Permutation Generation on Vector Processors

M. Mor and A. S. Fraenkel*

Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel 76100

An efficient algorithm for generating a sequence of all the permutations $P(i)$ on N symbols using parallel processors, all of which perform identical operations, is presented ($0 \le i < N!$). Starting from a naïve version in which all permutations have to be kept in memory, a final version in which each processor has to store only a single permutation is evolved. The algorithm is most efficient if the number of processors is $s!$ for some $s \le N$ but can be used, at the price of some bookkeeping, for any number of processors. Some properties of the sequence $\{P(i)\}$ are pointed out, such as the fact that the $k!$ permutations on $1, \ldots, k$ are generated prior to moving element $k+1$ ($1 \le k < N$). The correspondence $i \leftrightarrow P(i)$ is given explicitly.

1. INTRODUCTION

Generating all $N!$ permutations of N elements is usually done by generating the $(i+1)$th permutation from the ith permutation, for $i = 1, \ldots, N! - 1$ (see e.g. Refs. 1 and 2). Such algorithms are therefore very inefficient for SIMD (single-instruction stream multiple-data stream) machines,[3] systolic machines,[4] and vector processors,[5,6] where the same operation is performed synchronously in parallel on different data. This paper describes a simple efficient algorithm for generating all $N!$ permutations of N elements on such machines.

The basic idea of the *p*arallel *p*ermutation-generation on vector processors (in short: PEP) algorithm is to arrange the $N!$ permutations of N elements into a specific order such that large blocks of permutations can be generated simultaneously by performing the same transformation on previous blocks of permutations. The naïve PEP scheme that will be described in Section 2 enables one to generate iteratively $kk!$ permutations in the $(k+1)$th iteration using $k!$ SIMD-processors (or, equivalently, using a vector processor for vectors of length $k!$). If only $t < k!$ SIMD processors are available, the scheme can easily be modified to enable generation of t permutations at almost every step, by dividing every iteration into $\lfloor k!/t \rfloor$ subiterations. The naïve PEP scheme assumes that all permutations can be kept in memory. In Section 3 this assumption will be removed, and a PEP algorithm for t processors when $t = s!$ for any s will be described where every processor needs to store only one permutation. This PEP algorithm is further expanded in the final Section 4, to enable its application for any number of processors, but in this case some extra bookkeeping is needed.

2. THE NAIVE PEP SCHEME

We may assume that all permutations will be stored in a matrix A of $N!$ rows and N columns, where each row $A(j)$ of the matrix contains one permutation $p(j, 1), p(j, 2), \ldots, p(j, N)$ of N elements. ($0 \le j \le N! - 1$).

Let us assume that the kth subset ($k < N$) of permutations, consisting of all $k!$ permutations of the first k elements, has been generated after k *iterations*, and stored in the first $k!$ rows of A, numbered 0 to $k! - 1$:

$$\begin{array}{l} p(0,1), p(0,2), \ldots, p(0,k), k+1, k+2, \ldots, N \\ p(1,1), p(1,2), \ldots, p(1,k), k+1, k+2, \ldots, N \\ \vdots \\ p(k!-1,1), p(k!-1,2), \ldots, p(k!-1,k), k+1, \\ \quad k+2, \ldots, N. \end{array}$$

The next step in the algorithm is to interchange the $(k+1)$th column with the kth column in the above displayed $k!$ rows $A(0), \ldots, A(k!-1)$, using $k!$ processors simultaneously, and store them in A as the next $k!$ rows with serial numbers $A(k!)$ to $A(2k!-1)$. The next step is to interchange the kth column with the $(k-1)$th column in $A(k!)$ to $A(2k!-1)$, etc. until finally the second and first column are interchanged in $A(kk!)$ to $A((k+1)k!-1)$. At this stage the $(k+1)$th iteration terminates, the $(k+1)$th subset of permutations has been generated and stored in $A(0)$ to $A((k+1)!-1)$ and the process is continued with $(k+2)$. Table 1 shows the permutations of four elements as generated by this algorithm.

