A TAXONOMY OF PARALLEL SORTING ALGORITHMS

by

David J. DeWitt, Dina B. Friedland
David K. Hsiao and Jaishankar Menon

Computer Sciences Technical Report #482

August 1982

A Taxonomy of Parallel Sorting Algorithms

by

David J. DeWitt[+], Dina B. Friedland[++]
David K. Hsiao[+++] and Jaishankar Menon[++++]

[+] Computer Science Department, University of Wisconsin,
Madison, Wisconsin

[++] Department of Applied Mathematics, Weizmann Institute,
Rehovot, Israel

[+++] Department of Computer Science, Naval Postgraduate School,
Monterery, California

[++++] IBM Research, San Jose, California

ABSTRACT

This paper provides a taxonomy of parallel sorting. The literature on parallel sorting is surveyed, and it is shown that most existing algorithms belong to one of two broad categories: the network sorting algorithms, or the shared memory model algorithms. Then, implementation and feasibility issues are addressed. Known parallel sorting algorithms (that typically require n processors to sort n records) are extended to "block sorting algorithms", that can sort n=M*p records with p processors. Finally, the notion of external parallel sorting is defined, and the topic of parallel file sorting is proposed as a new research direction.

1. INTRODUCTION

Sorting is defined as the process of rearranging a sequence of items into ascending or descending order. A basic sorting operation deals with items which are all key (that is, the order is defined on the items themselves). A more general sorting procedure deals with records, where one of the record fields or the concatenation of several fields constitute the key according to which the records are to be sorted.

The implications of dealing with record sorting and significant in terms of storage and data movement, since typically, a record contains several hundred bytes, while the key may only be a few bytes long.

A high percentage of computer resources are utilized for sorting. This is because sorting is often required, either to deliver to a user a well-organized output, or as an intermediate step in the execution of a complex algorithm. Another reason that explains this consumption of computer resources, is the fact that sorting is a time consuming operation, even when a very efficient sorting algorithm is used.

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 faster sorting algorithms.

Many serial sorting algorithms that perform in optimal time (that is, sort n items in time O(n logn)) are known. However, the introduction of parallel processing has added a new dimension to research on sorting algorithms. With the use of multiple processors, sorting time can be reduced, at least in theory, to O(logn). During the past decade, numerous results on parallel sorting have been published. In particular, Batcher's sorting networks [Batc68] exhibited a complexity of $O(\log^2 n)$; later, several optimal parallel sorting algorithms, of complexity O(logn) were developed for a theoretical parallel processor model [Hirs78, Prep78]. The most striking property of all these algorithms is perhaps, the very large number of processors that they require: typically, n processors are required to sort n elements.

This paper proposes a taxonomy of parallel sorting. The evolution of research on parallel sorting is analyzed - from the earliest sorting networks until the shared memory model algorithms. An attempt is made to classify all the existing parallel sorting algorithms, according to various criteria that include not only their efficiency, but also the architectural requirements that they rely upon. In addition, several directions for future research in the area of parallel sorting are identified. In particular, implementation issues are addressed, and the storage efficiency of various algorithms is considered.

The paper is organized as follows. In Section 2, we show that certain fast serial sorting algorithms can be parallelized but this tact 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 [Vali75, Gavr75], and the shared memory sorting algorithms [Hirs78, Prep78].

In Section 5, we define "block-sorting" parallel algorithms, that sort Mp elements with p processors, and identify two methods for deriving a block-sorting algorithm.

In Section 6, we briefly address the problem of sorting a large file in parallel. We show that previous research has mostly dealt with internal sorting algorithms, and propose external parallel sorting as a new research direction.

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. Another possibility is to compare a key to many other keys simultaneously. For example, in [Mull75], 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 this parallel scheme, 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 by 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 an optimal serial algorithm sorts n keys in time O(nlogn), the optimal speedup would be achieved when, using n processors, n keys are sorted in time O(logn). It does not, however, seem possible to achieve this bound by simply parallelizing one of the well-known optimal serial sorting algorithms. These algorithms appear to have several serial constraints that cannot be relaxed. On the other hand, parallelization of straight sorting methods (that is brute force methods requiring $O(n^2)$ comparisons) seems easier, but this approach does not lead to optimal parallel algorithms. 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 simple algorithm is used in the database Tree Machine [Bent79].

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 [Knut73, 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 [Knut78, p. 65).

This algorithm calls for a straightforward parallelization [Baud78]. 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.

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

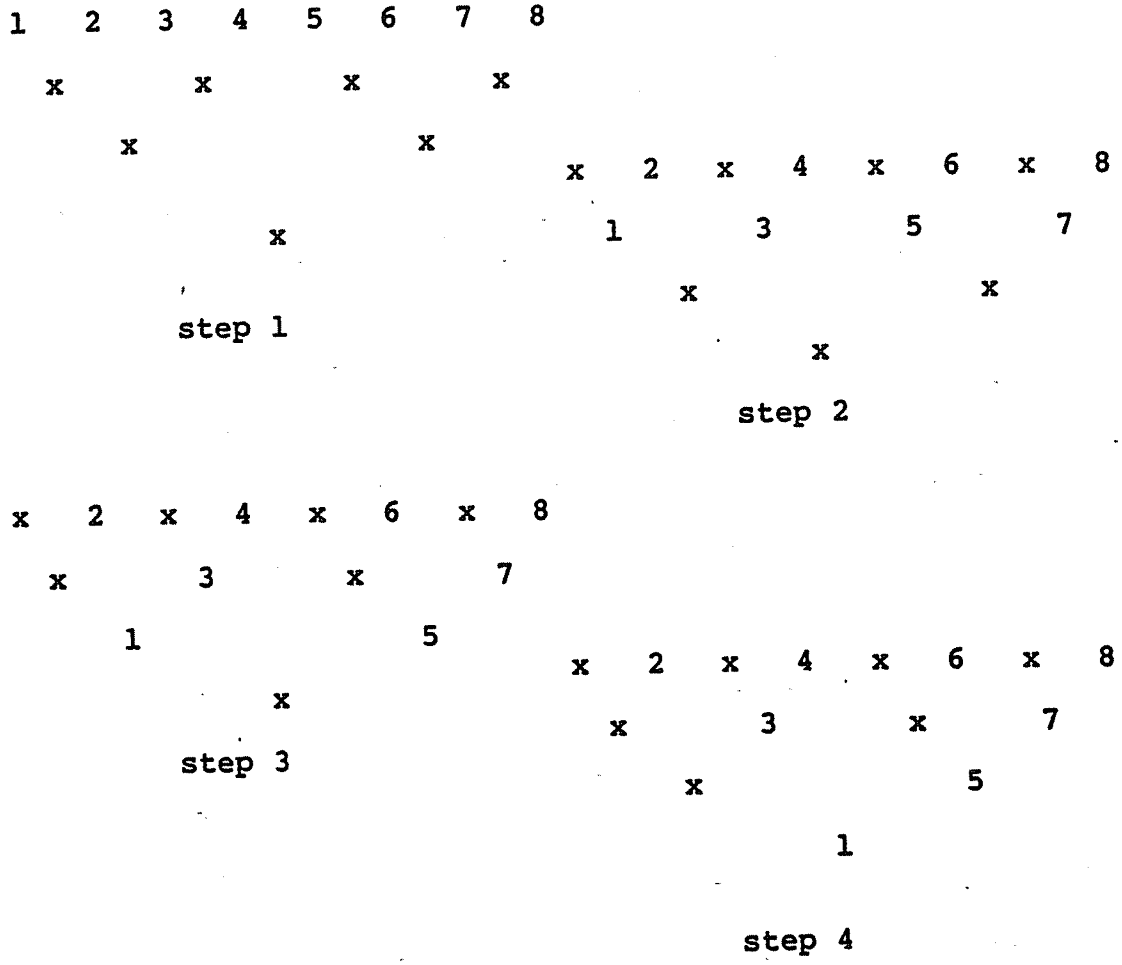


Figure 1. Tree selection sort

of the binary tree, a parallel tree-sort is obtained [Bent79].

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).

The speedup achieved with these simple parallelization schemes (log(n) for n processors) is not satisfactory, and many efforts have been made to achieve a higher performance. The first major improvement was reached with sorting networks, that sort n numbers in time $\log^2(n)$ and thus, achieve a speedup of n/log(n) [Batc68]. Later, Preparata [Prep78] demonstrated that the optimal bound (time: O(log(n), speedup : n) can be achieved with a theoretical model of n processors accessing a large shared memory.

Another important issue is the validity of the performance criteria by which parallel sorting algorithms have been previously evaluated. Clearly, assuming that the number of processors can be as large as the number of elements to be sorted, and counting the number of comparisons required by a parallel algorithm, is not satisfactory. Communication, I/O costs and hardware complexity must be incorporated in a comprehensive cost model, general enough to accommodate a wide range of parallel processors.

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 [Voor71, Batc68]. In Section 3.1, two types of sorting networks are described, that are respectively based on the odd-even and bitonic merge rules. In Section 3.2, we show that parallel sorting algorithms for SIMD (Single Instruction Multiple Data stream) machines can be derived from the bitonic network sort. In particular, we describe two bitonic sort algorithms for a mesh-connected processor [Thom77, Nass79].

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 [Batc68]. To design a fast sorting network, it is necessary to exploit the potential provided by a number of comparator modules of performing comparisons in parallel. Implementing a serial sorting algorithm on a network of comparators 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 [Batc68], 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 higher output line L and the maximum on its lower 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 NOR 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.

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 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

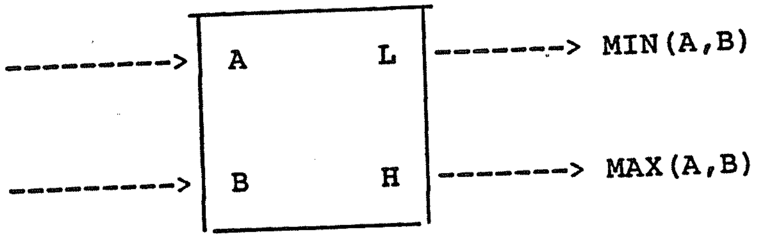


Figure 2. A comparison-exchange module

$(b_2, b_4, ...)$. Finally, the sequences $(c_1, c_2, ...)$ and $(d_1, d_2, ...)$ are merged into $(e_1, e_2, ..., e_n)$ by applying the following comparison-exchanges:

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

The resulting sequence will be sorted (for a proof, the reader is referred to [Knut73, 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 $(k^2-k+4)2^{k-2}-1$ [Batc68]. Several subsequent efforts have been able to reduce this number of comparators [Knut73], but only for particular cases (for example $k<5$).

