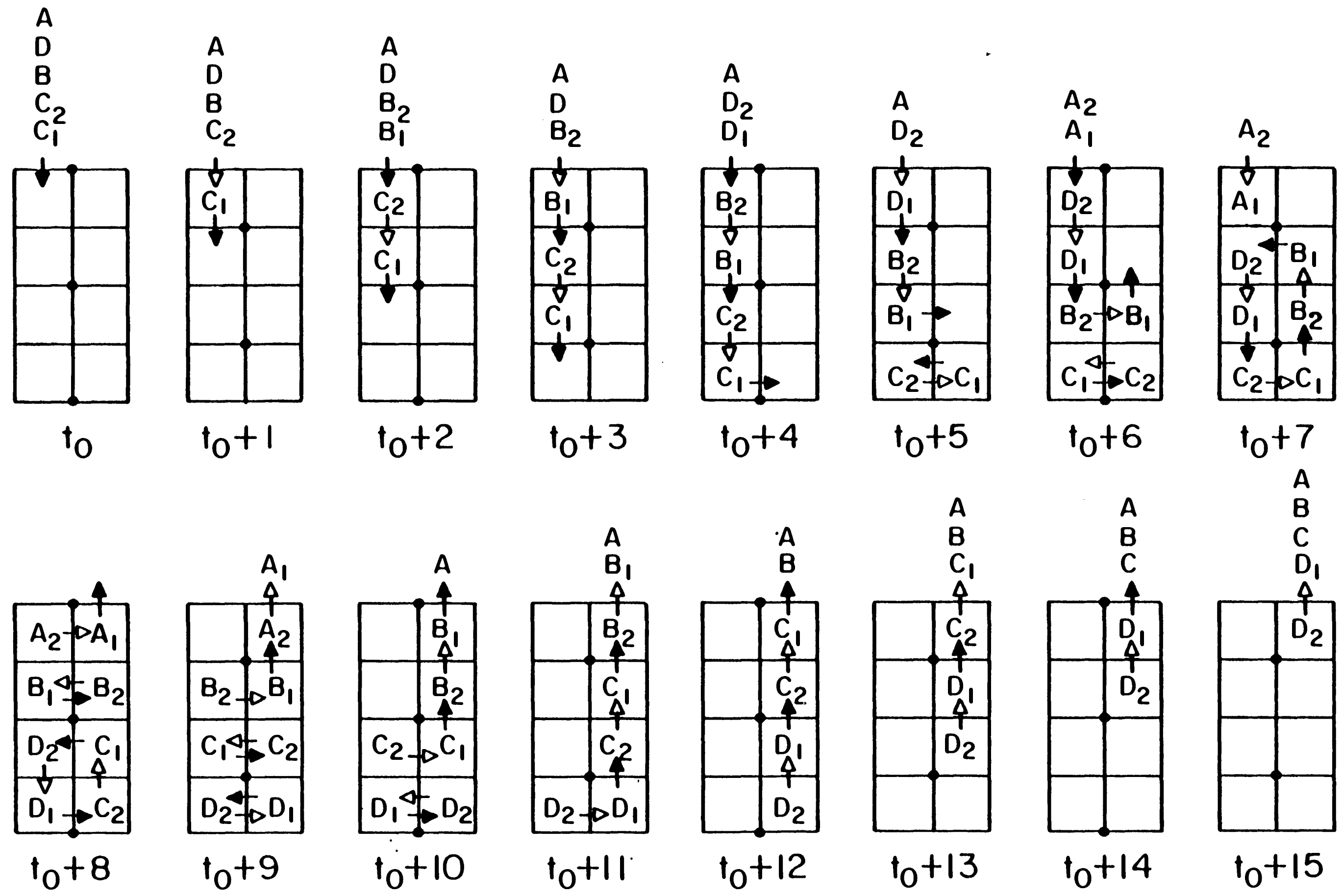


FIGURE 19: THE REBOUND SORT FOR 4 RECORDS [Chen et al. 1978]

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 2 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 that the complete sorted sequence is produced in the next N steps.

