

# An Optimal Systolic Algorithm for Generating Permutations in Lexicographic Order<sup>1</sup>

SELIM G. AKL AND HENK MEIJER

*Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6*

AND

IVAN STOJMENOVIC

*Computer Science Department, University of Ottawa, Ottawa, Ontario, Canada K1N 9B4*

A systolic algorithm is described for generating all permutations of  $n$  elements in lexicographic order. The algorithm is designed to be executed on a linear array of  $n$  processors, each having constant size memory, and each being responsible for producing one element of a given permutation. There is a constant delay per permutation, leading to an  $O(n!)$  time solution. This is an improvement over the best previously known techniques in two respects: the algorithm runs on the (arguably) weakest model of parallel computation, and is cost optimal (assuming the time to output the permutations is counted). The algorithm is extended to run adaptively, i.e., when the number of available processors is other than  $n$ . © 1994 Academic Press, Inc.

## 1. INTRODUCTION

An important branch of computer science is concerned with the study of algorithms for combinatorial problems [17]. In this paper we investigate one such problem, namely the generation of permutations of an ordered set of elements  $\{p_1, p_2, \dots, p_n\}$ , where  $p_1 < p_2 < \dots < p_n$ . The long and distinguished history of this problem, along with sequential algorithms for its solution, are surveyed in [18]. More recently, a number of parallel algorithms for generating permutations were proposed. In order to characterize existing approaches, we begin by listing some desirable properties of parallel permutation generation techniques.

*Property 1.* The permutations are listed in lexicographic order, i.e., if  $A = (a_1, a_2, \dots, a_n)$  and  $B = (b_1, b_2, \dots, b_n)$  are permutations of  $\{p_1, p_2, \dots, p_n\}$ , then  $A$  precedes  $B$  lexicographically if and only if, for some  $j \geq 1$ ,  $a_i = b_i$  when  $i < j$ , and  $a_j < b_j$  [16].

*Property 2.* The algorithm is cost-optimal; i.e., the number of processors it uses multiplied by its running

<sup>1</sup> This research was supported by the Natural Sciences and Engineering Research Council of Canada.

time matches, up to a constant factor, a lower bound on the number of operations required to solve the problem.

This property can be further specified according to the way in which the lower bound is defined. We identify two such definitions:

(a) The time required to "create" the permutations, without actually outputting the  $n$  elements of each permutation, is counted. Optimal sequential algorithms in this sense generate permutations in  $O(n!)$  time, i.e., time linear in the number of permutations of  $n$  elements. Examples of such algorithms are the ones based on interchanging adjacent elements [8, 10, 18, 20].

(b) The time to output each permutation in full is counted. Here, optimal sequential algorithms run in  $O(n * n!)$  time, since it takes  $O(n)$  time to produce a permutation. In this paper we adopt this measure; designing an optimal parallel permutation algorithm under measure (a) remains an open problem.

*Property 3.* The time required by the algorithm between any two consecutive permutations it produces is constant. A constant time delay between outputs is particularly important in applications where the output of one computation serves as input to another.

*Property 4.* The model of parallel computation should be as simple as possible. Arguably, the simplest such model is a linear array of  $m$  processors, indexed 1 through  $m$ , where each processor  $i$  ( $2 \leq i \leq m - 1$ ) is connected by bidirectional links to its immediate left and right neighbors,  $i - 1$  and  $i + 1$ , and processors 1 and  $m$  are each connected to one neighbor. This model is practical, as it is amenable to VLSI implementation [1].

*Property 5.* Each processor needs as little memory as possible, preferably a constant number of words, each of log  $n$  bits and hence capable of storing an integer no larger than  $n$ . This implies that no processor can store an array of size  $n$ , or a counter up to  $n!$ .

*Property 6.* The algorithm should produce all permutations of  $n$  elements for a given  $n$ .

We now review some existing parallel algorithms for generating permutations, and provide a brief assessment of each in light of the above properties. Two such algorithms are described in [1, 2] for generating permutations in lexicographic order. The first runs on a "farm" of processors, i.e.,  $k$  independent processors ( $1 \leq k \leq n!$ ) all connected to a master computer whose job is to start the processors and distribute the tasks among them. The algorithm is cost-optimal in our sense. However, it takes  $O(n)$  memory space per processor since, in fact, each processor produces a subset of the  $n!$  permutations. The algorithm deals with large integers in order to find the initial permutation for each processor. Furthermore, the delay between permutations is nonconstant (since each processor produces a full permutation at a time). The second algorithm runs on the well-known Exclusive-Read Exclusive-Write Parallel Random Access Machine (EREW PRAM), a powerful theoretical model in which processors share a common memory through which they can communicate and exchange data. The algorithm uses  $n$  processors and a memory of size  $O(n)$ . Each processor produces at every step one element of the current permutation. The algorithm requires  $O(\log n)$  time per permutation and, therefore, is not cost-optimal. It is interesting to note, however, that this was the first permutation generation algorithm in which a given permutation is produced by  $n$  processors rather than one only; we follow this paradigm in the present paper.