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

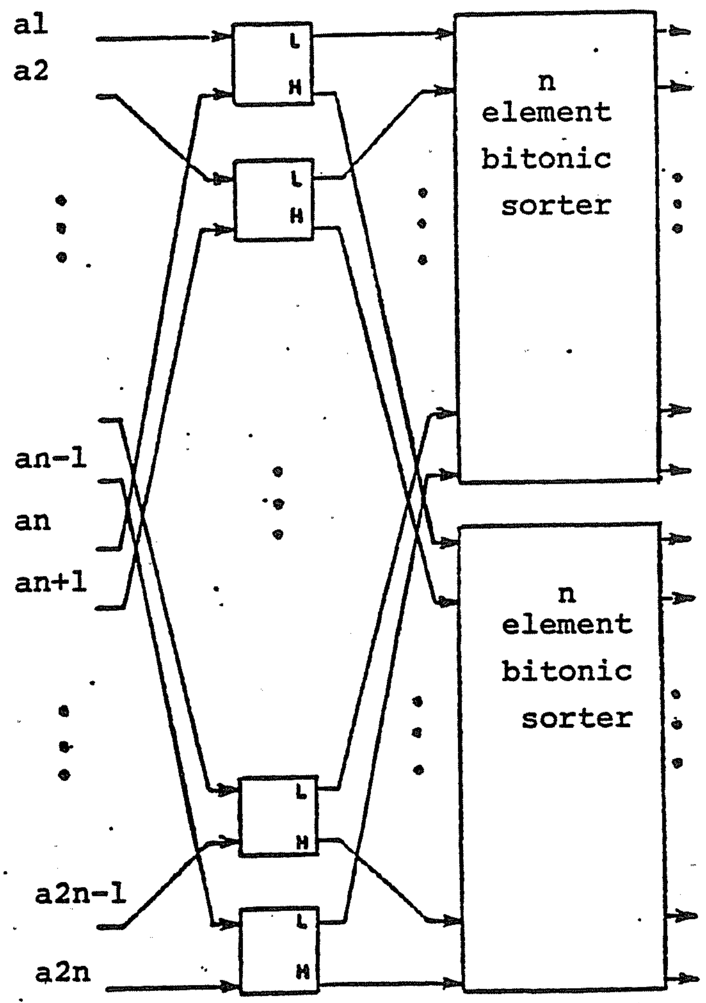


Figure 4. The iterative rule for the bitonic merge

that a bitonic sequence can be split into two bitonic sequences by performing a single step of comparison-exchanges. Let $(a_1, a_2, \ldots, a_{2n})$ be a bitonic sequence such that $a_1 \leq a_2 \leq \ldots \leq a_n$ and $a_{n+1} \geq a_{n+2} \geq \ldots \geq a_{2n}$. Then the sequences

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

and

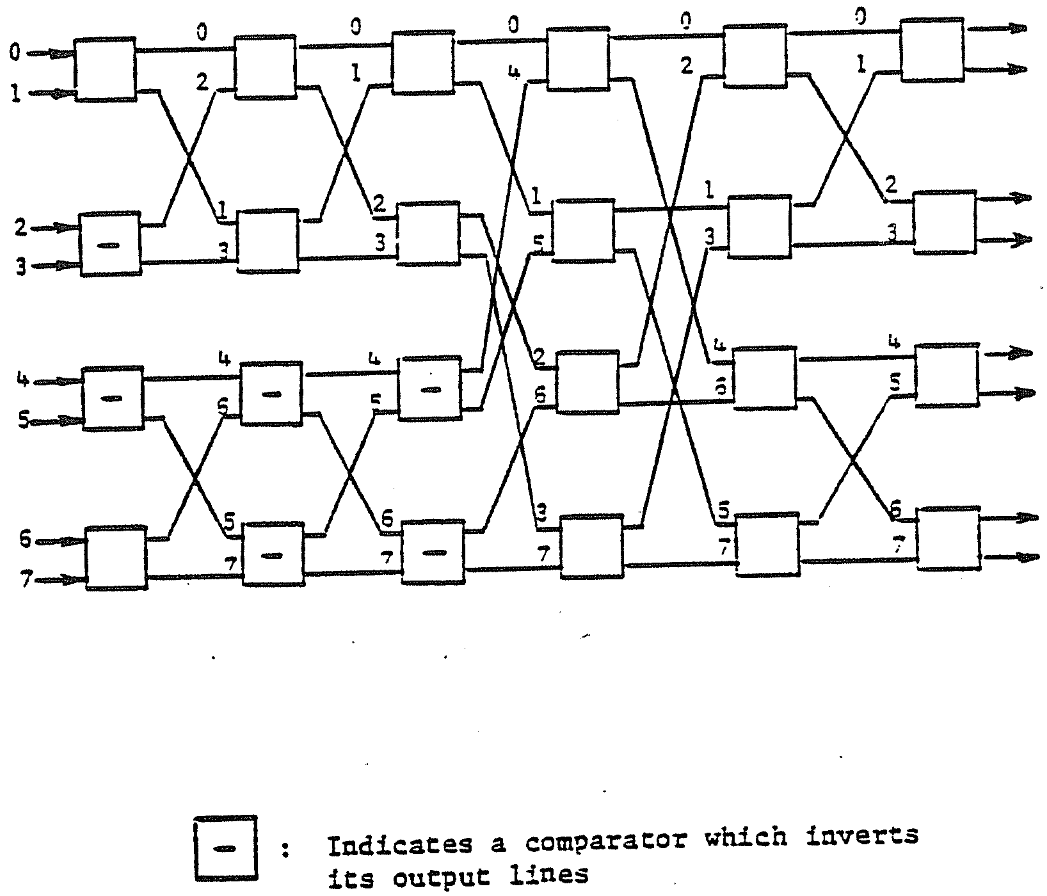
$$\max(a_1,a_{n+1}), \max(a_2,a_{n+2}), \ldots$$

are both bitonic.

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

To sort 2^k numbers using the bitonic iterative rule, we can successively 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.

Since the first version of the bitonic sort was presented, the algorithm has been considerably improved by the introduction of the <u>perfect shuffle</u> interconnection [Ston71]. Stone noticed



: Indicates a comparator which inverts its output lines

Figure 5. Batcher´s bitonic sort for 8 elements

that if the inputs were labelled by a binary index, then the indices of every pair of keys that enter a comparator at any step of the bitonic sort 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 $(\log n)-i$ steps of the ith stage do not exchange their inputs. Their only use is to shuffle their input lines.

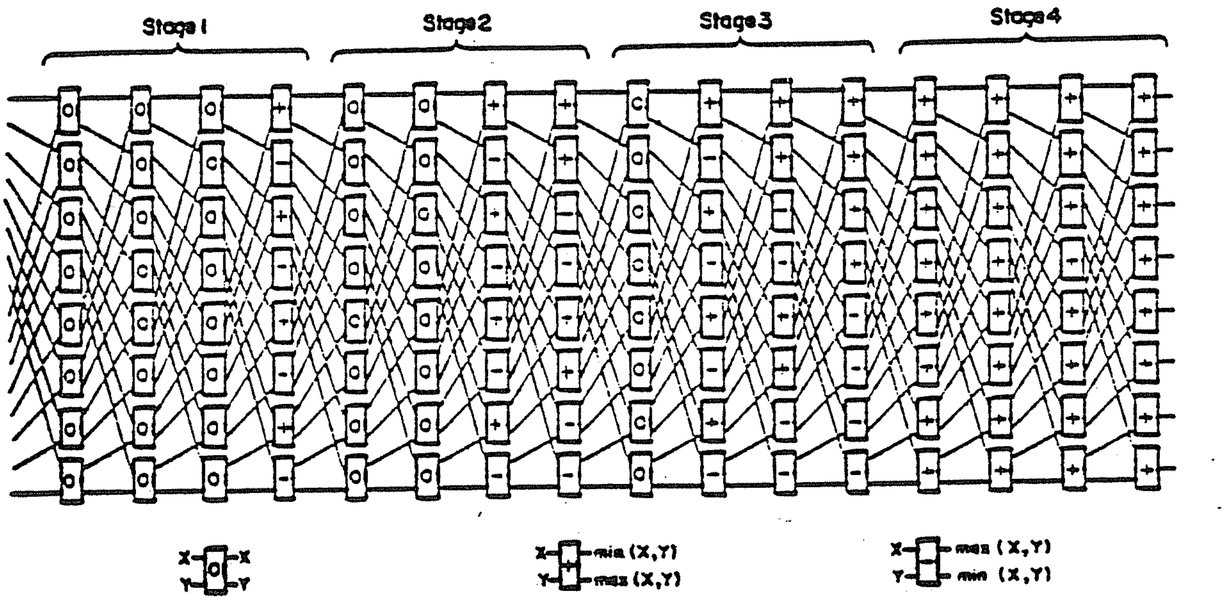


Figure 6. Stone's modified bitonic sort

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

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

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 $O(\log n/n)$ over the $O(n\log n)$ complexity of serial sorting. Therefore, it improves significantly the previous known bound of $O(n)$ for parallel speedup processors.

Siegel [Sieg77] has shown that the bitonic sort can be also performed by other types of networks in time $O(\log^2 n)$. Among the networks he considered, are the Cube and the Plus-Minus 2^i networks. Essentially, these networks can sort because they are able to simulate the perfect shuffle interconnection. Siegel proves that the simulation takes $O(\log^2 n)$ time, and thus, that sorting can also be performed within this time limit.

3.2. SORTING ON AN SIMD MACHINE

Sorting networks are characterized by their "non adaptivity" property. They perform the same sequence of comparisons regard-