7.2. 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 the top of each other. It is assumed that the entire record fits in a cell (thus 2 records are stored in a compare/steer unit). The up-down sort is illustrated in Figure 20 for N=3 (3 compare/steer units sorting 6 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 2 keys (to the unit beneath them). During the upward output phase, each unit pops up the smaller of its 2 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. While the rebound sorter needs multiple control lines to individually activate switches of

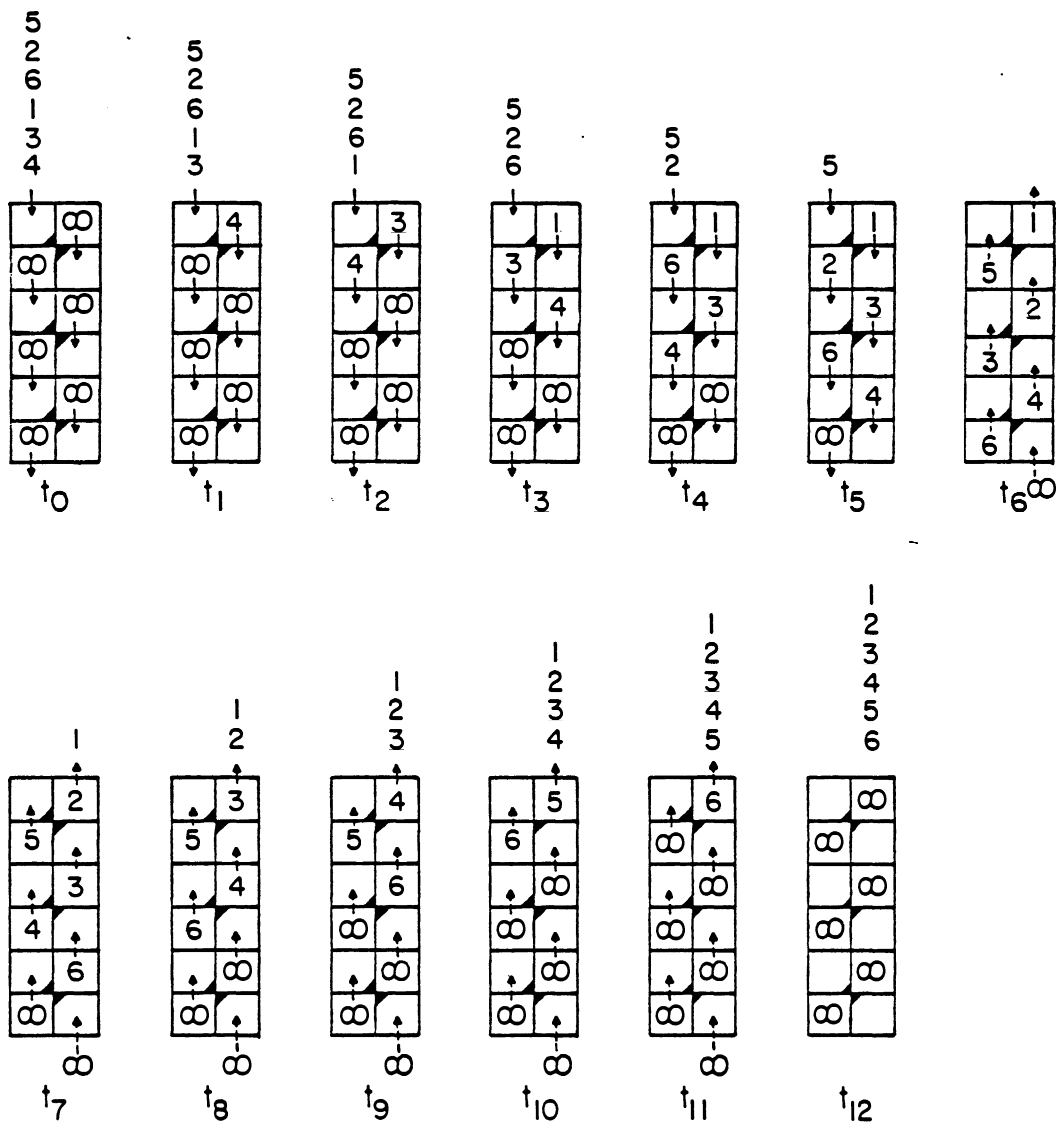


FIGURE 20: OPERATION OF THE UP-DOWN SORTER FOR 6 RECORDS

(only the keys are shown) [Lee et al. 1981]

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.

7.3. 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 in [Chung et al. 1980] also attach comparators to the bubble memory. However, in addition, they eliminate the I/O 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 for cost-effective mass-storage systems. If technology advances 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 I/O operations or CPU intervention.