In [5] the permutations of at most  $m$  out of  $n$  elements are generated on a linear array of processors, also equipped with a so-called selector. The algorithm is cost-optimal. However, the permutations are not produced in lexicographic order, each processor needs memory of size  $O(m)$ , again to produce a full permutation, and the delay between permutations is nonconstant (owing to the fact that nonpermutations are produced during the generation process). The algorithm of [15] runs on a vector computer. It is not cost optimal and deals with large integers. Furthermore, the delay between generating two permutations is nonconstant, each processor requires memory of size  $O(n)$ , and the order of generation is not lexicographic. Recently, algorithms were described in [6, 11] that satisfy Properties 4 and 6 only.

Algorithms enjoying one or more of Properties 1–6, but not all, are described in [4]. Another algorithm for the linear array, satisfying Properties 2–6, is given in [12, 13, 14]. However, the algorithm requires that the elements be decimal numbers, i.e., drawn from the set  $\{1, 2, \dots, n\}$ , since arithmetic is performed on the actual elements.

Finally, as observed in [2], the other known parallel permutation generation methods ([9] and others cited in [1, 2]) are restricted in at least one of the following two

ways: they are capable of generating only a subset of all possible permutations; and/or they are not cost-optimal in any sense.

In this paper, we describe a parallel algorithm for generating all permutations of  $\{p_1, p_2, \dots, p_n\}$  on a linear array of  $n$  processors. The algorithm satisfies Properties 1–6 listed above, and is therefore an improvement over all existing parallel permutation generators. It should be emphasized that the algorithm, like previous algorithms for the linear array, operates in systolic fashion: all processors execute the same algorithm simultaneously, with each processor passing data to neighboring ones in a regular rhythmic pattern [21].

The remainder of the paper is organized as follows. In Section 2, we focus on the property that has eluded previous algorithms for generating permutations on a linear array, namely lexicographic order, and detail our approach for achieving it. The algorithm itself is described in Section 3. It is shown in Section 4 how the algorithm can be made to run adaptively, i.e., when a number of processors other than  $n$  is available. In Section 5, we briefly discuss the more general problem of generating permutations of  $m$  out of  $n$  elements. Some open problems are suggested in Section 6.

## 2. GENERATING PERMUTATIONS IN LEXICOGRAPHIC ORDER

The algorithm that we will describe generates permutations of the set  $\{p_1, p_2, \dots, p_n\}$ . However, for simplicity of exposition we assume that the permutation are generated from  $\{1, 2, \dots, n\}$ . There is no loss of generality due to this assumption since no arithmetic is performed on the actual elements in the algorithm (elements are rather shifted in both directions). Our algorithm to generate the permutations of  $\{1, 2, \dots, n\}$  uses a linear array of  $n$  processors, indexed 1 to  $n$ , and  $n$  variables  $D.1, D.2, \dots, D.n$ , where  $\{D.1, D.2, \dots, D.n\} = \{1, 2, \dots, n\}$ . Each processor  $i$  is responsible for maintaining  $D.i$  by reading only data from processors  $i - 1$  and  $i + 1$ . The processors act in lock-step fashion, and each step produces a new permutation. We begin our description of the algorithm by showing how lexicographic order of generation can be achieved.

Let  $s$  be the smallest natural number for which  $(s - 1)! \geq 2n$  is satisfied. For example, when  $n = 9$ , we have  $s = 5$ . Thus  $s = o(\log n)$ . For a given  $n$ , and the corresponding  $s$ , processors 1, 2, ...,  $s - 1$  will be referred to as the "small numbered" (SN) processors, while processors  $s, s + 1, \dots, n$  will be "large numbered" (LN). Our rationale for this choice of  $s$  will become clear from the subsequent analysis.

As a matter of convenience, we shall index the processors and their associated variables from right to left. This is illustrated in Table I which shows the values of  $D.9$ ,

TABLE I  
A 4-Run of Permutations

	n	5	4	3	2	1	indices of processors
	.....						
1)	4	3	9	8	7	1	2 5 6
2)	4	3	9	8	7	1	2 6 5
3)	4	3	9	8	7	1	5 2 6
4)	4	3	9	8	7	1	5 6 2
5)	4	3	9	8	7	1	6 2 5
6)	4	3	9	8	7	1	6 5 2
7)	4	3	9	8	7	2	1 5 6
8)	4	3	9	8	7	2	1 6 5
9)	4	3	9	8	7	2	5 1 6
10)	4	3	9	8	7	2	5 6 1
11)	4	3	9	8	7	2	6 1 5
12)	4	3	9	8	7	2	6 5 1
13)	4	3	9	8	7	5	1 2 6
14)	4	3	9	8	7	5	1 6 2
15)	4	3	9	8	7	5	2 1 6
16)	4	3	9	8	7	5	2 6 1
17)	4	3	9	8	7	5	6 1 2
18)	4	3	9	8	7	5	6 2 1
19)	4	3	9	8	7	6	1 2 5
20)	4	3	9	8	7	6	1 5 2
21)	4	3	9	8	7	6	2 1 5
22)	4	3	9	8	7	6	2 5 1
23)	4	3	9	8	7	6	5 1 2
24)	4	3	9	8	7	6	5 2 1
25)	4	5	1	2	3	6	7 8 9
26)	4	5	1	2	3	6	7 9 8
	.....						

permutations 7)-12) make a 4-block

permutations 13)-18) make the next 4-block

this finishes a 4-run (24 permutations)

$D.8, \dots, D.1$  for a subset of the permutations of 9 elements. As Table I also shows, LN processors will rarely change the data they produce; on the other hand, SN processors will do it more often. In fact, processor  $i$  will always produce the same element  $(i - 1)!$  times.

