Reprinted from *IEEE Transactions on Computers*, Volume C-20, Number 2, February 1971, pages 153-161. Copyright © 1971 by The Institute of Electrical and Electronics Engineers, Inc.

Parallel Processing with the Perfect Shuffle

HAROLD S. STONE, MEMBER, IEEE

Abstract—Given a vector of N elements, the perfect shuffle of this vector is a permutation of the elements that are identical to a perfect shuffle of a deck of cards. Elements of the first half of the vector are interlaced with elements of the second half in the perfect shuffle of the vector.

We indicate by a series of examples that the perfect shuffle is an important interconnection pattern for a parallel processor. The examples include the fast-Fourier transform (FFT), polynomial evaluation, sorting, and matrix transposition. For the FFT and sorting, the rate of growth of computational steps for algorithms that use the perfect shuffle is the least known today, and is somewhat better than the best rate that is known for versions of these algorithms that use the interconnection scheme used in the ILLIAC IV.

Index Terms—Data paths, fast Fourier transform, interconnection patterns, matrix transposition, parallel processing, perfect shuffle, polynomial evaluation, sorting.

I. Introduction

One of the most important, yet least understood, aspects of parallel processing is the effect of interconnections of registers and processing units on computational speed. Particular parallel algorithms are well suited for particular interconnection patterns, but these "good" patterns vary widely from algorithm to algorithm. The mesh pattern, for example, was thought to have wide application when it was proposed for the SOLOMON computer [1] ILLIAC IV, the successor to SOLOMON, has a slightly altered form of the mesh pattern, and it is used quite differently than originally conceived in the SOLOMON [2]. The modifications of the mesh pattern can be attributed to greater understanding of parallel algorithms and their data structures.

In this paper we propose an interconnection pattern that is called the *perfect shuffle*. A shuffle of elements of a vector is equivalent to viewing that vector as a card deck, and shuffling them so that after a shuffle the elements from the two halves of the vector alternate. It is not immediately apparent from this description that the perfect shuffle is a useful pattern. The thesis of this paper is that the perfect shuffle is not only useful for particular algorithms but has a wide variety of applications.

This paper is organized as follows. In Section II we describe the perfect shuffle from several points of view in order to build intuition for understanding its application. The next four sections describe particular applications of the perfect shuffle. For each of these applications, the perfect shuffle fits the algorithm better in varying degrees than the cyclically symmetric pattern implemented on the ILLIAC IV. The applications are the fast Fourier transform, polynomial evaluation, sorting, and matrix transposition. The final section discusses the perfect shuffle from a general viewpoint, and summarizes the characteristics of algorithms for which it is well suited.

II. The Perfect Shuffle

The perfect shuffle refers to the interconnection pattern shown in Fig. 1. On the left in the figure is a vector of operands with indices running from 0 to $N-1$, where $N=2^m$ for some integer m. The operands are connected to the vector on the right in the figure through an interlaced interconnection pattern. For reasons we have discussed earlier the interconnection pattern is called a *perfect shuffle*.

Manuscript received December 1, 1969; revised April 10, 1970. This research was supported by the U. S. Atomic Energy Commission under contract AT (04-3) 326 PA 23.

The author is with the Department of Electrical Engineering and Computer Science, Stanford University, Stanford, Calif.

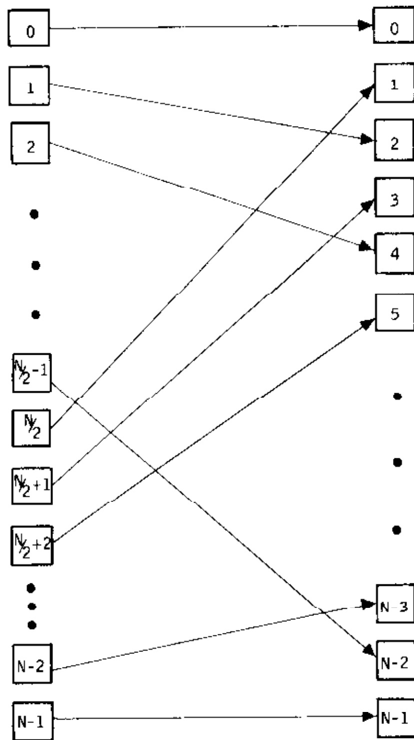


Fig. 1. The perfect shuffle of an N element vector.

Careful study of Fig. 1 shows that the indices on the left are mapped onto the indices on the right according to the permutation P such that

$$\begin{aligned} P(i) &= 2i & 0 \le i \le N/2 - 1 \\ &= 2i + 1 - N & N/2 \le i \le N - 1. \end{aligned} \quad (1)$$

We draw the analogy with the shuffle of a card deck. Divide the vector on the left into two equal pieces, then combine the two pieces by "shuffling" them as it were. That is, alternate the elements of the two pieces as shown in Fig. 1.

Another view of the shuffle is related to the binary representation of the indices of the elements of the vector. We claim that ith element is shuffled to the position i' where i' is the number obtained by cyclically rotating the bits in the binary representation of i one bit position to the left. More specifically, let the binary expansion of i be the following:

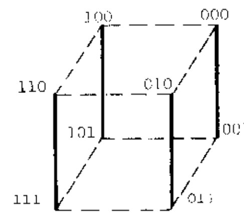
$$i = i_{m-1}2^{m-1} + i_{m-2}2^{m-2} + \cdots + i_1 2 + i_0.$$

Then i' is given by

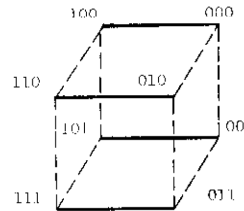
$$i' = i_{m-2}2^{m-1} + i_{m-3}2^{m-2} + \cdots + i_0 2 + i_{m-1}.$$

If $i_{m-1}=0$, then $i'=2i$. If $i_{m-1}=1$, then $i'=2i+1-2^m$. Comparison of the last two statements with (1) shows that they are equivalent.