This scheme evidently has the following properties: (a) each permutation is generated by a single interchange of two (adjacent) elements; (b) the kth subset of permutations is generated before the $(k+1)$th element is moved. (This property, shared by some serial algorithms, is discussed, e.g. in Refs. 2, 7 and 8); (c) the reflection of the jth permutations is the $(N! - j - 1)$th permutation, $0 \le j < N!$. ($A(i)$ is a *reflection* of $A(j)$ if $p(i, N-l) = p(j, l+1)$, $0 \le l < N$.) Consequently, the first and the second half of the sequence of permutations generated do not contain the reflection of any permutation of that half. A sequence of permutations, with the latter property is called by Roy[2] a 'reflection free' sequence and it is claimed to be an advantage in many applications.[9]

Properties (a) and (b) are obvious. As to property (c), we prove:

Lemma

The jth permutation generated by the naïve PEP scheme is the reflection of the $(N! - j - 1)$th permutation ($0 \le j < N!$).

*Partially supported by a grant of Bank Leumi Le-Israel.

Table 1. Permutations of four elements generated by the naïve PEP scheme

Row No.	Permutation			
0	1	2	3	4
1	2	1	3	4
2	1	3	2	4
3	2	3	1	4
4	3	1	2	4
5	3	2	1	4
6	1	2	4	3
7	2	1	4	3
8	1	3	4	2
9	2	3	4	1
10	3	1	4	2
11	3	2	4	1
12	1	4	2	3
13	2	4	1	3
14	1	4	3	2
15	2	4	3	1
16	3	4	1	2
17	3	4	2	1
18	4	1	2	3
19	4	2	1	3
20	4	1	3	2
21	4	2	3	1
22	4	3	1	2
23	4	3	2	1

Proof. Without loss of generality, the elements permuted are the first N natural numbers here and below. For $N = 2$ the assertion is true, since the permutations generated by the PEP scheme are 12 and 21 in this order. Let us assume that the assertion is true for $N = k$, that is,

$$p(k! - j - 1, k - l) = p(j, l + 1) \qquad (0 \le j < k!, 0 \le l < k).$$

Now let $N = k + 1$. We have to show that

$$p((k+1)! - j - 1, k + 1 - l) = p(j, l + 1) \qquad (0 \le j < (k+1)!, 0 \le l < k + 1).$$

For $0 \le j < (k+1)!$, let $q = \lfloor j/k! \rfloor$. Then $0 \le q \le k$. In permutation j, the element $(k + 1)$ is at position $k + 1 - \lfloor j/k! \rfloor = k + 1 - q$, and in permutation $(k+1)! - j - 1$ it is at position $k + 1 - \lfloor((k+1)! - j - 1)/k!\rfloor = q + 1$. Thus the element $k + 1$ is indeed reflected properly.

We can think of the generation of the $(k + 1)!$ permutations as consisting of two stages:

(i) Write down $k + 1$ blocks of $k!$ rows each, consisting of the $k!$ permutations of $1, \ldots, k$ generated by the algorithm.

(ii) In the qth block, insert the element $(k + 1)$ at position $k + 1 - q$ $(0 \le q \le k)$.

For stage (i), the induction hypothesis implies, since all blocks are congruent,

$$p(tk! - j - 1, k - l) = p(j, l + 1) \qquad (0 \le j < tk!, 0 \le l < k, 0 \le t \le k + 1),$$

and, in particular,

$$p((k+1)! - j - 1, k - l) = p(j, l + 1) \qquad (0 \le j < (k+1)!, 0 \le l < k).$$

When stage (ii) is performed, element $(k + 1)$ is inserted symmetrically: at position $k + 1 - q$ in permutation j and at position $q + 1$ in permutation $(k+1)! - j - 1$. Hence the elements $1, \ldots, k$ preserve their symmetry also after stage (ii) was performed, and

$$p((k+1)! - j - 1, k + 1 - l) = p(j, l + 1) \qquad (0 \le j < (k+1)!, 0 \le l < k + 1).$$

Therefore permutation $(k+1)! - j - 1$ is a reflection of permutation j.

Remark. By using the reflection property, the parallelism in the naïve PEP scheme can be about doubled as follows: (a) Two copies of the permutations 0 to $(N!/2) - 1$ are generated in parallel, that is, the naïve PEP algorithm is terminated after $N!/2$ permutations have been generated. (b) In one of the two copies, element $(j + 1)$ is exchanged with $N - j$ $(0 \le j < N)$.