In what follows we consider the permutation generation problem for LN and SN processors separately. Two definitions will be useful. A  $k$ -block is defined as the  $(k - 1)!$  permutations obtained when elements in processors  $i$  for  $i \geq k$  are fixed. A  $k$ -run is defined as a  $(k + 1)$ -block. Table I illustrates four 4-blocks that constitute a 4-run.

2.1.  $k$ -Blocks/ $k$ -Runs of Large Numbered (LN) Processors ( $k > s$ )

LN processors will always produce the same element unless they are advised to change their output. Assume that we know how to get permutations for the last  $s$  processors (one  $s$ -run) (two procedures for doing this will be

described later: one is given at the beginning of Section 3 and requires  $o(\log n)$  time between permutations, the second is given in Section 3 and requires constant time between permutations). Thus, we should be able to handle the major changes in the system, namely to set up a new  $s$ -run. Each time a new  $s$ -run begins, it is, in fact, the beginning of an  $i$ -block for each  $i, s \leq i \leq k$ , but not the beginning of a  $(k + 1)$ -block. Here,  $k$  is the smallest index of a processor such that  $k > s$  and  $D.k < D.(k - 1)$ . For example, permutation (25) in Table I is the beginning of a 5-run and also the beginning of a 6-, 7-, and 8-block;  $k = 8$  since it is not the beginning of a 9-block.

Our algorithm should be able to determine processor  $k$ , called a *turning point*, and to update (on time) all the values in processors  $i$  for  $i \leq k$ .

We decide that a search for the turning point starts whenever SN processors recognize the beginning of the last  $s$ -block. Processors  $s - 1, s - 2, \dots, 2, 1$  form a "train". Initially, the train contains elements  $D.(s - 1), D.(s - 2), \dots, D.1$ . The train moves toward processor  $n$ , advancing one step per permutation. When the end of the train (element  $D.1$ ) leaves processor  $s$ , element  $D.s$  is appended to the train and becomes its new end. This continues in the same fashion for processors  $s + 1, s + 2, \dots, k - 1$ , with elements  $D.(s + 1), D.(s + 2), \dots, D.(k - 1)$  appended in this order to the end of the train. When the head of the train (i.e., element  $D.(s - 1)$ ) reaches an element  $D.k$  such that  $D.k < D.(k - 1)$ , it recognizes  $k$  as the turning point. Note that the elements in the train are sorted in increasing order, i.e.,  $D.(s - 1) < D.(s - 2) < \dots < D.1 < D.s < D.(s + 1) < \dots < D.(k - 1)$ .

Once the turning point has been reached, the train is reflected and moves towards processor 1 with elements  $D.(s - 1), D.(s - 2), \dots, D.1, D.s, D.(s + 1), \dots, D.(k - 1)$  visiting the turning point in this order, and changing direction at that point. The first element of the train, say  $D.m$ , that is larger than  $D.k$ , is exchanged with  $D.k$ .

Now, the head of the train is dropped at processor  $k - 1$  and becomes the new value of  $D.(k - 1)$ . As the train moves to the right, towards processor 1, its second, third, ...,  $(k - 1)$ -st elements are dropped in processors  $k - 2, k - 3, \dots, 1$ , and become new values for elements  $D.(k - 2), \dots, D.1$ . For example, consider the permutation 439871256 as shown in Table II. The initial train consists of elements 1256, and the turning point is element 3. As it moves toward the turning point, the contents of the train are 12567, 125678, and 1256789. When 5 reaches processor 8, it is exchanged with 3, and the train returns as 1236789. Elements dropped from the train are shown in boldface.

Since each element in the train has to "travel" at most  $2n$  steps, the correct placement requires  $2n$  steps in the worst case. Since the last  $s$ -block produces  $(s - 1)!$  permutations, all elements will be placed on time for future

TABLE II  
Permutation Train

processors	9	8	7	6	5	4	3	2	1	
elements $D_i$	4	3	9	8	7	1	2	5	6	
left 1	-	-	-	-	1	2	5	6	-	
left 2	-	-	-	1	2	5	6	-	-	
left 3	-	-	1	2	5	6	-	-	-	
left 4	-	1	2	5	6	-	-	-	-	
left 5	-	2	5	6	7	-	-	-	-	append 7 to train
right 5	-	2	1	-	-	-	-	-	-	1 is dropped
left 6	-	5	6	7	-	-	-	-	-	exchange 5 and 3
right 6	-	3	2	-	-	-	-	-	-	3 replaces 5 in train
left 7	-	6	7	8	-	-	-	-	-	append 8 to train
right 7	-	6	3	2	-	-	-	-	-	drop 2
left 8	-	7	8	-	-	-	-	-	-	
right 8	-	7	6	3	-	-	-	-	-	
left 9	-	8	9	-	-	-	-	-	-	append 9 to train
right 9	-	8	7	6	3	-	-	-	-	3 is dropped
left 10	-	9	-	-	-	-	-	-	-	
right 10	-	9	8	7	6	-	-	-	-	
right 11	-	-	9	8	7	6	-	-	-	6 is dropped
right 12	-	-	-	9	8	7	-	-	-	
right 13	-	-	-	-	9	8	7	-	-	7 is dropped
right 14	-	-	-	-	-	9	8	-	-	
right 15	-	-	-	-	-	-	9	8	-	8 is dropped
right 16	-	-	-	-	-	-	-	9	-	
right 17	-	-	-	-	-	-	-	-	9	9 is dropped
new $D_i$	4	5	1	2	3	6	7	8	9	