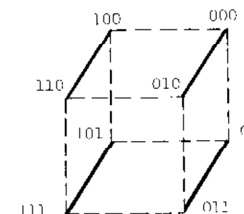
In later sections we will often use the shuffle in a network in which the data undergoes a sequence of shuffles. In these networks the element i is moved to i', then i'', i''', etc., until it eventually returns to its initial position i. Since the shuffle is a permutation of a finite number of objects, after repeating it a sufficient number of times all elements will be returned to place. The cyclic shift rule indicates that after m shuffles and no fewer we return all elements to their starting positions simultaneously.



(a)



(b)



(c)

Fig. 2. Pairings on the 3-cube obtained by perfect shuffles.

In the sequel we present algorithms for which it is necessary to operate on pairs of elements whose indices differ in their binary representation by one bit. This is equivalent to pairing elements on the binary m cube as shown in Fig. 2. The nodes of this cube are labeled with the binary representations of the integers 0, 1, $\cdots$, 7 such that 2 nodes are adjacent on the cube if and only if their labels differ in a single bit position. Each coordinate of the binary representation is associated with one dimension of the cube. Now suppose it is necessary to match pairs of elements in each of the three dimensions as indicated by the bold lines in Figs. 2(a), (b), and (c). We observe that the pairs of Fig. 2(a) are adjacent on the left in Fig. 1, and that after the perfect shuffle, the pairs shown in Fig. 2(b) are adjacent. After one more shuffle, the pairs shown in Fig. 2(c) become adjacent so that all adjacencies on the m-cube are available if a single adjacency and the shuffle interconnection are both available. This behavior is explained rather trivially by the cyclic shift description used above. For example, let elements whose indices differ in bit position 0 be paired in a processor. After shuffling, the pairing index will shift cyclically one position to the *right*, and the paired items will then have indices that differ in bit position $m-1$.

As we see by the examples in the later sections the behavior of the perfect shuffle is rather important for parallel processing.

III. The Fast Fourier Transform

One of the most important advances in computational algorithms in recent years has been the development of the fast Fourier transform [3]. Pease [4] discovered that the

perfect shuffle interconnection pattern is sufficient for executing the transform algorithm on a parallel processor. The discussion in this section is essentially a summary of Pease's work.

Let $A(k), k=0, 1, \cdots, N-1$, be N samples of a time function, sampled at instants that are spaced equally apart. For our discussion, we assume that N is a power of 2, in particular $N=2^m$. The discrete Fourier transform of $A(k)$ is defined to be the discrete function $X(j), j=0, 1, \cdots, N-1$ where

$$X(j) = \sum_{k=0}^{N-1} A(k)W^{jk} \qquad j = 0, 1, \cdots, N-1 \quad (2)$$

and $W=e^{2\pi i/N}$.

The fast Fourier transform algorithm is derived most directly by forming the binary expansions of the indices j and k. We substitute the identities

$$j = j_{m-1}2^{m-1} + \cdots + j_1 \cdot 2 + j_0$$

and

$$k = k_{m-1} \cdot 2^{m-1} + \cdots + k_1 \cdot 2 + k_0$$

into (2) and obtain

$$X(j_{m-1}, \cdots, j_0) = \sum_{k_0} \sum_{k_1} \cdots \sum_{k_{m-1}} A(k_{m-1}, \cdots, k_0)W^{jk} \quad (3)$$

where each of the indices k_i are summed over the binary values 0 and 1.

To compute the Fourier transform of $A(k)$, we shall form m different arrays, $B_1, B_2, \cdots, B_m$, where each B_i is computed from B_{i-1} for $i>1$. The last array of this sequence, B_m, has elements that are the values of $X(j)$, but the elements are scrambled in what is known as *reverse binary order*.

We define the arrays as follows.

$$B_1(j_0, k_{m-2}, \cdots, k_0) = \sum_{k_{m-1}} A(k_{m-1}, \cdots, k_0)W^{j_0 k_{m-1} 2^{m-1}} \quad (4)$$

$$B_s(j_0, \cdots, j_{s-1}, k_{m-s-1}, \cdots, k_0) = \sum_{k_{m-s}} B_{s-1}(j_0, \cdots, j_{s-2}, k_{m-s}, \cdots, k_0) \cdot W^{(j_{s-1}2^{s-1} + \cdots + j_0)k_{m-s}2^{m-s}} \qquad \text{for } s = 2, 3, \cdots, m. \quad (5)$$

We may use the relation $W^{jk_{m-s}} = W^{(j_{s-1}2^{s-1} + \cdots + j_0)k_{m-s}2^{m-s}}$ in the summation of (3) and we find

$$B_m(j_0, j_1, \cdots, j_{m-1}) = \sum_{k_0} \sum_{k_1} \cdots \sum_{k_{m-1}} A(k_{m-1}, \cdots, k_0)W^{jk} = X(j_{m-1}, j_{m-2}, \cdots, j_0).$$

To obtain the value of $X(j)$ when given the vector B_m, one reverses the binary digits in the expansion of j to obtain a new index which we call j'. Then $X(j)=B_m(j')$.