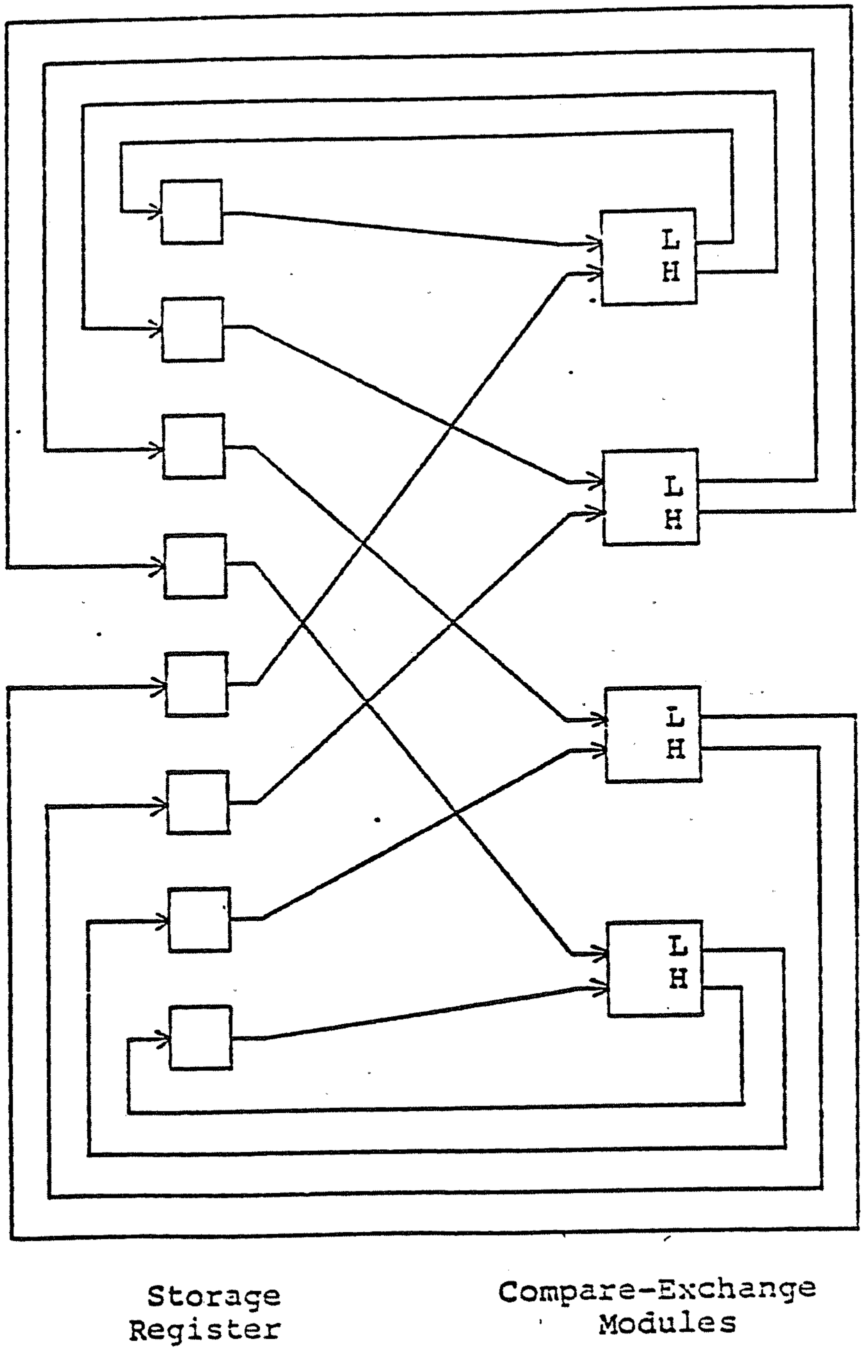
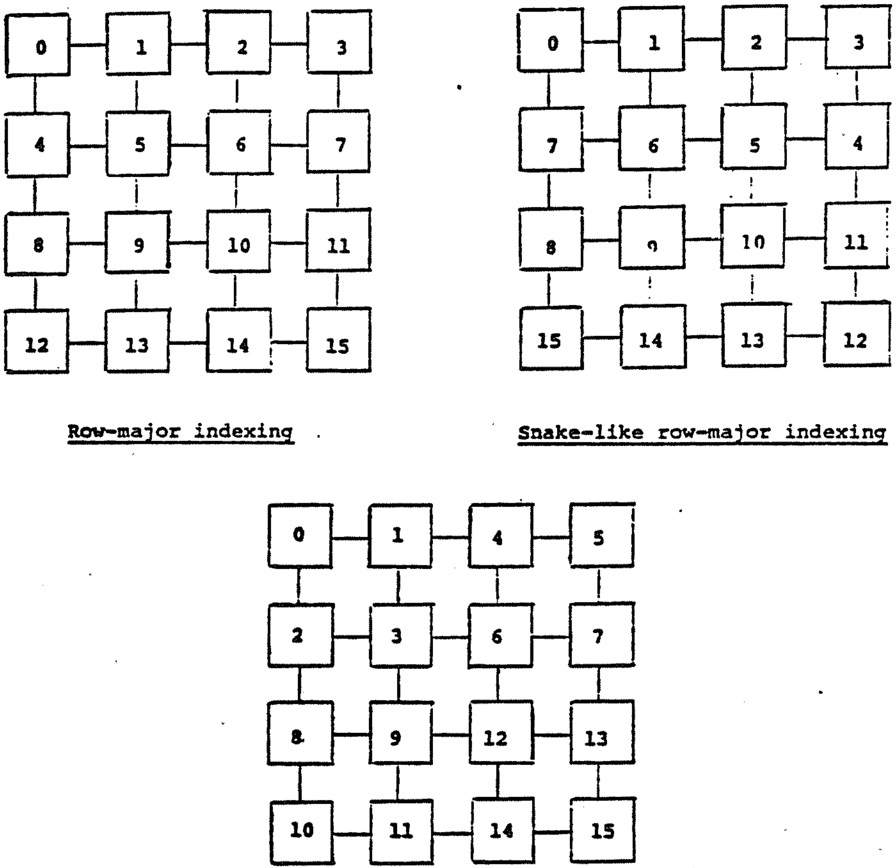


Figure 7. Stone's architecture for the bitonic sort

less 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. In particular, the sequence of comparisons and transfers to be executed by all the processors is determined when the sorting operation is initialized. Thus, a central controller can supervise the execution by broadcasting at each time step the appropriate compare-exchange instruction to the processors.

3.2.1. Sorting on an array processor

A different problem is considered in [Thom75], where the processors of an n by n mesh-connected multiprocessor are indexed according to a prespecified rule. The indexing rules considered are the row-major, the snake-like row-major, and the shuffled row-major rules (shown in Figure 8). Assuming that n^2 keys with arbitrary values are initially distributed so that exactly one key resides in each processor, the sorting problem consists of moving the ith smallest key to the processor indexed by i, for $i=1,...,n^2$. As with the sorting networks, parallelism is used to simultaneously compare pairs of keys, and a key is compared to only one other key 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.



Row-major indexing

Snake-like row-major indexing

Shuffled row-major indexing

Figure 8. Indexing Scheme [Thom75]

This computation model is SIMD since at each time unit a single instruction (compare or move) can be broadcast for concurrent execution by the set of processors specified in the instruction. The complexity of a method 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. This is true for any indexing scheme. In this sense a sorting algorithm which requires O(n) moves is optimal.

In [Thom75], two algorithms are presented that perform this array sort in O(n) comparisons and moves. 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. Recently, a third algorithm that sorts with row-major indexing with similar performance has been published [Nass79]. This algorithm is also an adaptation of the bitonic sort where the iterative rule is a merge of two dimensional arrays.

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. While the sorting network algorithms are based on comparison-exchanges of pairs, the shared-memory algorithms 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. The first enumeration type parallel sorting is a modified sorting network scheme, that sorts n elements with $O(n^2)$ processing elements. By embedding this type of network in a more general multiprocessor model in which the processors have access to a large shared memory, algorithms that are as fast, but require only $O(n)$ processors, can be obtained.

4.1. A MODIFIED SORTING NETWORK

In [Mull75], a very fast sorting network is proposed, that uses comparators as shown in Figure 9. This type of comparator has 2 inputs and one output. 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

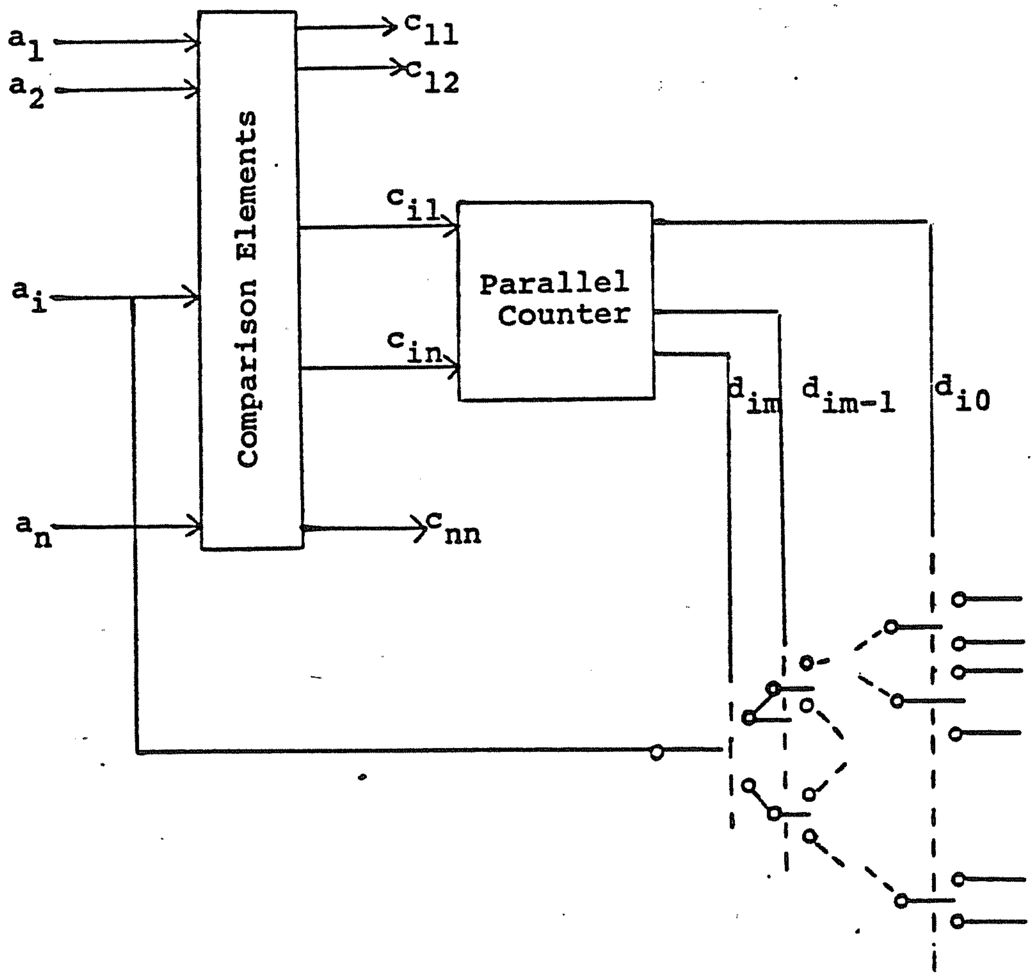


Figure 9. Muller´s sorting network

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 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 basic modules are more complex than comparison-exchange modules, and the network uses more of them), 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 scheme for parallel sorting. The idea of sorting by enumeration was exploited to develop other very fast parallel sorting algorithms [Hirs78, Prep78], that improve Muller'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. An optimal sorting algorithm should achieve the same speed with only O(n) processors (in order to exhibit a parallel speedup of order n).

4.2. FASTER PARALLEL MERGE METHODS

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 [Vali75] presents an inductive 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). On the other hand, Gavril [Gavr75] 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 $p \leq n \leq m$. 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 algorithms assume a shared memory model. That is, all the processors utilized can simultaneously access elements of the initial data, or intermediate computation results.

4.3. BUCKET SORTS

Hirschberg´s algorithm [Hirs78] is a "bucket sort" that sorts n elements with n processors in time O(logn), provided that the numbers to be sorted are in the range $\{0,1,\ldots,m-1\}$. A side effect of this algorithm is that duplicate numbers are eliminated. If memory conflicts were ignored, 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 realism 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 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 achieve the optimal time complexity O(logn), are described in [Prep78]. 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 [Vali75], 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).

Despite the improvement achieved by eliminating memory conflicts, these algorithms are still not very realistic. 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 make no account for the overhead associated with the reallocation of processors during the various stages of the sort algorithm.

However, the results achieved are of major theoretical importance, and the methods used demonstrate the intrinsic parallel nature of certain sorting procedures. Furthermore, basic ideas in these algorithms can inspire the design and implementation of realistic sorting methods.

5. BLOCK SORTING ALGORITHMS

For all the parallel sorting algorithms described in previous sections, the problem size (that is, the number of records 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. The first reason is that, in a switching network, the processors are simple hardware boxes, which compare and exchange their two inputs. The second is that, since the number of processors is determined by the number of input lines to the network, it can never be prohibitively expensive.

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.