In the above described naïve PEP scheme the jth permutation can easily be obtained from the number j, $0 \le j < N!$, as follows. (The transformation $j \to A(j)$ is called 'orderly listing' or 'ranking'.[10])

Algorithm A

Let the $N!$ permutations be numbered 0 to $N! - 1$. In order to find the jth permutation $0 \le j \le N! - 1$:

(1) Represent j by its factorial representation (Ref. 10, p. 7)

$$j = a_{N-1}(N-1)! + a_{N-2}(N-2)! + \cdots + a_2 2! + a_1 1! \qquad 0 \le a_i \le i \quad (i = 1, 2, \ldots, N - 1).$$

(2) Starting from the identity permutation $1, 2, \ldots, N$, move the $(i + 1)$th element a_i positions leftward for $i = 1, 2, \ldots, N - 1$. The result is the jth permutation.

The correctness of algorithm A follows from the correctness proof of algorithm B below.

Example. Evaluate permutation no. $j = 1\,000\,000$ of the PEP scheme for $N \ge 10$.

(1) $1\,000\,000 = 2 \times 9! + 6 \times 8! + 6 \times 7! + 2 \times 6! + 5 \times 5! + 1 \times 4! + 2 \times 3! + 2 \times 2! + 0 \times 1!$

(2) The identity permutation: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots, N$.

Element	Positions moved	Result
2	0	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, . . . N . . .
3	2	3, 1, 2, 4, 5, 6, 7, 8, 9, 10, . . . N . . .
4	2	3, 4, 1, 2, 5, 6, 7, 8, 9, 10, . . . N . . .
5	1	3, 4, 1, 5, 2, 6, 7, 8, 9, 10, . . . N . . .
6	5	6, 3, 4, 1, 5, 2, 7, 8, 9, 10, . . . N . . .
7	2	6, 3, 4, 1, 7, 5, 2, 8, 9, 10, . . . N . . .
8	6	6, 8, 3, 4, 1, 7, 5, 2, 9, 10, . . . N . . .
9	6	6, 8, 9, 3, 4, 1, 7, 5, 2, 10, . . . N . . .
10	2	6, 8, 9, 3, 4, 1, 7, 10, 5, 2, . . . N . . .

Hence the permutation numbered 1 000 000 (starting the enumeration from 0) is: 6, 8, 9, 3, 4, 1, 7, 10, 5, 2, 11, . . . , N. . . . Obtaining conversely, the serial number j of a permutation Π_j generated by the PEP scheme, is a special case of algorithm C below.

3. LIMITED MEMORY SPACE

Since $N!$ grows exponentially, the above described scheme can be used for small N only. (Already $11! = 39\,916\,800$ exceeds the memory space of all current computers.)

In this section a slight modification of the PEP scheme will be given, which enables to run it on t processors when $t = s!$ for any s, and where every processor needs to store only one permutation at a time. (In the next section the restriction $t = s!$ will be removed at the price of some additional bookkeeping.)

Definitions

A *direction pointer* is a pointer attached to every element m in the range $s < m \leq N$, pointing to the left or to the right, denoting the direction in which the element m should be moved. A *movable element m* is every element $s < m \leq N$ such that there exists an element k, $1 \leq k < m$ in the direction of the direction pointer of m. The *smallest movable element* is a movable element m_s such that for any other movable element m, $m_s < m$.

The PEP-algorithm

Initialization. Assign the permutation with serial number j to processor p_j, $j = 0, \ldots, s! - 1$ by a direct serial process such as algorithm A, or by the naïve PEP scheme outlined in Section 2. Set all direction pointers pointing left.

Generation. Interchange the smallest movable element m_s with the nearest element k satisfying $1 \leq k < m_s$ in the direction of the pointer of m_s, simultaneously in all $s!$ processors.

Change the direction of the pointer in all elements l satisfying $s < l < m_s$. Assign to every newly generated permutation the serial number of the permutation from which it was created, increased by $s!$. If there is no movable element, stop.

Examples of using this algorithm for $N = 4$ and $s = 2$ and $s = 3$ are given in Tables 2 and 3, respectively.

At each step $s!$ permutations are generated simultaneously, one in every processor. We enumerate the permutations (here and below) starting from the first permutation of the first processor, then the first permutation of the second processor etc. (as in Tables 2 and 3).