The role of the perfect shuffle becomes evident when we inspect (4). Each element of B_i is the weighted sum of two elements of B_{i-1} (or of A). To determine which two elements of B_{i-1} are combined to form $B_i(j)$, we form the binary expansion of j, and observe the coefficient of 2^{m-i}. Now, we assume that complementing this coefficient yields the representation of the integer $\hat{j}$. Then the two elements of B_{i-1} that contribute to $B_i(j)$ are the elements $B_{i-1}(j)$ and $B_{i-1}(\hat{j})$. Fig. 2 shows the pairwise combinations that form the fast Fourier transform for $N=8$.

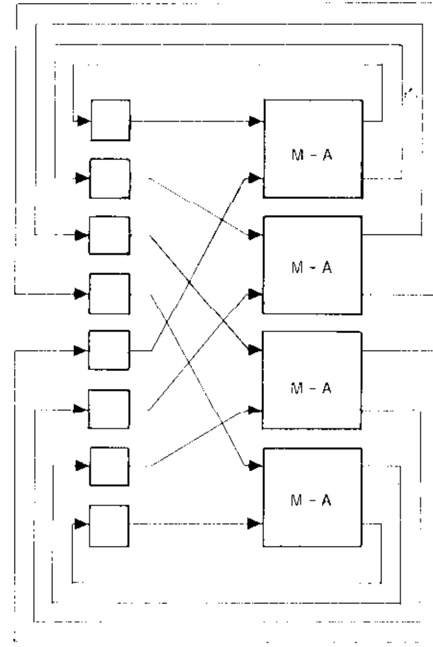


Fig. 3. A processor for the FFT. The "M–A" modules produce weighted sums of their inputs on their outputs.

The importance of the perfect shuffle is clearly shown in the parallel processor shown in Fig. 3. Each of the modules labeled "M–A" in the figure is a "multiply–add" module. A module of this type is capable of computing two weighted sums of its two inputs simultaneously, with the results appearing on the two outputs. The entire processor repeats the following sequence $\log_2 N=m$ times to compute a Fourier transform:

1) shuffle;
2) multiply–add;
3) transfer results back to input of shuffle network.

From the discussion in Section II, it is clear that the network in Fig. 3 first combines pairs of numbers whose indices differ by $2^2=4$ in their binary expansions. After one shuffle, the pairs combined will differ by $2^1=2$, and finally by $2^0=1$. Of course, the resulting transform will have the elements arranged in reverse binary order, and the shuffle by itself cannot bring them back into normal order. Nevertheless, the shuffle in this instance permits the parallel processor to perform the fast Fourier transform algorithm at peak efficiency.

IV. Polynomial Evaluation

A problem that has grown out of the study of parallel processing is the problem of evaluating polynomials in minimum time. The natural lower bound on the minimum time is $\lceil \log_2 N \rceil$ if the polynomial has degree N. The bound arises since it takes at least $\log_2 N$ successive squarings of x to compute x^N when N is a power of 2. In this section we show that there exists an algorithm with a complexity proportional to that of the lower bound when executed on a processor with the shuffle interconnection pattern.

The problem that we wish to solve is the following. Let $a_0, a_1, \cdots, a_{N-2}$ be the coefficients of a polynomial of degree

DATA	MASK	DATA	MASK	DATA	MASK	DATA
a_0	0	a_0	0	a_0	0	a_0
a_1	1	a_1x	0	a_1x	0	a_1x
a_2	0	a_2	1	a_2x^2	0	a_2x^2
a_3	1	a_3x	1	a_3x^3	0	a_3x^3
a_4	0	a_4	0	a_4	1	a_4x^4
a_5	1	a_5x	0	a_5x	1	a_5x^5
a_6	0	a_6	1	a_6x^2	1	a_6x^6
x	1	x^2	1	x^4	1	x^8
	1st iteration Multiply by x		2nd iteration Multiply by x^2		3rd iteration Multiply by x^4	

Fig. 4. The evaluation of a polynomial of degree 6.

$N-2$. We wish to compute $\sum_i a_i x^i$ on a parallel processor that can perform up to N operations simultaneously. Moreover, we seek to find a solution that performs the computation in a time which is proportional to $\log_2 N$.

The solution to the problem is presented schematically in Fig. 4 for the case $N=8$. The coefficients a_i, $i=0, 1, \cdots, N-2$, are arranged in consecutive locations in a block of memory. The processor operates by broadcasting a single number to all locations in the block, and, by assumption, at each location in the block a computation takes place. For this problem we assume that the computation is multiplication. The current contents of a memory register are replaced by the product of the current contents and the broadcast datum. We also assume that there is a masking capability in that the computation only takes place in those words which are identified by a "1" in a corresponding position in a mask register. If a mask bit is "0", the corresponding memory register is unchanged during an operation. Finally, we assume that the numeric value of x is stored in position $N-1$ of the coefficient vector, the last register in the block.

In Fig. 4, the value of x is broadcast, and the setting of the mask register causes every second memory register to update its contents. Now in position 7 we have computed the value of x^2. This value is loaded into the broadcast register as shown in Fig. 4, and the operation is repeated, except that the contents of the mask register have been modified. A third iteration is shown in Fig. 4, this time using the value x^4 as the broadcast datum since x^4 has been computed in the previous step.

At the completion of the third step, we begin a new sequence of operations that forms the sum of the first seven locations in the block. The result of the computation is

$$\sum_{i=0}^{6} a_i x^i.$$

The summing operation is described later in this section.