use because of the relationship between  $s$ , and  $n$ ,  $(s - 1)! \geq 2n$ , as stated earlier.

It is important to note that once processor  $i$  ( $1 \leq i \leq k$ ) receives the new value of  $D_i$ , it continues to produce the old value of  $D_i$  as output until the changes become effective. Deciding exactly when to produce new element  $D_i$  for  $i \leq k$  can be done by using a counter that is broadcast by SN processors. They initialize the counter to  $(s - 1)!$ . As the train moves, the counter decreases by 1 with each step, and each LN processor  $i$  ( $i \leq k$ ) receives the current value of the counter. When the counter goes down to 0, and this happens simultaneously for all processors  $i$ ,  $1 \leq i \leq k$ , the new values  $D_i$  are produced.

A termination condition for the algorithm can be included using the following approach. If a search for the turning point does not find one (i.e., the head of the train reaches processor  $n$  but  $D_n = n$ ), then processor  $n$  sends back a termination message to all processors. They will terminate simultaneously at the appropriate time. This completes our description of the algorithm for SN processors. Summarizing, we obtain the following lemma.

LEMMA. The algorithm described above generates the first permutations of  $k$ -blocks/ $k$ -runs of LN processors in lexicographic order and with  $\leq (s - 1)!$  delay per permutation on a linear array of  $n$  processors; furthermore, each processor has a memory of constant size.

### 3. CONSTANT DELAY LEXICOGRAPHIC GENERATION OF PERMUTATIONS

A simplified version of our main algorithm (to be presented below) that enjoys all the required properties except Property 3 (the delay between two permutations is  $O(s) = o(\log n)$  rather than constant) operates as follows. The algorithm of Section 2.1 is used to produce  $k$ -blocks/ $k$ -runs of large numbered processors ( $k > s$ ). An  $s$ -block and an  $s$ -run are produced by adapting a sequential backtracking algorithm (cf. [1]) for generating permutations of  $n$  elements in lexicographic order.

We are now ready to introduce our systolic algorithm for generating permutations in lexicographic order and constant time between permutations. The algorithm uses the same procedure for generating  $k$ -blocks for LN processors (Section 2.1); thus we only describe a new procedure for producing an  $s$ -block and an  $s$ -run.

#### 3.1. Producing an $s$ -Block and an $s$ -Run with Constant Delay

We now describe how to produce an  $s$ -run assuming the first permutation in the  $s$ -run is known. If  $k < 4$ , a new  $k$ -block will be produced by a straightforward sequential procedure. Thus, in what follows we assume  $k \geq 4$ .

For  $1 \leq k \leq s$  it can be proved that  $k! = o(n \log^2 n)$ . Thus we may use counters for repetitions, since these are not very large numbers ( $k!$  requires  $O(\log n)$  bits to be represented). Processor  $k$  repeats the same element  $(k - 1)!$  times ( $1 \leq k \leq s$ ) and then starts to produce the next element. The latter is determined as follows.

Each processor  $k$  has two counters,  $rc$  (repetition counter) and  $bc$  (block counter), and both are initialized to 1 with the first permutation  $123 \dots n$ . The repetition counter is responsible for repeating the same element  $(k - 1)!$  times and increases by 1 with each permutation, i.e.,  $rc \leftarrow rc + 1 \pmod{(k - 1)!}$  while the block counter counts blocks in a given  $k$ -run, i.e., if  $rc = 1$  then  $bc \leftarrow bc + 1 \pmod{k}$ . In particular, the first and last blocks in a given  $k$ -run are recognized by  $bc = 1$  and  $bc = 0$ , respectively.

Whenever a new  $k$ -block begins (i.e.,  $rc = 1$ ), except when it is the last block in a given  $k$ -run (i.e.,  $bc = 0$ ), processor  $k$  is the leader of a permutation  $D[k \dots 1]$  that should be stored, modified to  $D'[k \dots 1]$  (which is the first permutation of the next  $k$ -block), and replaced by  $D'[k \dots 1]$  at the time the new  $k$ -block begins.

For example, let a 7-block begin with  $D[7 \dots 1] =$

7125689;  $D[7 \dots 1]$  is stored, modified to  $D'[7 \dots 1] = 8125679$ , and replaced by  $D'[7 \dots 1]$  when a new 7-block begins. Simultaneously, 6-, 5-, and 4-blocks begin with  $D[6 \dots 1] = 125689$ ,  $D[5 \dots 1] = 25689$ , and  $D[4 \dots 1] = 5689$ , respectively. They are all stored separately (multiple copies are made of the same element in several blocks), and modified to  $D'[6 \dots 1] = 215689$ ,  $D'[5 \dots 1] = 52689$ , and  $D'[4 \dots 1] = 6589$ , respectively. The replacements are, however, not simultaneous, since the block size is different.