The algorithm has the following properties:

(a) Each permutation is generated by a single interchange of two elements, not necessarily adjacent.
(b) The kth subset of permutations is generated before the $(k + 1)$th element is moved.
(c) The sequence of permutations is reflection-free, (follows from (b)), but the jth permutation is not necessarily a reflection of the $(N! - j - 1)$th permutation (see e.g. Table 2).

Table 2. Generating all permutations of four elements by 2! SIMD processors ($s = 2$)

	p_0					p_1			
0)	1	2	$\overleftarrow{3}$	$\overleftarrow{4}$	1)	2	1	$\overleftarrow{3}$	$\overleftarrow{4}$
2)	1	$\overleftarrow{3}$	2	$\overleftarrow{4}$	3)	2	$\overleftarrow{3}$	1	$\overleftarrow{4}$
4)	$\overleftarrow{3}$	1	2	$\overleftarrow{4}$	5)	$\overleftarrow{3}$	2	1	$\overleftarrow{4}$
6)	$\overrightarrow{3}$	1	$\overleftarrow{4}$	2	7)	$\overrightarrow{3}$	2	$\overleftarrow{4}$	1
8)	1	$\overrightarrow{3}$	$\overleftarrow{4}$	2	9)	2	$\overrightarrow{3}$	$\overleftarrow{4}$	1
10)	1	2	$\overleftarrow{4}$	$\overrightarrow{3}$	11)	2	1	$\overleftarrow{4}$	$\overrightarrow{3}$
12)	1	$\overleftarrow{4}$	2	$\overleftarrow{3}$	13)	2	$\overleftarrow{4}$	1	$\overleftarrow{3}$
14)	1	$\overleftarrow{4}$	$\overleftarrow{3}$	2	15)	2	$\overleftarrow{4}$	$\overleftarrow{3}$	1
16)	$\overleftarrow{3}$	$\overleftarrow{4}$	1	2	17)	$\overleftarrow{3}$	$\overleftarrow{4}$	2	1
18)	$\overleftarrow{4}$	$\overrightarrow{3}$	1	2	19)	$\overleftarrow{4}$	$\overrightarrow{3}$	2	1
20)	$\overleftarrow{4}$	1	$\overrightarrow{3}$	2	21)	$\overleftarrow{4}$	2	$\overrightarrow{3}$	1
22)	$\overleftarrow{4}$	1	2	$\overrightarrow{3}$	23)	$\overleftarrow{4}$	2	1	$\overrightarrow{3}$

Table 3. Generating all permutations of four elements by 3! = 6 SIMD processors ($s = 3$)

	p_0					p_1					p_2			
0)	1	2	3	$\overleftarrow{4}$	1)	2	1	3	$\overleftarrow{4}$	2)	1	3	2	$\overleftarrow{4}$
6)	1	2	$\overleftarrow{4}$	3	7)	2	1	$\overleftarrow{4}$	3	8)	1	3	$\overleftarrow{4}$	2
12)	1	$\overleftarrow{4}$	2	4	13)	2	$\overleftarrow{4}$	1	3	14)	1	$\overleftarrow{4}$	3	2
18)	$\overleftarrow{4}$	1	2	3	19)	$\overleftarrow{4}$	2	1	3	20)	$\overleftarrow{4}$	1	3	2
	p_3					p_4					p_5			
3)	2	3	1	$\overleftarrow{4}$	4)	3	1	2	$\overleftarrow{4}$	5)	3	2	1	$\overleftarrow{4}$
9)	2	3	$\overleftarrow{4}$	1	10)	3	1	$\overleftarrow{4}$	2	11)	3	2	$\overleftarrow{4}$	1
15)	2	$\overleftarrow{4}$	3	1	16)	3	$\overleftarrow{4}$	1	2	17)	3	$\overleftarrow{4}$	2	1
21)	$\overleftarrow{4}$	2	3	1	22)	$\overleftarrow{4}$	3	1	2	23)	$\overleftarrow{4}$	3	2	1

Algorithm B for obtaining the jth permutation in the PEP algorithm

Let the $N!$ permutations be numbered 0 to $N! - 1$.