The shuffle instruction is used in the algorithm above to eliminate the need for storing the masks. To perform the computation illustrated in Fig. 4 with the aid of the shuffle instruction we initialize the mask register to the vector $\boldsymbol{v}=(0, 1, 0, 1, \cdots, 0, 1)$. Then we perform the steps below iteratively.

1) Load the broadcast register from position 7.
2) Broadcast and multiply under mask.
3) Shuffle the mask register.

At the close of this sequence we replace position 7 with the value 0 and form the total of the values in the block. Since the total number of iterations of the steps above grows as $\log_2 N$, and steps for the summing operation also grow as $\log_2 N$, the whole algorithm has a computational complexity which grows as $\log_2 N$.

The summing operation is simply a special case of the fast Fourier transform in which we use the network shown in Fig. 3 with unity weighting factors. After performing $\log_2 N$ iterations of the shuffle/add sequence (multiplication by unity in the multiply–add module need not be performed), each of the N elements of the resultant vector will contain the sum of components of the block.

To show the correctness of our initial sequence of operations, let us assume that each position in the block is associated with an index where the indices run from 0 to $N-1$. Since a_i is stored in the ith memory register, our algorithm is correct if and only if the contents of the ith memory register is multiplied by x^i. Now consider the binary representation of the index i below.

$$i = i_0 + i_1 \cdot 2^1 + \cdots + i_{m-1} \cdot 2^{m-1}. \tag{6}$$

Here we let $m=\log_2 N$. In Fig. 4, the ith register is multiplied by x^{2^j}, $j=0, 1, \cdots, m-1$, if and only if $i_j=1$. That is, the ith register after all cycles are completed is ultimately multiplied by

$$x_j^{\sum i_j \cdot 2^j} = x^i \tag{7}$$

We guarantee that (6) is satisfied by constructing the masks so that register i is multiplied by the broadcast item on cycle j if and only if $i_j=1$. The correctness of the algorithm follows directly from the observation that the masks that are required on successive iterations are obtained by successive perfect shuffles of the initial mask. The initial mask identifies all indices with $i_0=1$ and after each shuffle the identified bit shifts one bit position.

We remark here that the ILLIAC IV type of parallel processor can perform the mass multiplication computations of the first sequence in the same time as a processor that has the shuffle interconnection pattern. The savings attributable to the shuffle interconnection stem from the savings in memory storage, since the shuffle interconnections eliminate the need to store all of the masks. For the second sequence, the summation of the elements of a block, the processor with a shuffle interconnection pattern requires considerably fewer routing instructions than the ILLIAC IV type of computer. The reasoning here is essentially the same as that used in connection with the fast Fourier transform.

V. Sorting

In this section we show how the perfect shuffle can be used in a sorting algorithm that attains the least known processing time for any parallel sorting algorithm within the constraints that we set for the problem.

Efficient sorting has historically been a fascinating problem of computer science. If we assume that the basic operation of a sorting algorithm is the comparison of two numbers followed by a conditional exchange, then it is relatively easy to show that a lower bound for the maximum number of steps in a sort of N numbers is $\log_2 N!$. This follows because there are $N!$ possible initial arrangements of N numbers. The total number of outcomes of the comparisons must be at least as $N!$, and since each comparison has two outcomes, the lower bound follows.

Using Stirling's approximation for $N!$, we see that the lower bound for the maximum number of steps grows as $N \log_2 N$. There have been a number of sorting algorithms that achieve this growth rate for sequential algorithms. (See, for example, Hoare's sort or Floyd's tree sort [5], [6].) For a parallel processor we would expect a somewhat lower growth rate. A reasonable assumption to make is that up to $N/2$ comparisons can occur simultaneously. However, we must place one additional constraint on the problem for parallel processors. This constraint is that no datum can be compared to two or more other items simultaneously. If we were to relax this constraint the conditional exchanges would be rather complex and should not be counted as primitive computational steps.[1]

Close examination of many of the efficient sort algorithms shows that they do not attain the lower bound for computational steps when "parallelized" to execute on a parallel processor with N-fold parallelism. It is rather curious that the usual computational complexity attained is proportional to N rather than proportional to $\log_2 N$. This is true for most sort algorithms because there are initial arrangements of N numbers such that one item will be compared to $N-1$ others before the numbers are sorted. Our assumption that no item can be compared to more than one other item at a time bounds the number of steps to grow at least linearly with N.

In the remainder of this section we describe an efficient sorting algorithm that is the most efficient known for a parallel processor. The number of steps required for this algorithm is proportional to $(\log_2 N)^2$ when $N/2$ comparison–exchange operations can occur simultaneously. The algorithm is Batcher's bitonic sort algorithm [7], [8]. In the context in which it was originally described the algorithm was embedded in a sorting network that consisted of approximately $\frac{1}{2}(\log_2 N)^2$ ranks of comparison–exchange modules. The data to be sorted flows through the network, requiring a unit time to pass through one rank of modules. Consequently, the delay time for Batcher's network is proportional to $(\log_2 N)^2$. In this section we show that a single rank of comparison exchange modules is all that is necessary to perform a bitonic sort. Between successive passes through the comparison–exchange units data is shuffled one or more times. The material of this section is in part implicit in a report by Rohrbacher [8] that covers Batcher's work.

[1] On some processors such as the ILLIAC IV, a single datum can be compared simultaneously to N other items, but no other comparisons can take place at the same time. Sorting algorithms that make use of this capability have a minimum number of steps that is greater than the algorithm described in this section.