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=[n/p] 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

(i) the block residing in each processor´s memory constitutes a sorted sequence of length M, S_i.

(ii) 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>>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 step 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 [Baud78]; the other on a bitonic merge

[Hsia80]. 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.

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. Thus, a two-way merge-split step requires a local memory of size at least 4M. The contents of a processor´s memory before and after a two-way merge-split is shown in Figure 10. As indicated by the following code for the Procedure Merge, it can be executed in parallel, in SIMD mode, by several processors. The procedure is initiated after 2 sorted sequences of length M have been stored in a processor´s local memory, in two input buffers I1[1..M] and I2[1..M].

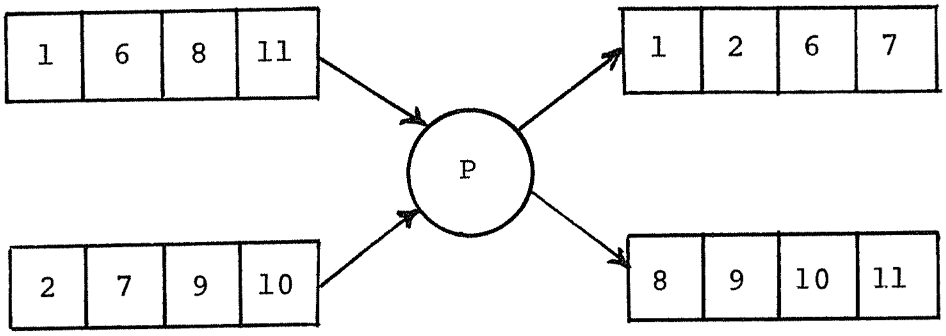


Figure 10. Merge-split based on 2-way merge

```
Procedure Merge;
begin
  I1[M+1]:=I2[M+1]:=BIG; {add a large number at the end
          of the source sequences, to stop the merge process
          when one sequence is exhausted}
  i:=j:=1;
  for k:=1 to 2M do
  begin
     if I1[i] < I2[j] do
     begin
        O[k] := I1[i];
        i:=i+1;
     end
     else
     begin
        O[k] :=  I2[j];
        j:=j+1;
     end;
 end;
end Merge;
```

After the processors have completed the parallel execution of the Procedure Merge, they split their output buffer O[1..2M], 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 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, by odd and even processors alternately, as follows:

```
Procedure Step(i); {as executed by odd (respectively even) processor}
Begin
   if i is odd (respectively even) then
   begin
        for j:=1 to M do I1[j] := O[j];
        receive M records from right neighbor;
        merge;
        send higher M records to right neighbor;
   end;
end Step;
```

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((logp)+1)(logp) 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((logp)+1)(logp) 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.

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. 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 based on 2-way merge. Thus a 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.

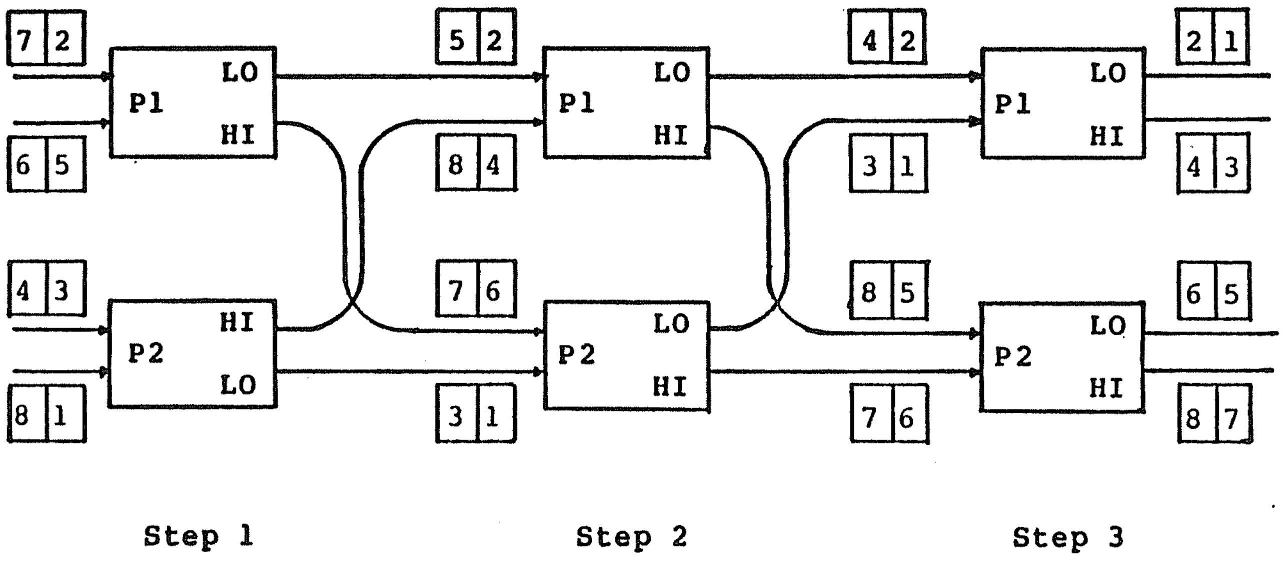


Figure 11. Block Bitonic Sort, based on 2-way merge

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 $(x_1, x_2, \ldots, x_M)$ are stored in increasing order in P_i's memory, and the M records $(y_1, y_2, \ldots, y_M)$ are stored in decreasing order in P_j's memory. Let P_j send y_1 to P_i. P_i then compares x_1 and y_1, keeps the lower of the 2 and sends back to P_j the higher record. This procedure is then repeated for $(x_2,y_2),\ldots,(x_M,y_M)$. 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 [Alek69, Knut73]. 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. In general, let R[i,1..M] and R[j,1..M] respectively denote a sorted sequence in processors P_i and P_j memories. Then, P_i and P_j perform a bitonic merge-exchange by executing con-

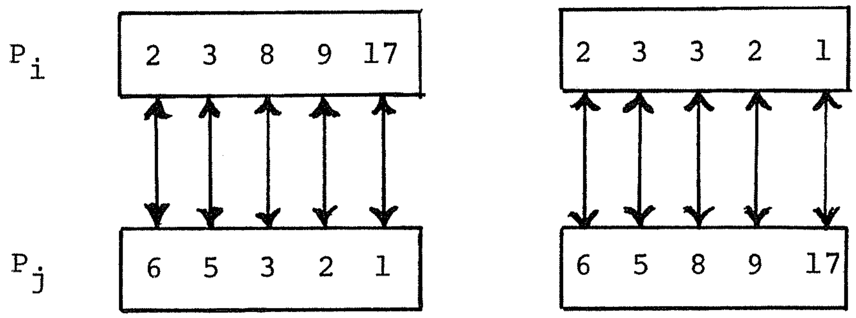


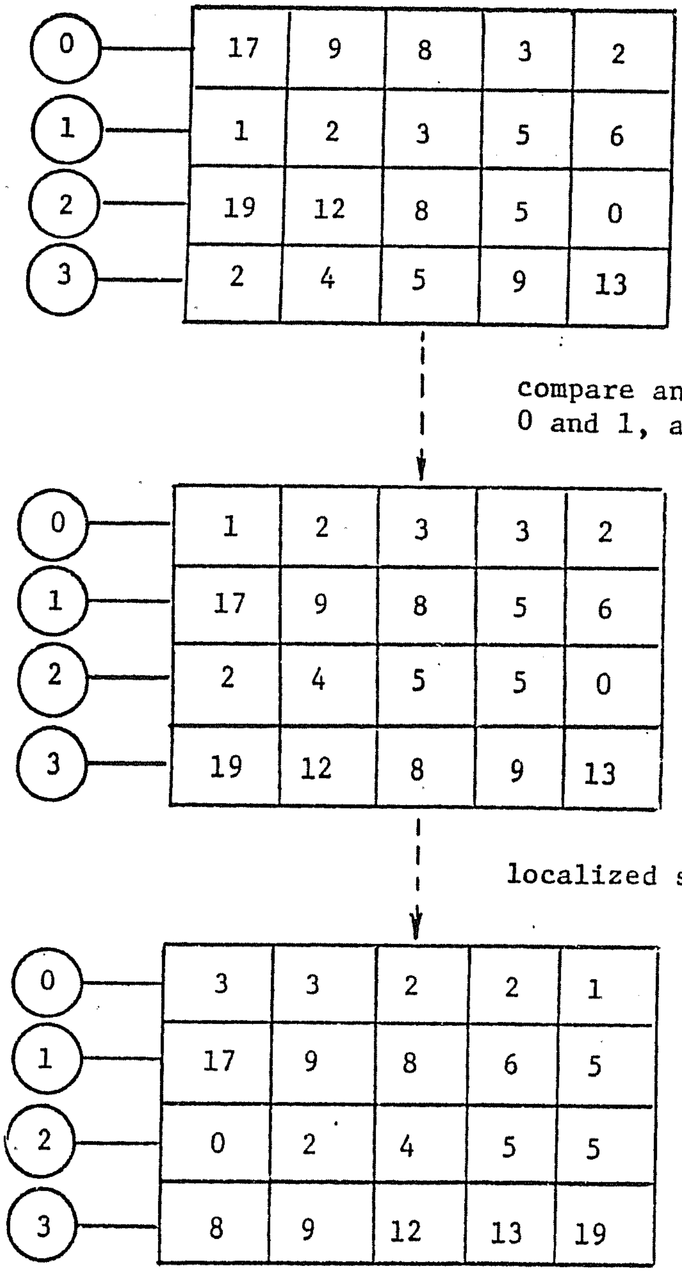
Figure 12. A bitonic merge-exchange step

currently the following Send() and Receive() procedures:

```
procedure Send(j); {as executed by $P_i$}
begin
   count:=1;
   while count≤M
   begin
      send R[i,count] to $P_j$;
      wait for return record from $P_j$;
      count:=count+1;
   end;
end;

procedure Receive(i); {as executed by $P_j$}
begin
    count:=1;
    while count≤M do
    begin
       wait for record R[i,count] from $P_i$;
       compare with own R[j,count];
       if R[j,count]<R[i,count] then
           interchange R[i,count] and R[j,count];
       send R[i,count] to $P_i$;
       count:=count+1;
    end;
end Receive;
```

The bitonic merge-exchange 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. Another advantage of this method is that the comparisons (of pairs of records) and the transfers are interleaved. 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 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,