Four models of intelligent bubbles are considered in [Chung et al. 1980], and for each model an alternative sorting scheme is proposed. The models differ by the size of the bubble loops and the number of switches required between the loops (to perform comparisons). The first two models implement a bubble sort and an odd-even transposition sort, respectively, while 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 21.a) 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 21.b). The (n-1) switches perform comparisons in parallel, according to the odd-even transposition scheme (Section 2.1). For the other two models (Figure 21.c, 21.d), the basic

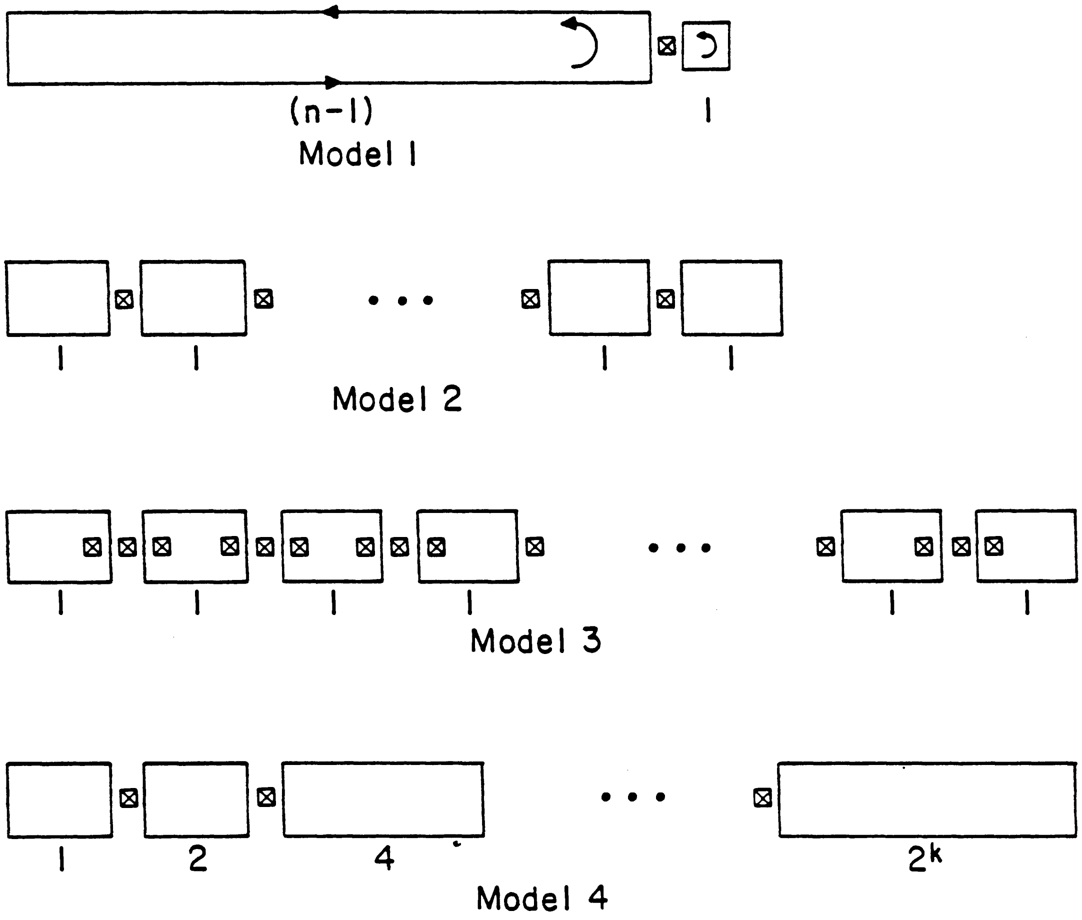


FIGURE 21: BUBBLE LOOP STRUCTURES FOR SORTING [Chung et al. 1980]

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 2 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 tradeoff is a higher hardware complexity (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 3 control states). On the other hand, a bitonic sorter sorts in time $O(n\log^2 n)$, but requires logn complex switches (each with 3logn control states). Thus, these detailed designs of bubble sorters provide an excellent illustration of cost-performance tradeoffs in sorting.

7.4. Summary and recent results

Several alternative designs of hardware sorters have recently been proposed, 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 designed 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 in [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 (as compared to either the sorting networks or the shared memory model algorithms). Although theoretical complexity bounds are being investigated for potentially faster VLSI sorters and important results have been achieved [Thompson 1983, Bilardi and Preparata 1983], the feasibility of fast, high-capacity VLSI sorters is still an open problem. 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, 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.