(1) Represent j by its factorial representation:

$$j = a_{N-1}(N-1)! + a_{N-2}(N-2)! + \cdots + a_2 2! + a_1 1!$$
$$0 \leq a_i \leq i \quad (i = 1, \ldots, N-1).$$

(2) For all $s \leq i \leq N - 1$ evaluate

$$u_i \equiv \left\lfloor \frac{j}{(i+1)!} \right\rfloor \pmod 2, \quad u_i = 0 \text{ or } 1.$$

(3) For $i = 1$ to $N - 1$, do the following:
Starting from the identity permutation, if $i < s$ or $u_i = 0$, then move the $(i + 1)$th element a_i positions leftward. Otherwise, ($i \geq s$ and $u_i = 1$) move the $(i + 1)$th element $i - a_i$ positions leftward.
The result is the jth permutation.

Note that for $s = N$ we get back algorithm A, since then $i < s$ for all $1 \leq i < N$.

Example. $N \geq 4, j = 19$.

(1) $19 = 3 \times 3! + 0 \times 2! + 1 \times 1!$,
$a_1 = 1, a_2 = 0, a_3 = 3.$

(2a) If $s = 2$,

$$u_2 \equiv \left\lfloor \frac{19}{3!} \right\rfloor \equiv 1 \pmod 2$$
$$u_3 \equiv \left\lfloor \frac{19}{4!} \right\rfloor \equiv 0 \pmod 2.$$

The identity permutation 1 2 3 4 5 . . . N.

Element	Positions moved	Result
2	1	2 1 3 4 5 . . . N
3	$2-0=2$	3 2 1 4 5 . . . N
4	3	4 3 2 1 5 . . . N

Therefore for $s=2$, the permutation for $j=19$ is 4 3 2 1 5 . . . N (see Table 2).

(2b) If $s=3$, only u_3 is pertinent.

Element	Positions moved	Result
2	1	2 1 3 4 5 . . . N
3	0	2 1 3 4 5 . . . N
4	3	4 2 1 3 5 . . . N

Therefore for $s=3$, the permutation for $j=19$ is: 4 2 1 3 5 . . . N (see Table 3).

Correctness proof of algorithm B. We shall establish a one-to-one correspondence, based on the factorial number scheme, between the permutations generated by the PEP algorithm and the permutations generated by algorithm B.

In the factorial number representation, digit a_i assumes integer values in the range $[0, i]$, and is incremented by 1 exactly after all the $i!$ possible increments of the digits a_k, $1 \le k < i$. In the PEP algorithm, element $i+1$, $s < i+1 \le N$, is moved alternately i times leftward and then i times rightward, since there are i elements smaller than $i+1$. It is moved exactly once after all $i!$ permutations of the elements $\{1, 2, \ldots, i\}$ were created, otherwise $i+1$ is not the smallest movable element. Element $i+1$ is alternately either in a 'left sweep mode' or in a 'right sweep mode', according to the indication of the direction pointer. The direction is changed when element $i+1$ has completed i moves, completing $(i+1)!$ permutations. Hence a correspondence between the $(i+1)$th element and digit a_i in the factorial representation of integers can be established. In particular, for permutation j generated by the PEP algorithm, digit a_i of the factorial representation of j, $j=\sum_{i=1}^{N-1} a_i i!$, represents the number of positions element $(i+1)$ was moved in the PEP algorithm. The direction of its movement is determined by the parity of $u_i = \lfloor j/(i+1)! \rfloor$ and since u_i starts from an even value (0), and $i+1$ starts with a leftward move, therefore, if u_i is even, $(i+1)$ is in a left sweep mode, and if u_i is odd, then $(i+1)$ is in a right sweep mode. Therefore the position of every element in a specific permutation can be determined as described above, and this is exactly performed by algorithm B.

A similar reasoning can be applied and the same correspondence can be established for $1 \le i+1 \le s$, but in this case every element $i+1$ performs in the PEP algorithm only a single left sweep, therefore there is no need to check the parity of the corresponding u_i.

Conclusion. If $u_i \equiv 0 \pmod 2$ $(1 \le i < N)$, that is, all elements are in a left-sweep mode, then the serial numbers of the permutations in the unrestricted case—the naïve PEP scheme (Table 1), and in the PEP algorithm (Tables 2 and 3) are the same (e.g. permutations 0–5 and 12–17).