Consider also a 5-block that begins with  $D[5 \dots 1] = 71246$ , which is the last 5-block in a given 5-run. This is part of permutation ... 3871246, which is handled under the 7-block starting with  $D[7 \dots 1] = 3124678$ . Both the 7-block and the 5-block terminate with the same permutation ... 3876421, and the next permutation ... 4123678 is led by processor 7. Therefore the 5-block will not be processed separately since the new 5-block will "arrive" under the "supervision" of the 7-block. This explains the exception made above.

To avoid overloading the SN processors, storing  $D[k \dots 1]$  and retrieving  $D'[k \dots 1]$  is performed using a *shifting* operation (to the left for storing and to the right for retrieving). One shift is always done together with one production step (new permutation).

When a  $k$ -block starts, elements  $D[k \dots 1]$  are stored in another array  $M[k \dots 1]$  (at this moment  $M.(k-1) < \dots < M.1$  is valid).  $M[k \dots 1]$  is shifted to the left and copied into a stack  $Q$ . Processor  $k$  is assigned its own interval in  $Q$  containing  $k$  locations; i.e.,  $k$  processors (from processor  $k(k+1)/2-6$  to processor  $k(k-1)/2-5$ ) serve as storage for processor  $k$ . Whenever a new  $k$ -block begins, new  $i$ -blocks also begin simultaneously for each  $i < k$ . Thus there is a maximal  $t$  such that all processors  $i$  for  $i \leq t$  are leaders of a permutation to be processed.  $M[t \dots 1]$  serves then for  $M[i \dots 1]$  for each  $i \leq t$  at the same time. During the left shift, the array  $M[t \dots 1]$  will create a stack  $Q$  by copying its contents at appropriate times. First  $M[4 \dots 1]$  is copied to  $Q[4 \dots 1]$  immediately. Then  $M[5 \dots 1]$  is copied into  $Q[9 \dots 5]$  after doing 4 shifts to the left. Next,  $M[6 \dots 1]$  will be copied into  $Q[15 \dots 10]$  after 5 more shifts. For a given  $t$  such that  $4 \leq k \leq t$ ,  $M[k \dots 1]$  is copied into  $Q[k(k+1)/2-6 \dots k(k-1)/2-5]$  at the moment when  $M$  "covers" appropriate positions in  $Q$ .

Consider an example. Suppose that the permutation ... 7125689 has started to be produced. The contents of stack  $Q$  and the position of  $M$  are shown in Table III.

Therefore one element  $M.k$  is copied  $t-k+1$  times into stack  $Q$ , at appropriate positions. Processor  $k$  does not know  $t$  when it decides to copy  $D.k$  into  $M.k$  and to shift  $M.k$ . This can be resolved by setting a counter  $C.k$  which (when  $k$  has initiated a left shift for  $M.k$ ) will inform the next  $k-1$  incoming data  $M[k-1 \dots 1]$  that their

TABLE III  
The Contents of Stack  $Q$  and the Position of  $M$

$Q$ initially	5689
$M$ after 4 shifts	7125689
$Q$ after 4 shifts	25689 5689
$M$ after 5 more shifts	7125689
$Q$ after 5 more shifts	125689 25689 5689
$M$ after 6 more shifts	7125689
$Q$ after 6 more shifts	7125689 125689 25689 5689

copies are desired in the part of  $Q$  that is responsible for  $k$ . In this way the value of  $t$  is learned during the left shift.

While waiting in stack  $Q$ , the elements which originated from  $M[t \dots 1]$  are modified to get the new  $t$ -block. For  $t < k$  the block in the given  $t$ -run is obtained by simply interchanging the elements in positions  $t$  and  $t-1$ . In the above example, 5689, 25689, and 125689 are modified to 6589, 52689, and 215689, respectively. Processor  $t$  is looking for element  $M.j$  such that  $M.(j-1) < M.t < M.j$ ;  $M.j$  is the first element of the next  $t$ -block, and the elements  $M.t$  and  $M.j$  are interchanged at the appropriate step during the shift operation. In Table III,  $t = 7 < s$  and a new 7-block started to be produced by permutation 7125689; it is modified to 8125679. Clearly, the modification requires  $t-1$  steps for a  $t$ -block (for instance, shift  $M[t-1 \dots 1] = 125689$  to the left to determine  $M.j$ ; simultaneously shift  $M.t = 7$  to the right to find its new position; other elements do not change their position) and one step for a  $k$ -block when  $k < t$ . Thus stack  $Q$  is modified to 8125679|215689|52689|6589.

Each processor of stack  $Q$  has a counter to decide when to return data to the right. The counter is set to  $(k-1)! - k(k-1) + 12$  when the processor receives the corresponding datum from the left shift. When the count-down reaches zero, the appropriate positions of  $Q$  (that store a new  $k$ -block) are copied into an array  $R[k \dots 1]$  and shifted to the right back to desired positions. This completes our description of the main algorithm. Summarizing, we obtain the following theorem.