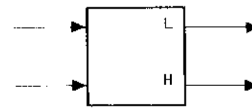


Fig. 5. A comparison–exchange module. The lower value of two inputs is placed on the output labeled "L" and the higher value is placed on the output labeled "H."

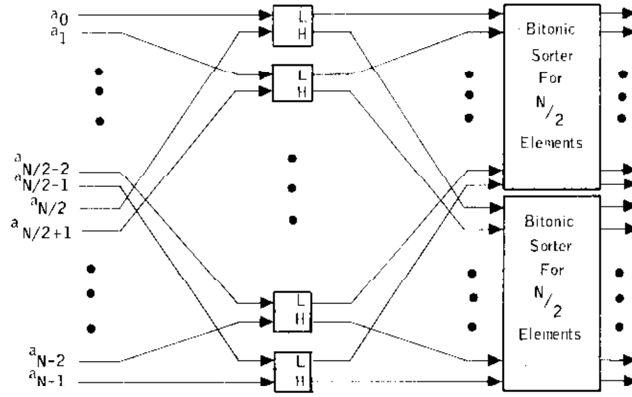


Fig. 6. The structure of a bitonic sorter that can sort sequences of length N.

The basic module for the bitonic sort algorithm is shown in Fig. 5. This module, a comparison–exchange module, can compare the two numbers on its inputs and perform a conditional exchange so that the smaller of the two appears on the output marked "L" and the larger appears on the output marked "H." For our algorithm we find it necessary to associate a mask bit with each module so that the module exchanges its outputs when its associated mask bit is "1" otherwise the outputs are as described in Fig. 5.

To develop the tools for describing and analyzing the sorting algorithm we introduce the following definition.

Definition: A sequence of real numbers $a_0, a_1, a_2, \cdots, a_{N-1}$ is *bitonic* if

1) there is an index i, $0 \le i \le N-1$, such that a_0 through a_i is monotonically increasing and a_i through a_{N-1} is monotonically decreasing; or if
2) the sequence can be shifted cyclically so that condition 1 is satisfied.

Examples of bitonic sequences are the following.

$$1, 3, 5, 7, 8, 6, 4, 2, 0$$
$$7, 8, 6, 4, 2, 0, 1, 3, 5$$
$$1, 2, 3, 4, 5, 6, 7, 8.$$

The central concept of the bitonic sort is that the network shown in Fig. 6 can sort a bitonic sequence. The network performs a comparison–exchange of a_i and $a_{i+N/2}$ for $0 \le i \le N/2-1$. We shall prove shortly that after the comparison–exchange the subsequences consisting of the first $N/2$ numbers and the last $N/2$ numbers are both bitonic. Moverover, every number in the first subsequence is no

greater than any number in the second subsequence. The outputs of the first rank of comparison–exchange modules is passed to a pair of bitonic sorters that operate on half as many numbers. Using a recursive construction technique we can synthesize the smaller sorters by copying the structure of the larger sorters. A complete sorter can be constructed from bitonic sorters by successively bitonic sorting and merging smaller sequences into larger sequences until we have a bitonic sequence of size N that can be input to a network of the type shown in Fig. 6. There remains to prove the assertion underlying Fig. 6.

Theorem [Batcher]: Let $a = a_0, a_1, a_2, \cdots, a_{N-1}$ be bitonic. Let $b_i = \min(a_i, a_{i+N/2})$, $c_i = \max(a_i, a_{i+N/2})$ for $0 \leq i \leq N/2 - 1$. The two sequences $\boldsymbol{b} = b_0, b_1, \cdots, b_{N/2-1}$ and $\boldsymbol{c} = c_0, c_1, \cdots, c_{N/2-1}$ are both bitonic. Moreover, $b_i \leq c_j$ for all i and j.

Proof: First we assume that $a_0, a_1, \cdots, a_{N/2-1}$ is monotonically increasing and the remainder of the sequence is monotonically decreasing. That is, the graph of the first subsequence looks like / and the graph of the second subsequence looks like \. The action of the comparison–exchange modules is to superpose the two graphs into a graph that looks like $\times$, then output the bottom half of this, $\wedge$, as the $\boldsymbol{b}$ sequence and the top half, $\vee$, as the $\boldsymbol{c}$ sequence.

Clearly, both of the output sequences are bitonic, and each of the elements of the $\boldsymbol{b}$ sequence do not exceed any elements of the $\boldsymbol{c}$ sequence. Thus the hypothesis is satisfied for our particular choice of the $\boldsymbol{a}$ sequence.

Our assumption on the $\boldsymbol{a}$ sequence can be violated in two different ways by a bitonic sequence. First, we may have cyclic shift of the $\boldsymbol{a}$ sequence. But the action of cyclically shifting the $\boldsymbol{a}$ sequence only causes the $\boldsymbol{b}$ and $\boldsymbol{c}$ sequences to be cyclically shifted also (with period $N/2$ instead of period N), in which case they remain bitonic. The second violation may be such that the two monotonic subsequences of $\boldsymbol{a}$ are not of equal size. But, by arguing as above, it is easily shown that $\boldsymbol{b}$ and $\boldsymbol{c}$ will still be cyclic shifts of graphs that look like $\wedge$ and $\vee$, respectively, and that no element of $\boldsymbol{b}$ exceeds an element of $\boldsymbol{c}$. Q.E.D.