8. CONCLUSIONS AND OPEN PROBLEMS

Over the last decade, parallel sorting has been the focus of active research. A large number of parallel sorting algorithms are presently known, and new algorithms are still 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 which by systematically 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(logn), on a hypothetical O(n)-processor parallel machine. Systems designers, on the other hand, have investigated aspects such as 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 that 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 non-adaptive, 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 Sections 3.1), 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 3.2.1), 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 conflict resolution policies. 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 I, we have briefly summarized the asymptotic bounds of the main algorithms in both the network and the shared memory categories, in terms of processors utilized and execution time (the latter being essentially estimated as the number of parallel comparison steps required by the algorithms).

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 size problem. First, we have dealt with block-sorting algorithms, that 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 have introduced parallel external sorting algorithms, that address the problem of sorting in parallel an arbitrarily large mass-storage file.

Table I
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.	$n\log n$	$O(\log n)$

Besides these three classes of parallel sorting algorithms, we have described (in Section 7) 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 is the exploration of algorithms that exploit the characteristics of new storage technologies such as magnetic bubbles. The other, 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 sorting algorithms' theoretical computation time, while other aspects (such as technology constraints or data dependency) have received little consideration. Typically, algorithms have been developed for hypothetical computers, that utilize unlimited parallelism and space to solve the sorting problem in asymptotically minimal time. Today, it seems that the complexity of sorting either on networks or on shared memory parallel processors is well understood. Remaining open problems 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 mainly devoted to the theoretical complexity of parallel sorting, aspects related to the feasibility of parallel sorting will now be more systematically explored, in the context of current or near-term technology. 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 that 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), or 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. Clearly, communication, I/O costs and hardware complexity must be integrated in a comprehensive cost model, general enough to include a wide range of parallel processors architectures. In particular, an issue that has been largely ignored by previous research on parallel sorting is the initial cost of reading the source data into the processors' memories. While it is justified to ignore this issue when considering a serial, internal sorting algorithm, the situation is quite different with parallel processing. On a single processor, the source data is 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 I/O by all 64 processors. However, when a significantly larger number of processors is involved, only part of them will be able to perform I/O 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 would be no point in using a parallel sorting algorithm that requires only O(logn) time, if the startup cost to get the data in memory were O(n). Modelling the cost of I/O 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 undoubtably motivate further research in this direction.

9. REFERENCES

[Ajtai et al. 1983] Ajtai M., J. Komlos and E. Szemeredi, "An O(nlogn) Sorting Network," Proceedings 15th Annual ACM Symp. Theory Comput., April 1983.

[Alekseyev 1969] Alekseyev V.E., Kibernetica, 5, 5, 1969, pp. 99-103.

[Banerjee and Hsiao 1978] Banerjee J. and D.K. Hsiao, "Concepts and capabilities of a database computer," ACM Trans. Database Systems, 3, 4, December 1978.

[Batcher 1968] Batcher K.E., "Sorting networks and their applications," 1968 Spring Joint Computer Conference, AFIPS Proceedings, Vol. 32.

[Baudet and Stevenson 1978] Baudet G., and D. Stevenson, "Optimal sorting algorithms for parallel computers," IEEE Trans. Comput., C-27, 1, January 1978.

[Bentley and Kung 1979] Bentley J.L. and H.T. Kung, "A tree machine for searching problems," Proceedings 1979 International Conference on Parallel Processing, pp. 257-266, August 1979.

[Bilardi and Preparata 1983] Bilardi G. and F.P. Preparata, "A minimum area VLSI architecture for O(logn) time sorting," TR-1006, University of Illinois at Urbana-Champaign, November 1983.

[Bitton 1981] Bitton-Friedland D., "Design, analysis and implementation of parallel external sorting algorithms," Ph.D. Dissertation, December 1981, University of Wisconsin, Madison.

[Bitton and DeWitt 1983] Bitton D. and D.J. DeWitt, "Duplicate record elimination in large data files," ACM Trans. Database Systems, June 1983.

[Borodin and Hopcroft 1982] Borodin A. and J.E. Hopcroft, "Routing, merging and sorting on parallel models of computation," Proc. 14th Annual ACM Symp. on Theory of Computation, 1982.

[Bryant 1980] Bryant Ray, "External Sorting in a Layered Storage Architecture," IBM Research Center, Yorktown Heights, N.Y., 1980.

[Chen et al. 1978] Chen T.C., V.Y. Lum, and C.Tung, "The rebound sorter: an efficient sort engine for large files," Proc. 4th VLDB, September 1978.

[Chung et al. 1980] Chung K., F. Luccio and C.K. Wong, "On the complexity of sorting in magnetic bubble memory systems," IEEE Trans. Comput., C-29, July 1980.