Algorithm C for obtaining the serial number of a permutation Π generated by the PEP algorithm

Let Π be a given permutation, generated by the PEP algorithm, and denote by a_i the position of element i in Π $(1 \le i \le N)$. The serial number of Π is calculated as follows.

```
(1) Initialization
    (a) $l \leftarrow \max_{a_i \neq i, a_j = j, j > i} (i)$, i.e. $l$ is the largest element
        that is moved in Π from its original position in the
        identity permutation.
    (b) $d_l \leftarrow l - a_l$, i.e. the number of positions $l$ is moved.
    (c) $S \leftarrow d_l(l-1)!$.
    (d) Delete all elements $j \ge l$ from Π and renumber Π
        (i.e. evaluate $a_1, \ldots, a_{l-1}$).
(2) For $i = l-1$ to 2 by $-1$ do
        If $i > s$ then $t \leftarrow d_{i+1} \pmod 2$
        $d_i \leftarrow \begin{cases} i - a_i & \text{if } i \le s \text{ or } t = 0 \\ a_i - 1 & \text{otherwise } (i > s \text{ and } t = 1) \end{cases}$
    $S \leftarrow S + d_i(i-1)!$
    Delete $i$ from Π and renumber Π
    end
    Finally $S$ contains the serial number of Π.
```

Example. Π = 4 2 1 3 5 6 7

```
(1) (a) $l \leftarrow 4$
    (b) $d_4 \leftarrow 3$
    (c) $S \leftarrow 3 \times 3! = 18$
    (d) Π = 2 1 3
        $a_1 = 2, a_2 = 1, a_3 = 3.$
(2)     $\underline{s = 2}$
    $\underline{i = 3 \quad (i > s)}$
    $t \leftarrow d_4 \pmod 2 = 1$
    $d_3 \leftarrow a_3 - 1 = 2$
    $S \leftarrow S + d_3 2! = 22$
      Π = 2 1
    $a_1 = 2, a_2 = 1$

    $\underline{i = 2}$
    $d_2 \leftarrow 2 - a_2 = 1$
    $S \leftarrow S + d_2 1! = 23$
      Π = 1

        $\underline{s = 3}$
    $\underline{i = 3 \quad (i \le s)}$
    $d_3 \leftarrow 3 - a_3 = 0$
    $S \leftarrow S + d_3 2! = 18$
      Π = 2 1
    $a_1 = 2, a_2 = 1$

    $\underline{i = 2}$
    $d_2 \leftarrow 2 - a_2 = 1$
    $S \leftarrow S + d_2 1! = 19$
      Π = 1.
```

Therefore the serial number of Π is 23 if $s=2$ (compare with Table 2) or 19 if $s=3$ (compare with Table 3).

Remark. In the unrestricted case (the naïve PEP-algorithm) always $i \leq s$, $1 \leq i \leq N$, and therefore Step 2 can be simplified accordingly.

Correctness proof of Algorithm C. An element i, $2 \leq i \leq N$ is moved in the naïve PEP scheme and in the PEP algorithm if and only if all elements $1 \leq k < i$ have completed all their possible $(i-1)!$ permutations. Hence if $d_i (2 \leq i \leq N)$, denotes the number of positions element i was moved to obtain the permutation Π, then the serial number of Π is obviously $\sum_{i=2}^{N} d_i (i-1)!$.

In order to obtain the values of d_i: For $2 \leq i < s$ or $i = N$, element i is moved only leftward, hence the position a_i of element i in Π determines $d_i = i - a_i$.

For $s \leq i \leq N-1$: Every time element $i+1$ is moved by one position, element i has completed a leftward or rightward sweep, and changed the direction of its movement. Hence the direction of the movement of element i can be determined by the parity of d_{i+1}.

If d_{i+1} is even, element i is in a leftsweep mode and therefore $d_i = i - a_i$ (as $2 \leq i \leq s$ or $i = N$). If d_{i+1} is odd, then element i is in a rightsweep mode, therefore $d_i = a_i - 1$.

A variation of the well-known Johnson–Trotter permutation generation algorithm,[11,12] as described by Even (Ref. 13, p. 3), is a special case of the PEP algorithm. If $s = 1! = 1$, that is only a single processor is available, the sequence of permutations generated by the PEP algorithm is shown in Table 4. This special case is not the same as the Johnson–Trotter algorithm since the smallest rather than the largest element is always moved, and therefore non-adjacent exchanges are also performed. On the other hand, this algorithm has the above mentioned properties which do not exist in the Johnson–Trotter algorithm. Especially important is the property of generating the kth subset of permutations before moving the $(k+1)$th element. This enables one to start with the same permutation sequence for any number of elements, and to continue the process if more permutations are needed.