A complete sorter for eight items that is constructed from bitonic sorters is shown in Fig. 7. The shaded modules in the figure have their mask bits set to "1", that is, they place the larger of the two inputs on the top output. The first rank of modules orders four pairs of numbers in ascending or descending order so that the output of the first rank is a pair of bitonic sequences of length 4. The next two ranks order the two bitonic sequences into ascending and descending sequences of length 4, thus forming a bitonic sequence of length 8. The next three ranks of modules order the bitonic sequence of length 8 into an ascending sequence of length 8.

It is evident from the labels on the modules that the indices of every pair of items that enter a comparison–exchange module differ by a single bit in their binary representations. This suggests that we can construct a sorting processor from a single rank of comparison–exchange modules and a shuffle network as shown in Fig. 8. Each comparison exchange module is connected to a pair of memory registers through a shuffle network. When the contents of the registers are shuffled once, the indices of the paired items differ in their

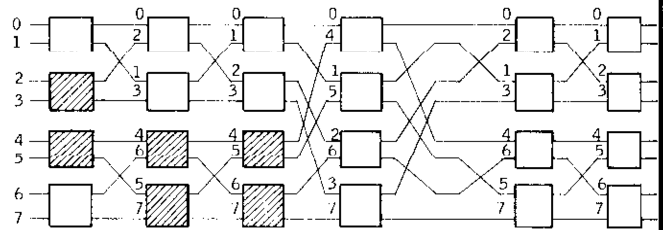


Fig. 7. A sorting network for eight items based upon Batcher's bitonic sorter. All modules are comparison exchange modules, and the shaded modules reverse their outputs.

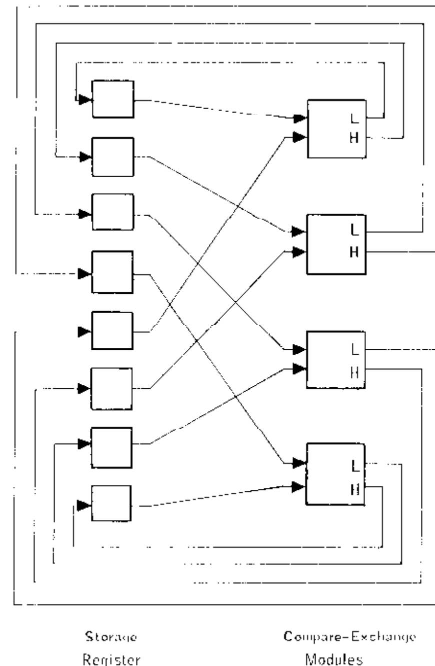


Fig. 8. The structure of a sorting processor for performing the Batcher sort algorithm.

most significant bit. After each shuffle, the indices differ in their next most significant bits. The behavior of the processor under shuffling leads to the following definition.

Definition: At each stage of the Batcher sorting algorithm, the *pivot bit* is defined to be bit position in which the indices of two comparands differ.

The Batcher algorithm calls for comparisons to be done as follows. Let the binary representation of the index i be $i_0 + i_1 \cdot 2 + \cdots + i_{m-1} \cdot 2^{m-1}$ where $2^m = N$. Then the pivot bits for successive stages of a Batcher sorting network are $i_0, i_1, i_0, i_2, i_1, i_0, \cdots, i_{m-1}, i_{m-2}, \cdots, i_1, i_0$. We noted above that if we pivot on i_s, then after one shuffle we can pivot on i_{s-1}. We see that the sequence of pivots naturally divides into m subsequences of increasing lengths, such that in each subsequence the pivot point decreases by 1 at each step. The strategy that we adopt is to shuffle the data as many times as necessary to arrange it properly for the beginning of a sequence of pivots. Then we alternately compare–exchange and shuffle for each of the pivots until we reach bit i_0, the end of the subsequence. Since the kth subsequence has k pivots, for the kth subsequence we have to shuffle data $m - k$ times to obtain the correct initial ordering, then we must perform the k pivots by k iterations of the compare–exchange and shuffle sequence.

There remains to show how to compute the mask that

determines the behavior of the compare–exchange modules. Since the outputs of the modules are connected to memory registers with indices that differ only in bit i_0, we shall number each module by the lower of the indices of the registers to which its output is connected. Inspection of the Batcher sorting network shows that the first rank produces $N/2$ sequences, half ascending and half descending. The mask bit of module i for this rank is given by

$$\text{mask bit } i = i_{m-1} \oplus i_{m-2} \oplus \cdots \oplus i_2 \oplus i_1$$

where $\oplus$ is the EXCLUSIVE-OR operation. In the next two ranks, the network produces $N/4$ sequences, half of which are sorted in ascending order, the other half in descending order. For this part of the network, the mask bit no longer depends on bit i_1 since the action of sorting into pairs permits modules operating on members of sorted pairs to operate identically. Hence, for the next two ranks, the mask bit is given by

$$\text{mask bit } i = i_{m-1} \oplus i_{m-2} \oplus \cdots \oplus i_3 \oplus i_2.$$

Following this reasoning, as progressively longer sorted sequences are produced in the network, the mask bit depends on fewer index bits, with the bits being dropped from the least significant end. We can compute the mask dynamically by computing $i_1, i_2, \cdots, i_{m-1}$ and subtracting it modulo 2 from the mask bit at the appropriate points in a computation. The values of $i_1, i_2, \cdots, i_{m-1}$ for each memory register can be obtained by shuffling the vector $v=(0, 1, 0, 1, \cdots, 0, 1)$ as shown in the previous section. Also subtraction modulo 2 is the same as addition modulo 2.