20 records in four processor memories after initial preprocessing

0	17	9	8	3	2
1	1	2	3	5	6
2	19	12	8	5	0
3	2	4	5	9	13

compare and exchange between 0 and 1, and 2 and 3

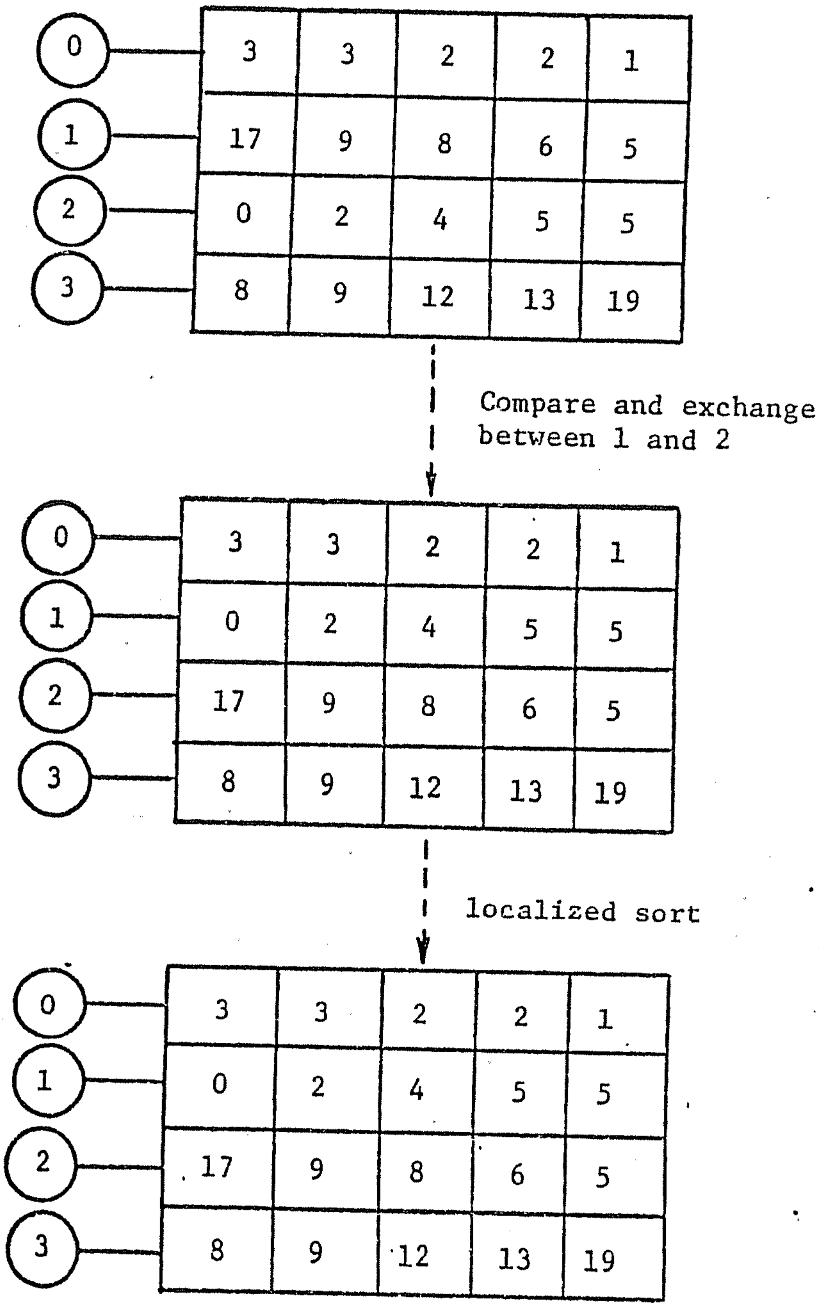
0	1	2	3	3	2
1	17	9	8	5	6
2	2	4	5	5	0
3	19	12	8	9	13

localized sort

0	3	3	2	2	1
1	17	9	8	6	5
2	0	2	4	5	5
3	8	9	12	13	19

Figure 13-a. Block odd-even sort for 20 records

Figure 13-b. Block odd-even sort

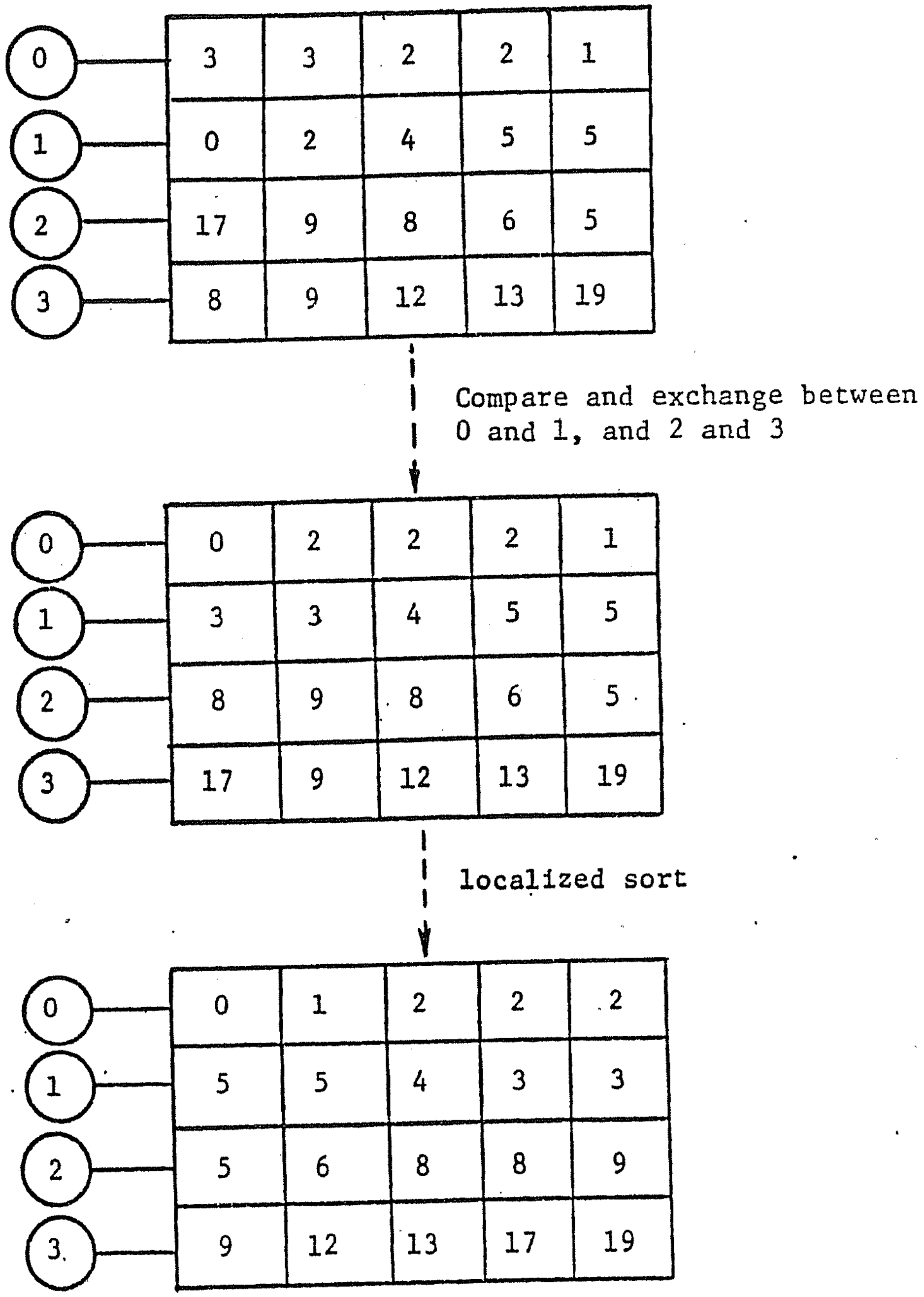


0	3	3	2	2	1
1	17	9	8	6	5
2	0	2	4	5	5
3	8	9	12	13	19

0	3	3	2	2	1
1	0	2	4	5	5
2	17	9	8	6	5
3	8	9	12	13	19

0	3	3	2	2	1
1	0	2	4	5	5
2	17	9	8	6	5
3	8	9	12	13	19

Figure 13-c. Block odd-even sort



0	3	3	2	2	1
1	0	2	4	5	5
2	17	9	8	6	5
3	8	9	12	13	19

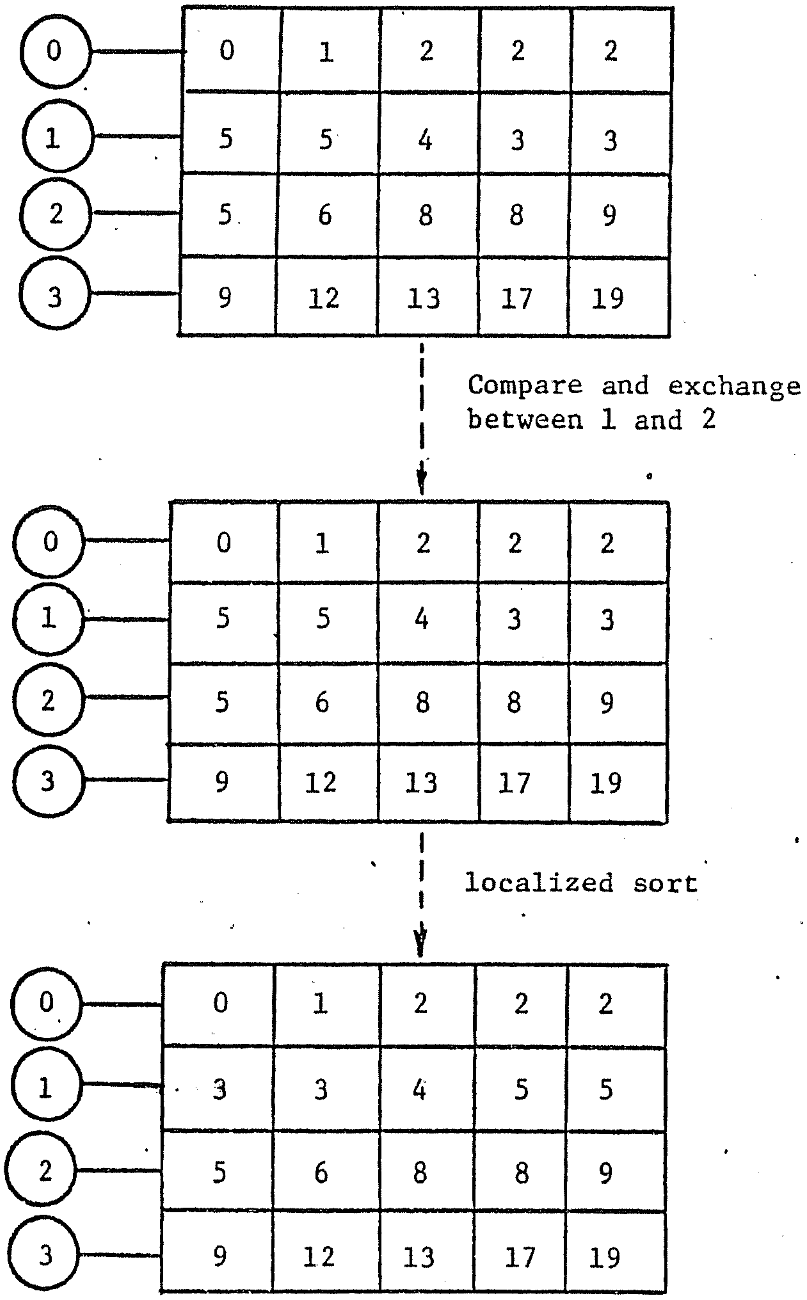
Compare and exchange between
0 and 1, and 2 and 3