**THEOREM.** *The algorithm described above generates all permutations of  $n$  objects in lexicographic order and with constant delay per permutation on a linear array of  $n$  processors, thus achieving an optimal cost of  $O(n * n!)$ ; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as  $n!$ .*

*Proof.* The number of steps between two  $k$ -blocks is  $(k-1)!$ . Shifting to the stack  $Q$  requires  $k(k-1)/2-6$

steps. Note that the positions in  $Q$  for storing and creating a new  $k$ -block are fixed, i.e., between positions  $k(k + 1)/2 - 6$  and  $k(k - 1)/2 - 5$ . While waiting in the stack, the block is modified, and just before the start of the new  $k$ -block the appropriate positions of  $Q$  (that store a new  $k$ -block) are copied into an array  $R[k \dots 1]$  and shifted to the right back to desired positions. The shift to the right requires another  $k(k - 1)/2 - 6$  steps. Thus, the waiting time is  $(k - 1)! - 2(k(k - 1)/2 - 6) = (k - 1)! - k(k - 1) + 12$ . The waiting time must be at least  $k - 1$  steps to allow the modification of the  $k$ -block. Therefore  $(k - 1)! - k(k - 1) + 12 \geq k - 1$ , or  $(k - 1)! \geq k^2 - 13$  which is satisfied for  $k \geq 4$ . Thus there is enough time for shifting.

There is no memory conflict in  $Q$  since for a given  $k$  there is exactly one acting permutation from a  $k$ -block in the stack and it is assigned designated locations in  $Q$ .

There is no overshifting at any processor (i.e., no processor is supposed to shift to the left more than one datum, and analogously for the shift to the right). There is no overlap in shifting to the left since the data from previous shifts have already "moved out" (one needs  $k$  steps to shift completely out of the first  $k$  positions, and the time to produce a  $k$ -block is  $(k - 1)!$  steps, with  $(k - 1)! > k$  for  $k > 3$ ; we use here the property that the element in position  $k$  will not start shifting to the left again before a  $k$ -block is produced). Conflicts in shifting to the right do not occur for the following reason. Before a right shift of a  $k$ -block the  $(k - 1)$ -block was shifted since it was supposed to start first (in fact, the  $(k - 1)$ -block changed  $k - 1$  times while the  $k$ -block was waiting in the stack).

We also note that while a left shift for a  $k$ -block is still taking place, the contents of stack  $Q$  for some  $i < k$  may already be back to produce new  $i$ -blocks, and may be shifting to the left or to the right. The size of stack  $Q$  is  $s(s + 1)/2 - 6 \leq n$  for all  $n \geq 15$ . Thus the stack "fits" into the linear array of processors.

Finally, since the current block is modified in the stack into the block that follows lexicographically, and according to the Lemma  $s$ -runs are also in lexicographic order, the algorithm produces permutations in lexicographic order. The other statements in the Theorem (constant delay, constant memory size, no large integers) are obvious from the description of the algorithm. ■

#### 4. ADAPTIVE ALGORITHM

The algorithm of Section 3 can be made *adaptive*, i.e., to run on a linear array consisting of an arbitrary number  $k$  of processors. There are three cases to consider:

(i)  $k < n$ : here each processor will do the job of  $n/k$  processors in the original algorithm (with  $n/k$  rounded

appropriately if not an integer, so that the last processor does slightly less work);

(ii)  $k > n$ , and  $r = k/n$  is an integer: here the array is divided into  $r$  groups of  $n$  processors each, such that each group produces an interval of consecutive permutations of  $n$  elements;

(iii)  $k > n$ , and  $r = k/n$  is not an integer: this case is handled by combining (i) and (ii).

Since (i) and (iii) are relatively straightforward, we now elaborate on case (ii).

Each of the  $r$  groups will produce  $[n!/r]$  permutations (may be less in the last group). One can use a numbering system to find the initial and final permutation in each group. However, all known numbering systems use large integers (up to  $n!$ ) and are, for practical purposes, inefficient. Recall that we are looking for algorithms that take *constant memory* per processor on a linear array of processors. Here we suggest a new numbering system which does not deal with large integers for a subset of permutations we need to decode.

The group  $t$  ( $1 \leq t \leq r$ ) will produce the permutations numbered from  $(t - 1)[n!/r] + 1$  to  $t[n!/r]$ . We use an example to show how to find the permutation with ordinal number  $(t - 1)[n!/r] + 1$  without the use of large integers. Let  $n = 8$ ,  $r = 11$ ,  $t = 3$ ;  $(t - 1) * [n!/r] = 2 * [40320/11] + 1 = 2 * 3665 + 1 = 7331$ .

$t - 1 = 2$ ,  $(2/11) * 8 = \underline{1} + 5/11$ ; the first element is  $\underline{1} + 1 = 2$ ; the remaining list is 1, 3, 4, 5, 6, 7, 8 ( $7331 - 1 * 5040 = 2291$ ); this step is for verification only, and is not part of the procedure).