The complete sorting algorithm in an ALGOL-like language is given as follows.

```
                    Initialization
COMMENT compute the initial value of the mask;
        R := vector (0, 1, 0, 1, · · · , 0, 1);
        mask := R;
        FOR i := 1 STEP 1 UNTIL m DO
            BEGIN
                mask := mask ⊕ R;
                Shuffle (mask)
            END;
COMMENT the array DATA contains the items to be sorted;
        Compare-Exchange (DATA);
        FOR i := 1 STEP 1 UNTIL m − 1 DO
            BEGIN
                COMMENT update mask;
                Shuffle (R);
                mask := mask ⊕ R;
        FOR j := 1 STEP 1 UNTIL m − 1 − i DO
            Shuffle (DATA);
        FOR j := m − i STEP 1 UNTIL m DO
            BEGIN
                Shuffle (DATA);
                Compare Exchange (DATA);
            END;
    END i loop.
```

Fig. 9 shows the state of register R and the mask register at the beginning of the sort and at the beginning of each iteration on index i for the case $N=8$. This can be compared with Fig. 7 to ascertain that the algorithm behaves correctly.

Inspection of Fig. 9 and the ALGOL-like description of the algorithm indicates that the total number of comparison–

	R Register	MASK Register
Initial Values	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1
At beginning of first loop		
i=1	0 1 0 1 0 1 0 1	0 1 0 1 0 1 0 1
i=2	0 1 0 1 0 1 0 1	0 0 0 0 0 0 0 0
i=3	0 1 0 1 0 1 0 1	0 0 1 1 0 0 1 1
At beginning of second loop		
i=1	0 1 0 1 0 1 0 1	0 0 1 1 1 1 0 0
i=2	0 0 1 1 0 0 1 1	0 0 0 0 1 1 1 1
After modification in last iteration	0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0

Fig. 9. The states of the mask register and the R register during the sort algorithm. $N=8$.

exchange steps is $1/2\ m(m+1)$, and the number of shuffles of the data is $m(m-1)$. There are also $2m-1$ shuffles of the mask and R registers. Since $m=\log_2 N$, the computational speed of the algorithm grows as $(\log_2 N)^2$ regardless of whether the dominant contribution is due to the comparison–exchange steps or to the shuffle steps. The Batcher sorting algorithm achieves the smallest known growth rate in computational steps of any parallel sorting algorithm. It is still an outstanding problem to determine if the growth rate is the least possible under the given constraints.

Evidently, the Batcher algorithm depends very significantly on the shuffle interconnection pattern. One question that remains to be answered is to determine if there exist efficient parallel sorting algorithms that require interconnections other than the perfect shuffle. It is worth mentioning that the interconnection pattern of the ILLIAC IV is restrictive for sorting purposes in that the best known sorts for that architecture have computational growths proportional to N.

VI. Matrix Transposition

One requirement of parallel processors that is rarely a requirement for serial processors is the necessity for rearranging data in order to take advantage of the opportunity for parallelism. Two-dimensional matrix calculations are particularly susceptible to these requirements. In many problems it is necessary to have parallel access to both rows and columns of a matrix. Moreover, for matrix multiplication it is necessary to align the rows of one matrix with the columns of another in some architectures to obtain maximum efficiency. One solution to these problems is to build an efficient mechanism for obtaining the matrix transpose. In this section we show how to obtain the transpose of a matrix with the shuffle interconnection pattern.

We assume that the elements of the matrix X, a 2^m by 2^m matrix, are stored in *row major* order in a computer memory. That is, the elements of X are stored in lexicographic order by index with the row index as major key and column index as minor key. By inspection, we find that $X[i, j]$ is displaced from $X[1, 1]$, the base of the array, by an amount given by

$$\text{displacement} = 2^m(i-1) + (j-1).$$

The matrix transpose of X is obtained by interchanging i and j in the formula. To obtain the transpose of X we store

Normal (Row major)	Transposed (Column major)
X[1, 1]	X[1, 1]
X[1, 2]	X[2, 1]
.	.
.	.
.	.
X[1, 2^m]	X[2^m, 1]
X[2, 1]	X[1, 2]
X[2, 2]	X[2, 2]
.	.
.	.
.	.
X[2, 2^m]	X[2^m, 2]
.	.
.	.
.	.
X[2^m, 1]	X[1, 2^m]
X[2^m, 2]	X[2, 2^m]
.	.
.	.
.	.
X[2^m, 2^m]	X[2^m, 2^m]

Fig. 10. The storage arrangement for a matrix in normal ordering and in transposed ordering.

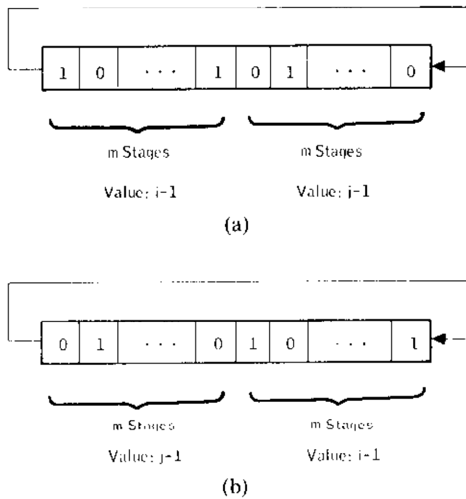


Fig. 11. A shift register analogy that shows that a sequence of m perfect shuffles transposes a $2^m \times 2^m$ matrix.