0	0	2	2	2	1
1	3	3	4	5	5
2	8	9	8	6	5
3	17	9	12	13	19

localized sort

0	0	1	2	2	2
1	5	5	4	3	3
2	5	6	8	8	9
3	9	12	13	17	19

Figure 13-d. Block odd-even sort



0	0	1	2	2	2
1	5	5	4	3	3
2	5	6	8	8	9
3	9	12	13	17	19

Compare and exchange between 1 and 2

0	0	1	2	2	2
1	5	5	4	3	3
2	5	6	8	8	9
3	9	12	13	17	19

localized sort

0	0	1	2	2	2
1	3	3	4	5	5
2	5	6	8	8	9
3	9	12	13	17	19

a serial sorting algorithm that permutes the records in place 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, so that sorting them requires at most (M/2)*log(M) 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. Thus, the algorithm executed by P_i is the following:

```
Odd-Even Sort; {as executed by all processor}
begin
    for step:=1 to p do Exchange(step);
end Odd-Even-Sort.

Procedure Exchange(step); {as executed by P_i}
begin
    if step is even and i=0 then stop;
    if step is even and p is odd and i=p-1 then stop;
    if step is odd and p is odd and i=p-1 then stop;
    if step is odd then
       if i is even then j:=i+1
          else j:=i-1;
    if step is even then
       if i is even then j:=i-1
          else j:=i+1;
    if j:=i+1 then
    begin
       local sort in increasing order;
       send(j);
    end
    else
    begin
       local sort in decreasing order;
       receive(j);
    end
end Exchange.
```

The algorithm is illustrated in Figure 13 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 using the following algorithm:

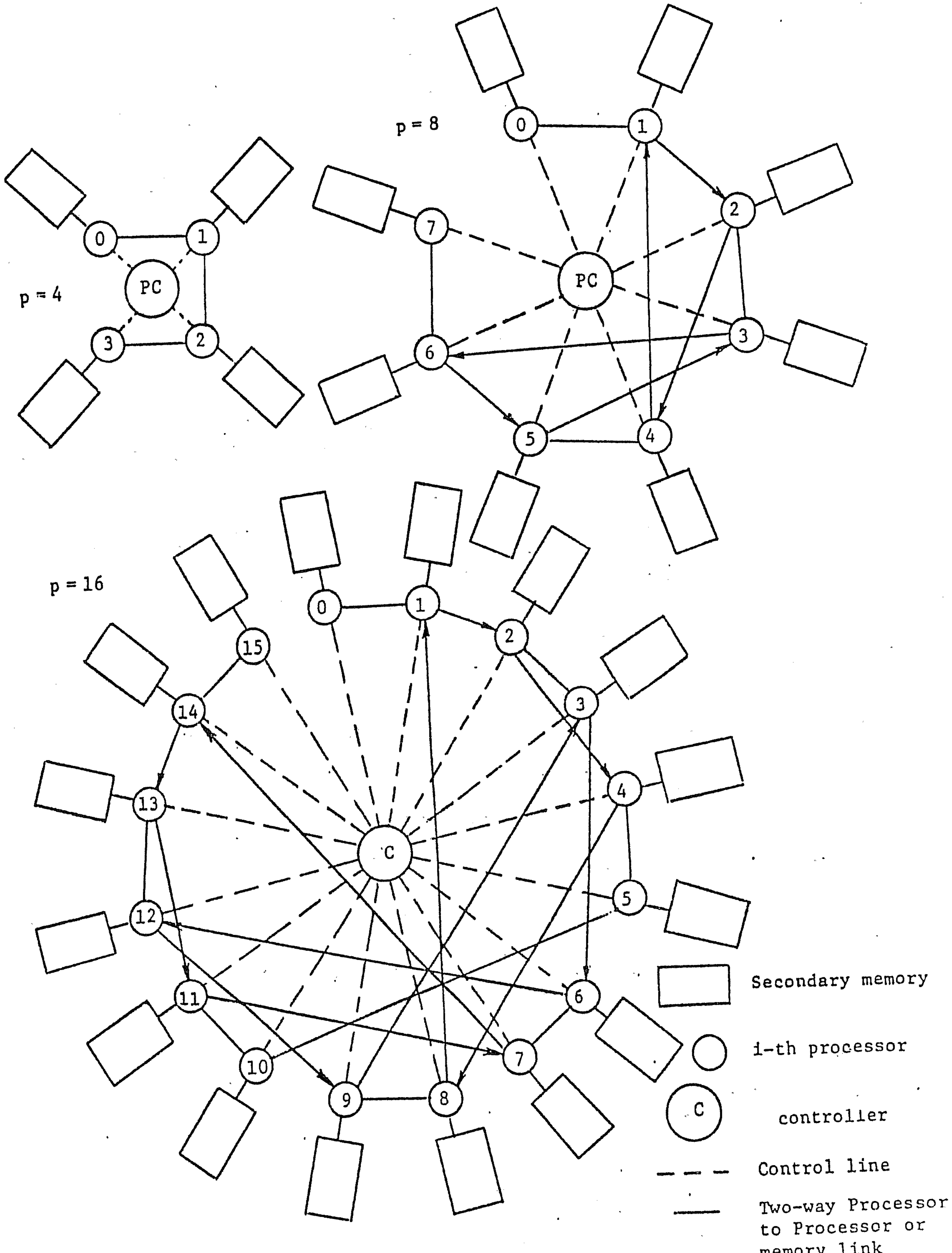


Figure 14.

Interconnection of processors for block bitonic sort

```
Bitonic-Sort;
begin
   stage := 1;
   while stage $\leq$ log(p) do
   begin
      step:=1;
      while (step $\leq$ log(p))do
      begin
         shuffle;
         if step > (log(p) - stage) then
            exchange(stage,step);
         step:=step+1;
      end;
   stage:=stage+1;
   end;
   local-sort in increasing order;
end.
```

Processor P_i executes the "B-Exchange" procedure as follows:

```
Procedure B-Exchange(stage,step);
begin
  q := step - log(p) + stage;
  r := i mod($2^{q+1}$);
  if r is even and r < $2^q$ then
  begin
       local-sort in increasing order;
       Send(i-1);
  end
  else
  if r is even
  begin
       local-sort in decreasing order;
       Receive(i+1);
  end
  else
  begin
       local-sort in decreasing order;
       Receive(i-1);
  end
end {Procedure B-exchange}
```

In procedure "Shuffle", each processor sends the records that were in its memory, in order, to the corresponding location of the shuffle processor´s memory and receives the records that were in the memory of the reverse shuffle processor. The procedures Send() and Receive() have been defined earlier (Section 5.2.1). Figure 15 illustrates this algorithm for p=4 and M=5.