[Dohi et al 1982] Dohi Y., A. Suzuki and N. Matsui, "Hardware sorter and its application to database machine," 9th Computer Architecture Conference, April 1982.

[Even 1974] Even S., "Parallelism in tape sorting," Commun. ACM, 17, 4, April 1974.

[Feng 1981] Feng Tse-yun, "A survey of interconnection networks," Computer, 14, 12, December 1981.

[Fishburn and Finkel 1982] Fishburn J.P and R.A. Finkel, "Quotient networks," IEEE Trans. Comput., C-31, 4, April 1982.

[Gavril 1975] Gavril F., "Merging with parallel processors," Commun. ACM, 18, 10, October 1975.

[Hirschberg 1978] Hirschberg, D.S., "Fast parallel sorting algorithms," Commun. ACM, 21, 8, August 1978.

[Hsiao et al 1980] Hsiao D.C. and Menon M.J., "Parallel record-sorting methods for hardware realization," TR OSU-CISRC-TR-80-7, The Ohio State University, Columbus, Ohio, July 1980.

[Knuth 1973] Knuth D.E., *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley, 1972.

[Kumar and Hirschberg 1983] Kumar M. and D.S. Hirschberg D.S., "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," IEEE Trans. Comput., C-32, March 1983.

[Lee et al. 1981] Lee D.T., H. Chang and K. Wong, "An on-chip compare/steer bubble sorter," IEEE Trans. Comput., C-30, June 1981.

[Leilich 1978] Leilich H.O., G. Stiege and H.C. Zeidler, "A search processor for database management systems," Proceedings 4th Conference on Very Large Database (1978).

[Leiserson 1981] Leiserson C. E., "Area-efficient VLSI computation," Ph.D. Dissertation, TR CMU-CS-82-108, October 1981.

[Muller and Preparata 1975] Muller D.E. and Preparata F.P., "Bounds for complexity of networks for sorting and switching," JACM, April 1975.

[Nassimi and Sahni 1979] Nassimi D. and S. Sahni, "Bitonic sort on a mesh connected parallel computer," IEEE Trans. Comput., C-27, 1, January 1979.

[Nassimi and Sahni 1982] Nassimi D. and S. Sahni, "Parallel algorithms to set up the Benes permutation network," IEEE Trans. Comput., C-31, 2, February 1982.

[Pease 1977] Pease M.C., "The indirect binary n-cube microprocessor array," IEEE Trans. Comput., C-26, 5, May 1977.

[Preparata 1978] Preparata F.P., "New parallel sorting schemes," IEEE Trans. Comput., C-27, 7, July 1978.

[Preparata and Vuillemin 1979] Preparata F.P. and J. Vuillemin, "The cube-connected-cycles," Proc. 20th Symp. on Foundations of Computer Science, 1979.

[Shiloach and Vishkin 1981] Shiloach Y. and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," J. Algorithms, 2, 1, March 1981.

[Siegel 1977] Siegel H.J., "The universality of various types of SIMD machine interconnection networks," Proceedings of the Fourth Annual Symposium on Computer Architecture, March 1977.

[Siegel 1979] Siegel H.J, "Interconnection networks for SIMD machines," IEEE Computer, 12, 6, June 1979.

[Stone 1971] Stone H.S., "Parallel processing with the perfect shuffle," IEEE Trans. Comput., C-20, 2, February 1971.

[Stone 1978] Stone H.S., "Sorting on Star," IEEE Trans. Software Eng., 4, 2, March 1978.

[Thompson and Kung 1977] Thompson C.D. and Kung H.T., "Sorting on a mesh-connected parallel computer," Commun. ACM, 20, 4, April 1977.

[Thompson 1980] Thompson C.D., "A complexity theory for VLSI," Ph.D. Dissertation, CMU-CS-80-140, August 1980.

[Thompson 1983] Thompson C.D., "The VLSI complexity of sorting," IEEE Trans. Comput., C-32, 12, December 1983.

[Valiant 1975] Valiant L.G., "Parallelism in comparison problems," SIAM J. Computing, Vol. 3, No. 4, September 1975.

[Vishkin 1981] Vishkin U., "Synchronized parallel computation," Ph.D. thesis, Technion Institute - Israel, 1981.

[Van Voorhis 1971] Van Voorhis D.C., Ph.D. Dissertation, Stanford Univ. , 1971.

[Yasuura H., N. Takagi and S. Yajima, "The parallel enumeration sorting scheme for VLSI," IEEE Trans. Comput., C-31,12, December 1982.