$(5/11) * 7 = \underline{3} + 2/11$ ; the second element is  $\underline{3} + 1 = 4$ -th one in the remaining list, which is 5; the remaining list is 1, 3, 4, 6, 7, 8 ( $2291 - 3 * 720 = 131$ ).

$(2/11) * 6 = \underline{1} + 1/11$ ; the 3-rd element is  $\underline{1} + 1 = 2$ -nd in the remaining list, i.e., 3; the remaining list is 1, 4, 6, 7, 8 ( $131 - 1 * 120 = 11$ ).

$(1/11) * 5 = \underline{0} + 5/11$ ; the 4-th element is  $\underline{0} + 1 = 1$ -st in the remaining list, i.e., 1; the remaining list is 4, 6, 7, 8.

$(5/11) * 4 = \underline{1} + 9/11$ ; the 5-th element is  $\underline{1} + 1 = 2$ -nd in the remaining list, i.e., 6; the remaining list is 4, 7, 8 ( $11 - 1 * 6 = 5$ ).

$(9/11) * 3 = \underline{2} + 5/11$ ; the 6th element is  $\underline{2} + 1 = 3$ -rd in the remaining list, i.e., 8; the remaining list is 4, 7 ( $5 - 2 * 2 = 1$ ).

$(5/11) * 2 = \underline{0} + 10/11$ ; the 7th element is  $\underline{0} + 1 = 1$ -st in the remaining list, i.e., 4; now 7 is left, which is the last, 8-th element.

Therefore, the required permutation is 2, 5, 3, 1, 6, 8, 4, 7.

In general, the procedure for finding the first permutation  $D.n, D.(n - 1), \dots, D.1$  in a given group  $t$  can be formally written as follows.

---

```

g ← t - 1
wait ← 0
for i=1 to n do remlist.i ← i endfor
for j=n to 1 do
  index ← (g*j)/r + 1 {integer division}
  g ← (g*j) mod r {remainder}
  D.j ← remlist.index
  if j ≤ s then wait ← wait + (index - 1) * (j - 1)!
  if j = s then for i ← s to 1 do D'.s ← remlist.i endfor
  for i ← index to j - 1 do remlist.i ← remlist.(i + 1) endfor
endfor

```

---

The above procedure is sequential and is supposed to be done by one processor in each of the groups. To avoid the need for more than constant space for this processor, one can decide that the processor which normally should produce the last element in any permutation (for a given group) will find the initial permutation as described. As soon as a new element of the initial permutation is found, all currently known elements are shifted toward the first element. In this way there is no need to store the full permutation by the chosen processor. The other difficulty is with storing the list of the remaining elements. The list can be stored one element per processor, and the desired element can be extracted by a shift operation. This will require  $O(n)$  time per element, or  $O(n^2)$  time to find the initial permutation for a given group.

The first permutation generated by a given group of  $n$  processors may not be the first permutation of an  $s$ -run. For example, the permutation 2, 5, 3, 1, 6, 8, 4, 7 (obtained in the above example) is not the first permutation of a 5-run ( $s = 5$  for  $n = 8$ ). However, to produce an  $s$ -run, the main algorithm of this paper starts from the first permutation of the  $s$ -run and updates the content of stacks involved in the algorithm. To resolve the problem, a given group indeed starts producing permutations from the first permutation of the  $s$ -run but does not output them until it arrives at the real beginning. In our example, the 5-run starts from the permutation 2, 5, 3, 1, 4, 6, 7, 8 and it is only 10 permutations later that the permutation 2, 5, 3, 1, 6, 8, 4, 7 is output ( $10 = 0 * 4! + 1 * 3! + 2 * 2! + 0 * 1!$ ). The waiting period is already included in the above algorithm through the calculation of the value *wait*, which is propagated to all processors. The first permutation in the corresponding  $s$ -run is memorized when  $D.s$  is being determined, and is denoted by  $D.n, D.(n - 1), \dots, D.(s + 1), D'.s, D'.(s - 1), \dots, D'.1$  in the above algorithm. The waiting period is at most  $O(s!) = O(n \log n)$ .

Since the computation described is merely a preprocessing step done once at the beginning, the time it takes to start up can be neglected.

## 5. CONCLUSION

We succeeded in deriving a cost-optimal algorithm for generating permutations of  $n$  objects on a linear array of processors. The algorithm uses  $n$  processors and produces permutations in constant time per permutation. This improves the previously known algorithms if the time to output the permutations is counted.

One can design an algorithm for generating permutations of  $m$  out of  $n$  elements as follows: Generate all combinations of  $m$  out of  $n$  elements [3], and for each generated combination, generate the corresponding permutations of  $m$  out of  $m$  elements (section 3). The algorithm satisfies Properties 2–6. It is an interesting open problem to describe an analogous cost-optimal algorithm for enumerating permutations of  $m$  out of  $n$  objects, in lexicographic order. Designing cost optimal algorithms for generating permutations with repetitions and permutations with some restrictions (cyclic, alternate, rosary, reflection-free) are also open problems.

## ACKNOWLEDGMENTS

We wish to thank John Calvert for his careful reading of the manuscript and his implementation of the main algorithm. We are also indebted to the referees of this journal for their many useful suggestions that led to an improved presentation of this paper.