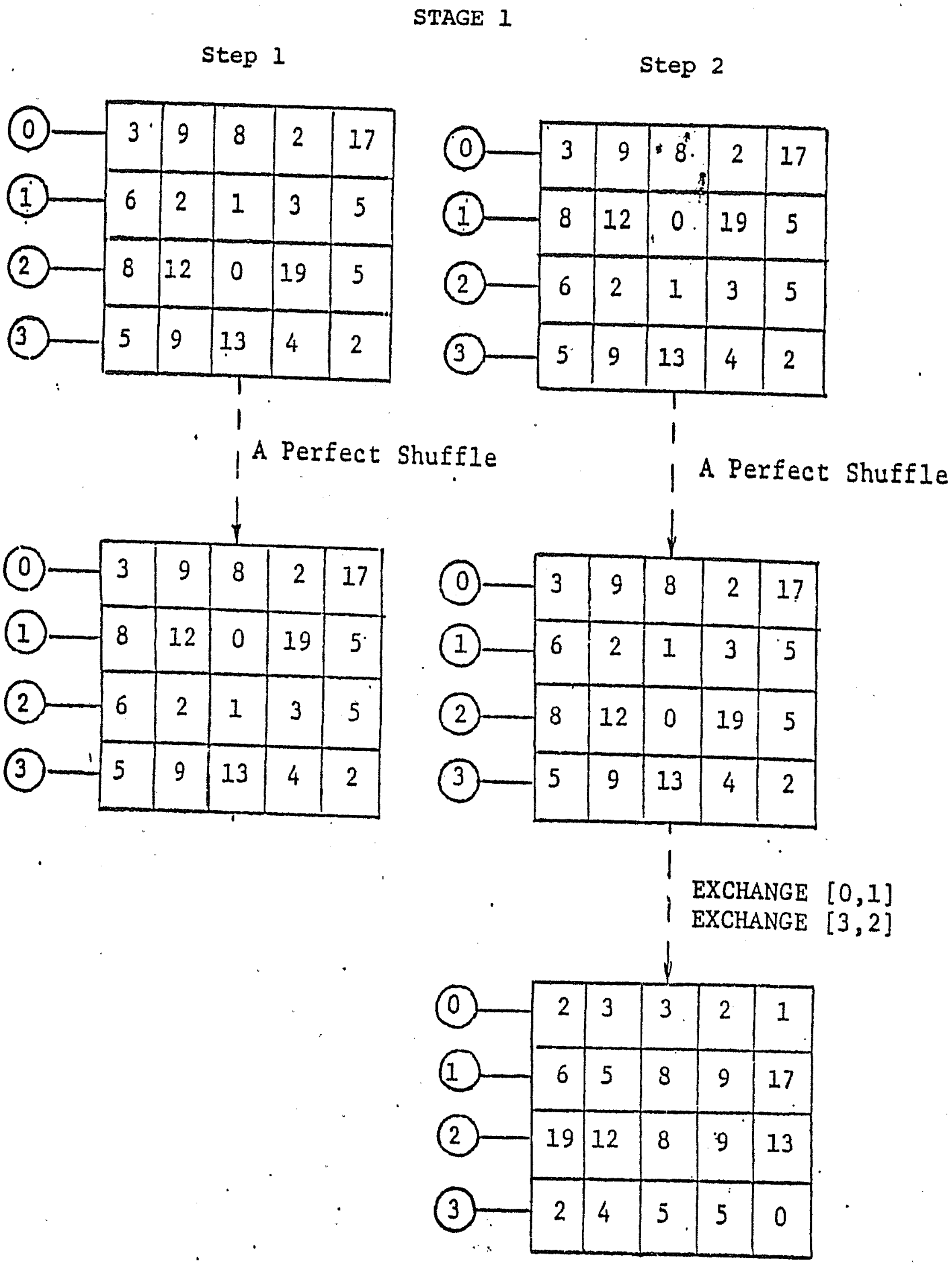
6. EXTERNAL PARALLEL SORTING

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_



Step 1

0	3	9	8	2	17
1	6	2	1	3	5
2	8	12	0	19	5
3	5	9	13	4	2

A Perfect Shuffle

0	3	9	8	2	17
1	8	12	0	19	5
2	6	2	1	3	5
3	5	9	13	4	2

Step 2

0	3	9	8	2	17
1	8	12	0	19	5
2	6	2	1	3	5
3	5	9	13	4	2

A Perfect Shuffle

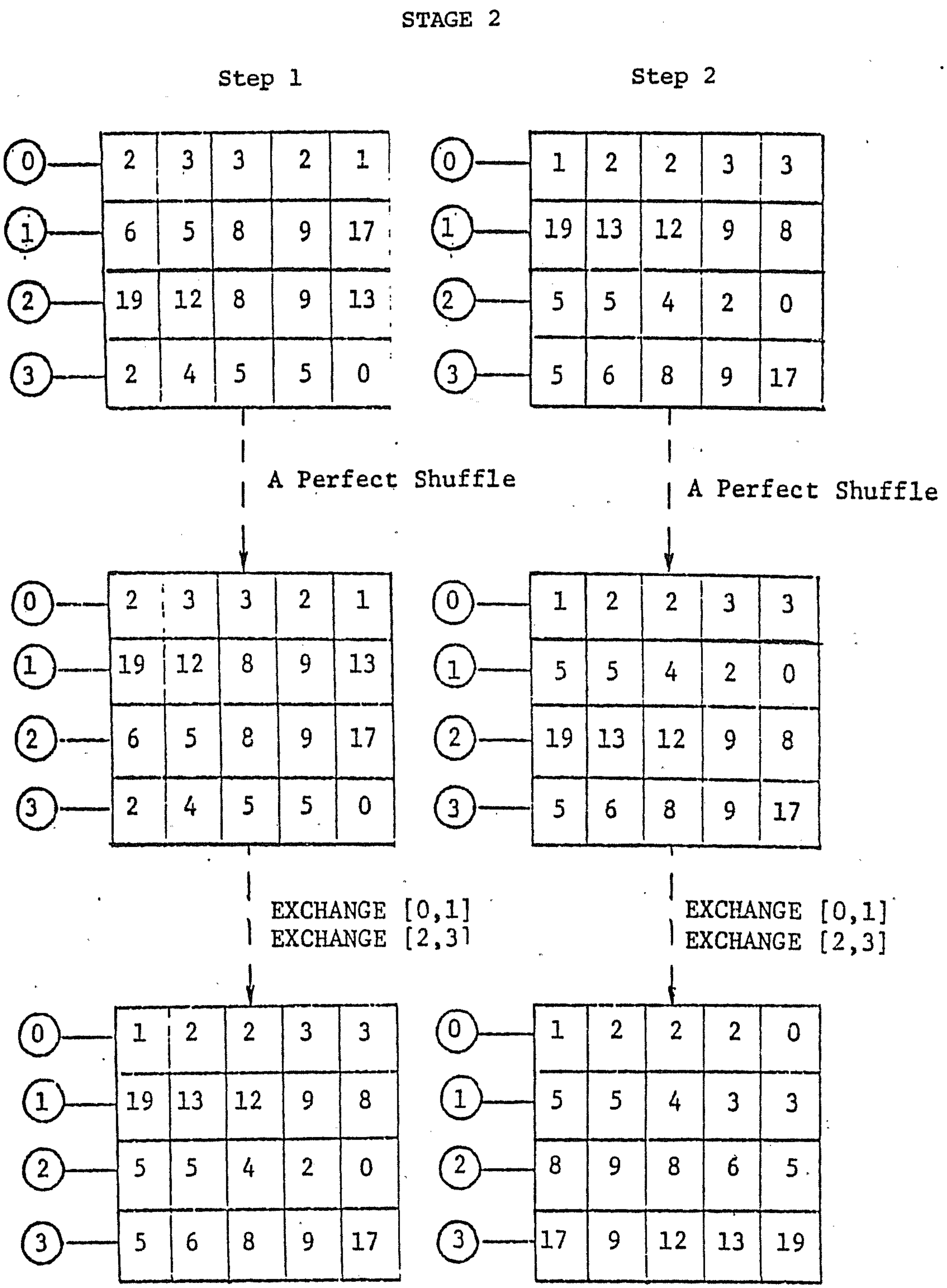
0	3	9	8	2	17
1	6	2	1	3	5
2	8	12	0	19	5
3	5	9	13	4	2

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

0	2	3	3	2	1
1	6	5	8	9	17
2	19	12	8	9	13
3	2	4	5	5	0

Figure 15-a. Block Bitonic Sort, based on bitonic exchange

Figure 15-b. Block Bitonic Sort



STAGE 2

Step 1

0	2	3	3	2	1
1	6	5	8	9	17
2	19	12	8	9	13
3	2	4	5	5	0

A Perfect Shuffle

0	2	3	3	2	1
1	19	12	8	9	13
2	6	5	8	9	17
3	2	4	5	5	0

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

0	1	2	2	3	3
1	19	13	12	9	8
2	5	5	4	2	0
3	5	6	8	9	17

Step 2

0	1	2	2	3	3
1	19	13	12	9	8
2	5	5	4	2	0
3	5	6	8	9	17

A Perfect Shuffle

0	1	2	2	3	3
1	5	5	4	2	0
2	19	13	12	9	8
3	5	6	8	9	17

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

0	1	2	2	2	0
1	5	5	4	3	3
2	8	9	8	6	5
3	17	9	12	13	19

Figure 15-c. Block Bitonic Sort

Final Local Sort

0	1	2	2	2	0
1	5	5	4	3	3
2	8	9	8	6	5
3	17	9	12	13	19

A Parallel localized sort

0	0	1	2	2	2
1	3	3	4	5	5
2	5	6	8	8	9
3	9	12	13	17	19

Final sorted sequence

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 back to the mass storage device in sorted order.

An early result on tape parallel sorting appeared in [Eve74]. Recently in [Frie81], several parallel sorting algorithms have been proposed for files residing on a modified moving-head disk[1].

6.1. PARALLEL TAPE SORTING

The sorting problem addressed in [Even74] is to sort a file of n records with p processors (where p<<n) and 4p magnetic tapes. The only internal memory requirement is that three records could fit simultaneously in each processor´s local

[1] 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.

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 shortly described as follows:

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 serially.

Method 2: the basic idea is that each processor performs a different phase of the serial merge procedure. The ith processor merges pairs of runs of size 2^{i-1} into runs of size 2^{i}. Ideally, n is a power of 2 and log(n) processors are available. A high degree of parallelism is achieved by using the output tapes of a processor as input tapes for the next processor, so that, as soon as a processor has written 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 write 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.

6.2. PARALLEL DISK SORTING

An alternative 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 [Bane78], and, in some cases, already built [Leil78].[2] They appear to constitute a good 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 [Frie81] 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, improving Method 1 of Section 6.1. The improvement is 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

[2] A 600-Mbyte drive with a 4-track parallel readout capability and a data transfer rate of 4.84 Mbytes/second is available for the Cray-1 for approximately $80,000 without controller.