X in *column major* order. That is, the elements of X are arranged lexicographically by index with the column index as the major key and the row index as the minor key. In this storage arrangement the displacement of $X[i, j]$ relative to $X[1, 1]$ is given by

$$\text{displacement} = 2^m(j-1)+(i-1).$$

The storage arrangements for X and the transpose of X are shown in Fig. 10.

We now show that the matrix transpose of X can be obtained by performing m perfect shuffles of X. To see this, consider the shift register with $2m$ stages that is shown in Fig. 11(a). The high-order m stages hold the binary representation of $i-1$ and the low-order m stages hold the binary representation of $j-1$. The shift register thus holds the representation of $2^m(i-1)+(j-1)$ which is the displacement of $X[i, j]$ relative to $X[1, 1]$. In Section II we note that the action of the perfect shuffle on the indices of a vector is the same as a cyclic shift of the binary representation of the indices. Therefore, after m perfect shuffles, the index whose representation is shown in Fig. 11(a) will be shifted into the position of the index whose representation appears in Fig. 11(b). But this is equivalent to saying the element that is displaced from $X[1, 1]$ by $2^m(i-1)+(j-1)$ before the shuffles is displaced from $X[1, 1]$ by $2^m(j-1)+(i-1)$ after the shuffles. Or equivalently, $X[i, j]$ is moved into the position occupied by $X[j, i]$ and we have the transpose of X.

If X is any square matrix of size less than 2^m by 2^m then it can be transposed in place by storing it in an upper left submatrix of a 2^m by 2^m square matrix. Masking can be used to inhibit operations on data stored outside the submatrix if necessary. Rectangular matrices cannot be transposed in place by this technique, but nevertheless, they can be transposed. Whether or not there is a problem caused by not transposing in place depends on the specific context.

VII. Summary and Conclusions

The principal purpose of this paper is to indicate that the perfect shuffle interconnection pattern has a fundamental role in a parallel processor. In Section II we have shown that the shuffle operates on elements of a vector in the same manner that the indices of the elements are permuted when their binary representations are cyclically permuted. The four examples show how this behavior can be exploited for some particular important applications.

Speaking from a general point of view, the examples share common characteristics. Some of the algorithms match elements whose indices differ by a single bit in their binary representation. This is equivalent to pairing adjacent nodes on a Boolean m-cube. For these algorithms the processor can be constructed to match pairs of adjacent elements along a single dimension of the cube. Then the shuffle can be used to "rotate" other dimensions into the computational position so that every adjacent pair of elements can be treated computationally.

A second aspect that the algorithms have in common is that the processing that is performed on a particular element is functionally related to the bits in the binary representation of its index. In essence, if i is the index of an element then the function $f(i)$ determines the processing to be done to that element. It so happens that $f(i)$ can be a relatively simple function of the bits of the binary representation of i, and, moreover, $f(i)$ can be computed serially by indexing over the bits in the representation of i. For such algorithms, the vector $\boldsymbol{v}=(0, 1, 0, 1, \cdots, 0, 1)$ is the value of the least significant bit of the indices of the vectors. Successive shuffles of $\boldsymbol{v}$ yields the values of successively more significant bits in the binary representations. Hence, $f(i)$ can be computed by simple operations on $\boldsymbol{v}$ and the shuffles of $\boldsymbol{v}$.

We have pointed out that for our examples there is some advantage for the shuffle interconnection scheme over the near-neighbor (or cyclically symmetric) interconnection scheme that is used in the ILLIAC IV. We are not suggesting that the ILLIAC IV scheme be dropped in favor of the perfect shuffle, because, undoubtedly, we can make a strong case for the ILLIAC IV interconnection scheme over the perfect shuffle if we are allowed to choose our examples. Moreover, our analyses compare computational growth rates and do not take into consideration the coefficients of such growth rates. For particular values of N the coefficients may alter

the conclusions we have drawn. For example, it is presumed that a growth rate of $(\log_2 N)^2$ is superior to a growth rate of N. Although this is true for sufficiently large N, it is not necessarily true for $N=64$, which is the size of the ILLIAC IV.

The appropriate point of view is that the perfect shuffle interconnection scheme deserves to be considered for implementation in advanced parallel processors. Whether this interconnection pattern should be used instead of, or in addition to, other interconnection patterns depends very much on the size and intended application of the parallel processor.

References

[1] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The SOLOMON computer," *1962 Fall Joint Computer Conf., AFIPS Proc.*, vol. 22. Washington, D. C.: Spartan, pp. 97–107, 1962.

[2] G. H. Barnes, "The ILLIAC IV computer," *IEEE Trans. Computers*, vol. C-17, pp. 746–757, August 1968. Numerous reports on the ILLIAC IV are available through the University of Illinois Urbana, Ill.

[3] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, pp. 297–301, April 1965.

[4] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *J. ACM*, vol. 15, pp. 252–264, April 1968.

[5] C. A. R. Hoare, "Algorithms 63 and 64," *Comm. ACM*, vol. 4, p. 321, July 1961.

[6] R. W. Floyd, "Algorithm 245," *Comm. ACM*, vol. 7, p. 701, December 1964.

[7] K. E. Batcher, "Sorting networks and their applications," *1968 Spring Joint Computer Conf., AFIPS Proc.*, vol. 32. Washington, D. C.: Thompson, pp. 307–314, 1968.

[8] D. L. Rohrbacher, "Advanced computer organization study," vols. I and II, Goodyear Aerospace Corp., Rept. GER-12314, April 1966. These reports may be obtained from D.D.C. under the accession numbers AD631 870 and AD631 871.