## REFERENCES

1. Akl, S. G. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1989.
2. Akl, S. G. Adaptive and optimal parallel algorithms for enumerating permutations and combinations. *Comput. J.* **30**, 5 (1987) 433–436.
3. Akl, S. G., Gries, D., and Stojmenović, I. An optimal parallel algorithm for generating combinations. *Inform. Process. Lett.* **33**, (1989/90), 135–139.
4. Akl, S. G., Meijer, H., and Stojmenović, I. Optimal parallel algorithms for generating permutations. Tech. Rep. No. 90-270, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, Jan. 1990.

5. Chen, G. H., and Chern, M.-S. Parallel generation of permutations and combinations. *BIT* **26**, (1986), 277–283.
6. Cosnard, M., and Ferreira, A. G. Generating permutations on a VLSI suitable linear network. *Comput. J.* **32**, 6 (1989), 571–573.
7. Er, M. C. A parallel algorithm for cost-optimal generation of permutations of  $r$  out of  $n$  items. *J. Inform. Optim. Sci.* **9**, 1 (1988), 53–56.
8. Even, S. *Algorithmic Combinatorics*. Macmillan, New York, 1973.
9. Gupta, P., and Bhattacharjee, G. P. Parallel generation of permutations. *Comput. J.* **26**, 2 (1983), 97–105.
10. Johnson, S. M. Generation of permutations by adjacent transposition. *Math. Comput.* (1963), 282–285.
11. Kokosinski, Z. On generation of permutations through decomposition of symmetric groups into cosets. *BIT* **30**, 4 (1990), 583–591.
12. Lin, C. J. Parallel algorithm for generating permutations on linear array. *Inform. Process. Lett.* **33**, (1990), 167–170.
13. Lin, C. J. Parallel generation of permutations on systolic arrays. *Parallel Comput.* **15**, 1 (1990), 267–276.
14. Lin, C. J. Parallel permutation generation on linear array. *Intern. J. Comput. Math.* **38**, (1991), 113–121.
15. Mor, M., and Fraenkel, A. S. Permutation generation on vector processors. *Comput. J.* **25**, 4 (1982), 423–428.
16. Ord-Smith, R. J. Generation of permutations in lexicographic order. *Commun. ACM* **11**, 2 (1968), 117.
17. Reingold, E. M., Nievergelt, J., and Deo, N. *Combinatorial Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1977.
18. Sedgewick, R. Permutation generation methods, *Comput. Surveys* **9**, 2 (June 1977), 137–164.
19. Stojmenović, I. An optimal algorithm for generating equivalence relations on a linear array of processors. *BIT* **30**, 3 (1990), 424–436.
20. Trotter, H. F. Algorithm 115. *Commun. ACM* **5**, 8 (Aug. 1962), 434–435.
21. Ullman, J. D. *Computational Aspects of VLSI*. Computer Science Press, Rockville, MD, 1984.

SELIM G. AKL received a Ph.D. degree in computer science from McGill University, Montreal, Quebec, Canada, in 1978. He is currently a professor of computing and information science at Queen's University, Kingston, Ontario, Canada. His research interests are primarily in the area of algorithm design and analysis, in particular for problems in parallel computation. He is author of *Parallel Sorting Algorithms* (Academic Press, 1985), *The Design and Analysis of Parallel Algorithms* (Prentice-Hall, 1989), and co-author of *Parallel Computational Geometry* (Prentice-Hall, 1992). He is an editor of *Information Processing Letters* (North-Holland), the *Journal of Cryptology* (Springer-Verlag), *Computational Geometry* (Elsevier), *Parallel Processing Letters* (World Scientific Publishing), and the *Journal of Parallel Algorithms and Applications* (Gordon and Breach), and a member of the IEEE Computer Society, the Association of Computing Machinery, and the European Association for Theoretical Computer Science. In 1990 Dr. Akl occupied the Louis Néel Chair at the Laboratoire de l'Informatique du Parallélisme of the Ecole Normale Supérieure de Lyon, France.

HENK MEIJER has obtained his Ph.D. degree in 1983 from the Department of Mathematics and Statistics at Queen's University in Kingston, Ontario, Canada. He is currently an associate professor in the Computing and Information Science Department at the same university. His research interests include cryptology, data security, and parallel algorithms.

IVAN STOJMENOVIĆ received the B.S. and M.S. degrees in 1979 and 1983 from the University of Novi Sad and Ph.D. degree in mathematics in 1985 from the University of Zagreb. In 1980 he joined the Institute of Mathematics, University of Novi Sad. During the winter 1985/86 he was a visiting researcher at the Electrochemical Laboratory, Tsukuba, Japan. In the fall 1987 and spring 1988 he was a visiting assistant professor at the Computer Science Department, Washington State University (Pulman, WA) and Department of Mathematics and Computer Science, University of Miami (Miami, FL), respectively. In fall 1988, he joined the faculty in the Computer Science Department at the University of Ottawa (Ottawa, Canada) where currently he is an associate professor. His research interests are parallel computing, computational geometry, combinatorial algorithms, and multiple-valued logic. He is an editor of *Parallel Processing Letters* and the *Journal of Parallel Algorithms and Applications*.

---

Received February 16, 1990; revised December 19, 1990; accepted April 16, 1992