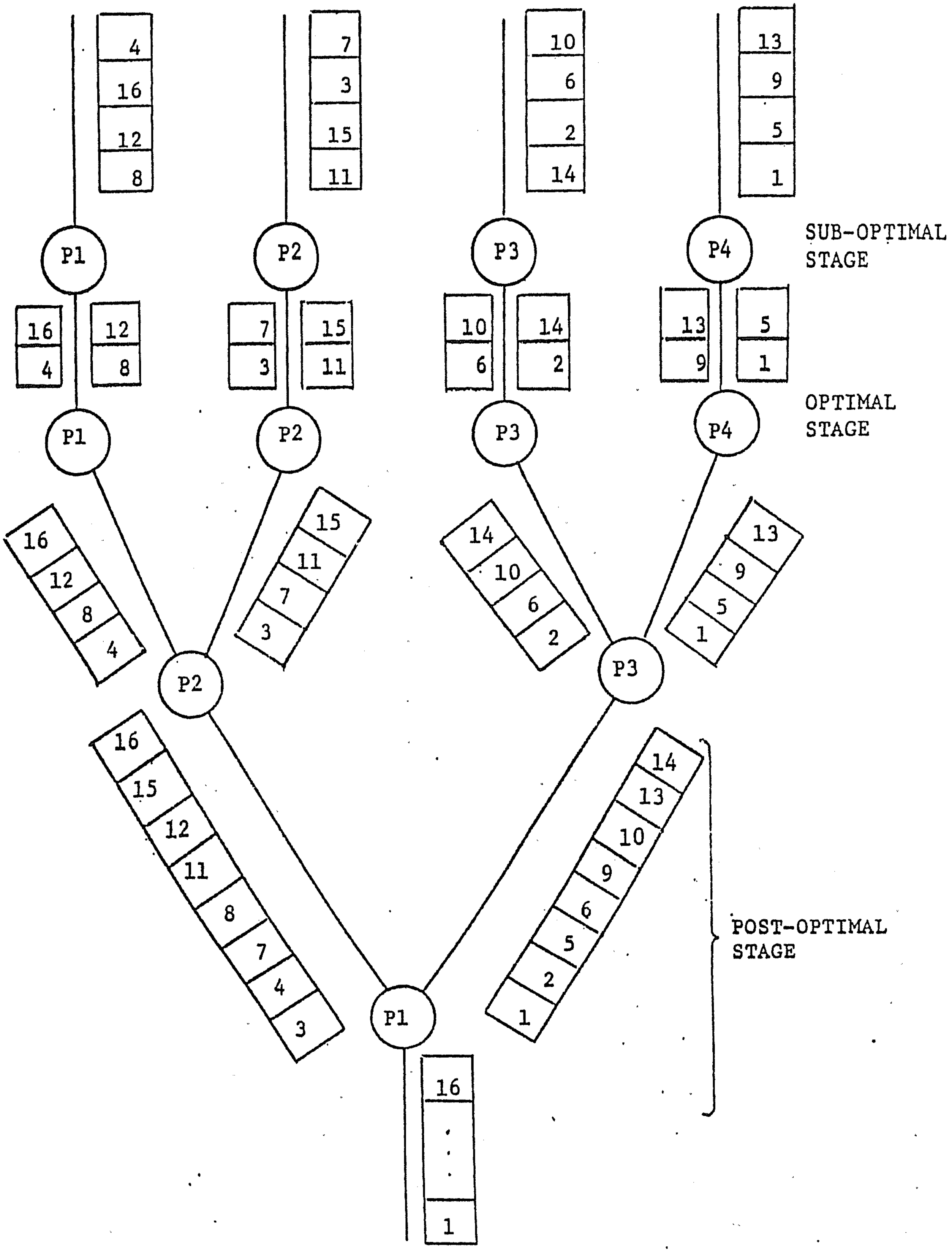


Figure 16. Parallel Binary Merge Sort

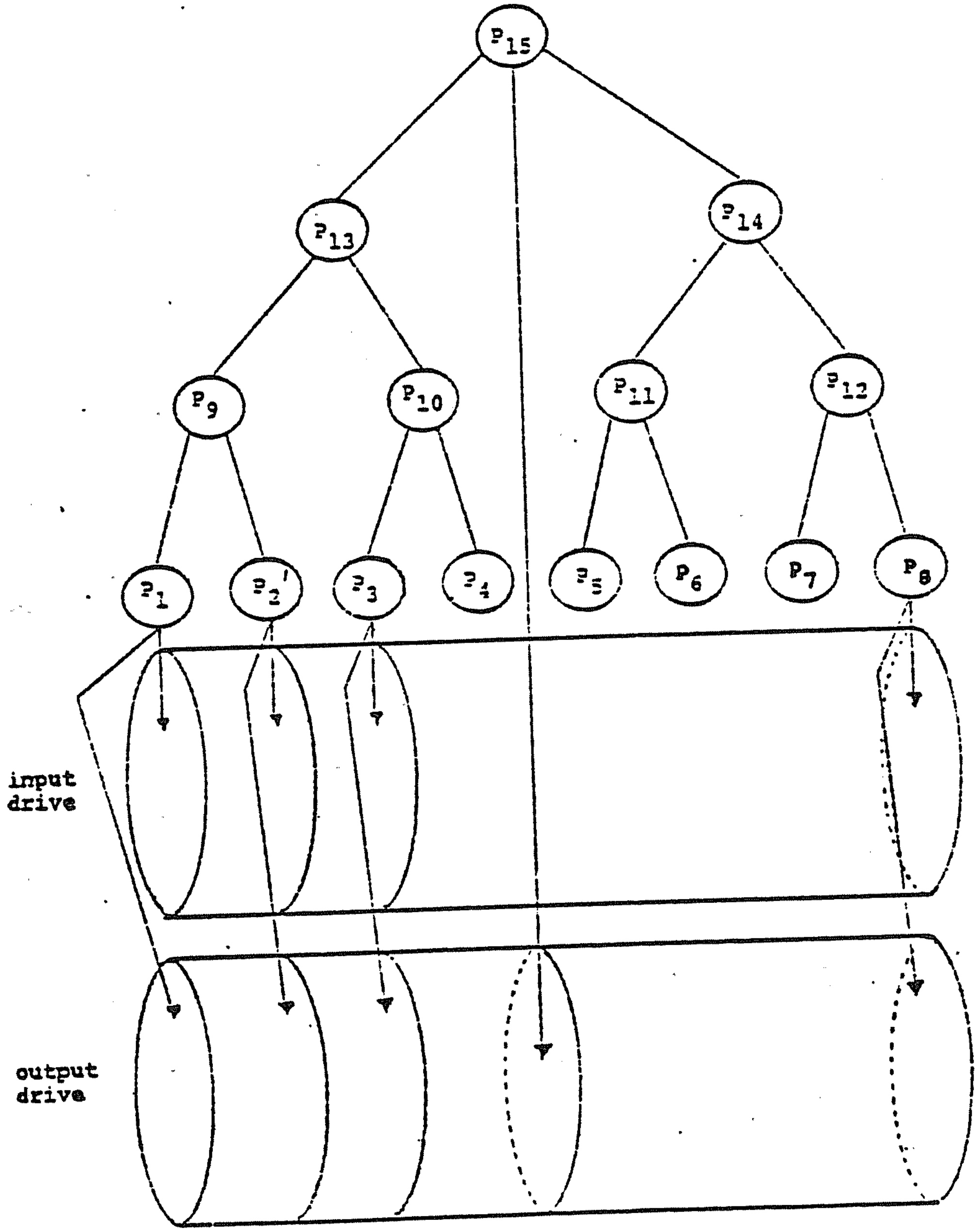


Figure 17. Architecture for the parallel binary merge-sort

phase 1 in Method 1), in which sorting is done by successively merging pairs of longer runs until the number of runs is equal to twice the number of processors. During the suboptimal stage, the processors operate in parallel, but on separate data. Parallel I/O is made possible by associating each processor with a surface of the read disk and a surface of the write disk.

When the number of runs equals 2*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 the (i-1) step 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 the leaf processor to do I/O in parallel, while reducing almost in half the number of processors that must actually do input/output).

6.3. ANALYSIS OF PARALLEL EXTERNAL SORTING ALGORITHMS

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. CONCLUSION

One conclusion emerges clearly from this survey of parallel sorting algorithms: Most research in the area of parallel sorting has concentrated on finding new ways to speedup the algorithm's theoretical computation time, while other aspects (such as technology constraints or data dependency) have received little consideration. Typically, algorithms have been developed that use a very large number of processors (e.g. n processors to sort n elements). Figure 18 summarizes the number of processors and the computation time required by this type of algorithms.

We have shown that most parallel sorting algorithms belong to one of two categories: the network sorting, or the shared memory model algorithms.

The shared memory model algorithms have the best asymptotic complexity. However, it is most unlikely that future technology will supply the tools for implementing any of these algorithms.

On the other hand, the network sorting algorithms can be extended to block sorting algorithms, that can sort (M*p) elements with p processors if each processor has a local memory of size O(M).

Another aspect that has been largely ignored by previous research efforts 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. But for a parallel processor, there is the possibility that several processors can simultaneously read or write. On the Illiac-IV, for example, a fixed-head moving 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

Algorithm	Number of Processors	Execution Time	Other characteristics
Odd-Even Transposition	n	O(n)	
Batcher's Bitonic	$n\log^2 n/2$	$O(\log^2 n)$	sorting network
Stone's Bitonic	n/2	$O(\log^2 n)$	"
Mesh-Bitonic	n^2	O(n)	sorts n^2 records
Muller-Preparata	n^2	O(logn)	
Hirschberg (1)	n	O(logn)	duplicates problem
Hirschberg (2)	$n^{1+1/k}$	O(klogn)	memory conflicts
Preparata (1)	nlogn	O(logn)	
Preparata (2)	$n^{1+1/k}$	O(klogn)	no memory conflicts

Figure 18. Processors required and computation time

point in using a parallel sorting algorithm that requires only O(log n) time, if the startup cost to get the data in memory was 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. Preliminary results in this direction were presented in Section 6.

8. REFERENCES

[Alek69] Alekseyev V.E., Kibernetica, Vol. 5, No. 5, 1969, pp. 99-103.

[Bane78] Banerjee J. and D.K. Hsiao, "Concepts and Capabilities of a Database Computer," ACM Transactions on Database Systems, Vol. 3, No. 4, December 1978.

[Bent79] 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).

[Batc68] Batcher K.E., "Sorting Networks and Their Applications," 1968 Spring Joint Computer Conference, AFIPS Proceedings, Vol. 32.

[Baud78] Baudet G., and Stevenson, D., "Optimal Sorting Algorithms for Parallel Computers,", IEEE-TC, Vol. c-27, No. 1, January 1978.

[Eve74] Even S., "Parallelism in Tape Sorting," CACM, Vol. 17, No. 4, April 1974.

[Frie81] Friedland D.B., "Design, Analysis and Implementation of Parallel External Sorting Algorithms, Ph.D. thesis, December 1981, Univeristy of Wisconsin, Madison.

[Gavr75] Gavril F., "Merging with Parallel Processors," CACM, Vol. 18, No. 10, October 1975.

[Hirs78] Hirschberg, D.S., "Fast Parallel Sorting Algorithms," CACM, Vol. 21, No. 8, August 1978.

[Hsia80] Hsiao D.C. and Menon M.J., "Parallel Record-Sorting Methods for Hardware Realization," Technical Report OSU-CISRC-TR-80-7, The Ohio State University, Columbus, Ohio, July 1980.

[Leil78] Leilich H.O., G. Stiege and H.C. Zeidler, "A Search Processor for Database Management Systems," Proceedings 4th Conference on Very Large Database (1978).

[Knut73] Knuth D.E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley, 1973.

[Mull75] Muller D.E. and Preparata F.P., "Bounds for Complexity of Networks for Sorting and Switching," JACM, April 1975.

[Nass79] Nassimi D. and Sahni S., "Bitonic Sort on a Mesh Connected Parallel Computer," IEEE-TC, Vol. c-27, No. 1, January

1979.

[Prep78] Preparata F.P., "New Parallel Sorting Schemes," IEEE-TC, Vol. c-27, No. 7, July 1978.

[Sieg77] Siegel H.J., "The Universality of Various Types of SIMD Machine Interconnection Networks," Proceedings of the Fourth Annual Symposium on Computer Architecture, March 1977.

[Ston71] Stone H.S., "Parallel Processing with the Perfect Shuffle," IEEE-TC, Vol. c-20, No. 2, February 1971.

[Ston78] Stone H.S., "Sorting on Star," IEEE Transactions on Software Engineering, Vol. 4, No. 2 (March 1978).

[Thom77] Thompson C.D. and Kung H.T., "Sorting on a Mesh Connected Parallel Computer," CACM, Vol. 20, No. 4, April 1977.

[Vali75] Valiant L.G., "Parallelism in Comparison Problems," SIAM Journal of Computing, Vol. 3, No. 4, September 1975.

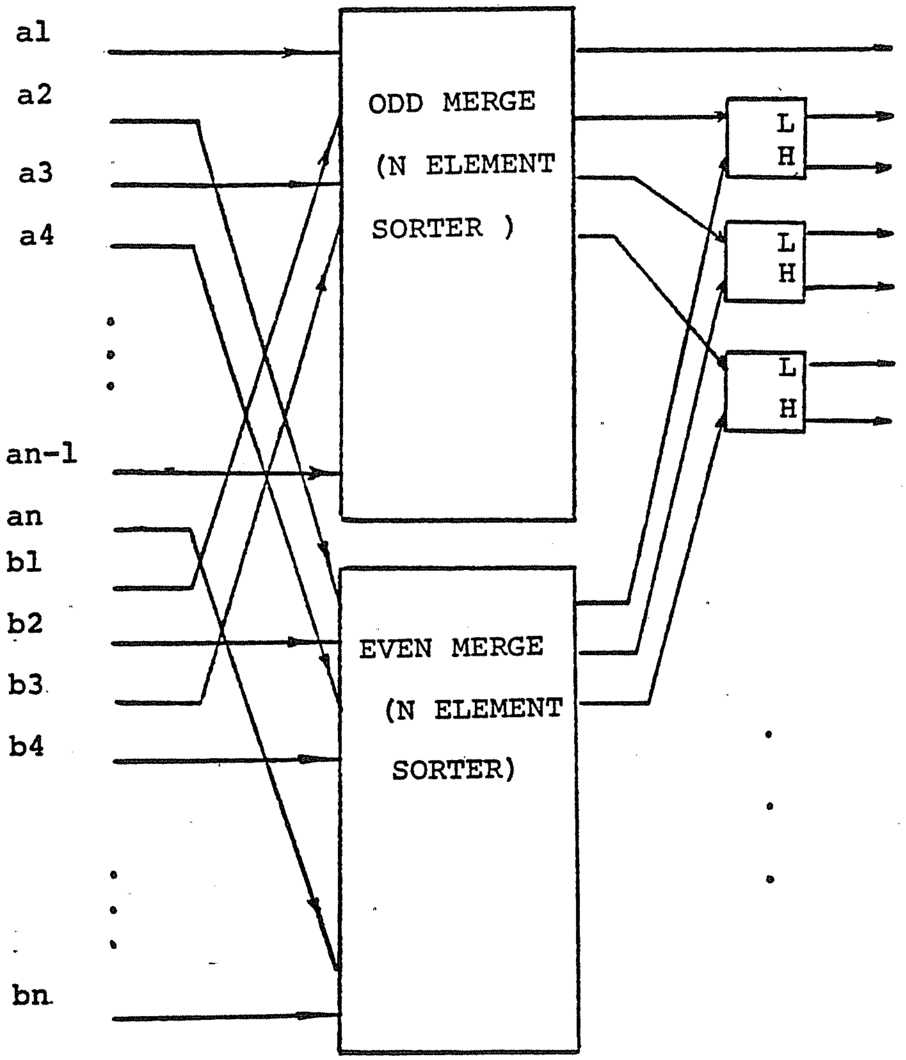


Figure 3. The iterative rule for the odd-even merge.