Table 4. Generating all permutations of four elements by 1! = 1 processor

1	$\overleftarrow{2}$	$\overleftarrow{3}$	$\overleftarrow{4}$	1	$\overrightarrow{3}$	$\overleftarrow{4}$	$\overleftarrow{2}$	$\overleftarrow{3}$	$\overleftarrow{4}$	1	$\overleftarrow{2}$
$\overleftarrow{2}$	1	$\overleftarrow{3}$	$\overleftarrow{4}$	$\overleftarrow{2}$	$\overrightarrow{3}$	$\overleftarrow{4}$	1	$\overleftarrow{3}$	$\overleftarrow{4}$	$\overleftarrow{2}$	1
$\overrightarrow{2}$	$\overleftarrow{3}$	1	$\overleftarrow{4}$	$\overrightarrow{2}$	1	$\overleftarrow{4}$	$\overrightarrow{3}$	$\overleftarrow{4}$	$\overrightarrow{3}$	$\overrightarrow{2}$	1
1	$\overleftarrow{3}$	$\overrightarrow{2}$	$\overleftarrow{4}$	1	$\overrightarrow{2}$	$\overleftarrow{4}$	$\overrightarrow{3}$	$\overleftarrow{4}$	$\overrightarrow{3}$	1	$\overrightarrow{2}$
$\overleftarrow{3}$	1	$\overleftarrow{2}$	$\overleftarrow{4}$	1	$\overleftarrow{4}$	$\overleftarrow{2}$	$\overleftarrow{3}$	$\overleftarrow{4}$	1	$\overrightarrow{3}$	$\overleftarrow{2}$
$\overleftarrow{3}$	$\overleftarrow{2}$	1	$\overleftarrow{4}$	$\overleftarrow{2}$	$\overleftarrow{4}$	1	$\overleftarrow{3}$	$\overleftarrow{4}$	$\overleftarrow{2}$	$\overrightarrow{3}$	1
$\overrightarrow{3}$	$\overrightarrow{2}$	$\overleftarrow{4}$	1	$\overrightarrow{2}$	$\overleftarrow{4}$	$\overleftarrow{3}$	1	$\overleftarrow{4}$	$\overrightarrow{2}$	1	$\overrightarrow{3}$
$\overrightarrow{3}$	1	$\overleftarrow{4}$	$\overrightarrow{2}$	1	$\overleftarrow{4}$	$\overleftarrow{3}$	$\overrightarrow{2}$	$\overleftarrow{4}$	1	$\overrightarrow{2}$	$\overrightarrow{3}$

4. LIMITED NUMBER OF PROCESSORS

In the previous section, the number of processors used for the generation of permutations was $t = s!$ for some $s < N$. This restriction can be removed at the price of additional initializing of the processors and/or not using a few of them. (In the above described algorithm, where the number of processors is $s!$, every processor is initialized only once.)

In order to achieve this, the following method can be used: For any t processors, if there is no integer s such that $t = s!$, then the smallest $r \leq t$ should be found such that t divides $r!$. Now execute the above described algorithm as if $r!$ processors were available, except that since only t processors are available, the algorithm must be repeated $d = r!/t$ times, that is, each processor must be initialized d times.

Example

If the number of available processors is $t = 3$, then there is no integer s such that $t = s!$. In this case the smallest integer $r \leq t$ such that t divides $r!$ is $r = 3$. Hence the PEP algorithm can be applied as if $r! = 6$ processors are available but the algorithm must be repeated $r!/t = 2$ times, i.e. every processor must be initialized twice. This is illustrated by Table 3, where the six processors can be substituted by three processors initialized twice.

In the example $r = t$, but for $t = 10$, $r = 5$, so $r < t$.

Proposition 1

For any two integers r and t, write $t = \prod_{i=1}^{n} q_i^{\alpha_i}$ where the q_i are the distinct prime factors of t and $\alpha_i \geq 1$ are their multiplicity. Then $t|r!$ if and only if $1 \leq \alpha_i \leq \sum_{k>0} \lfloor r/q_i^k \rfloor$ $(1 \leq i \leq n)$.

Proposition 2

For any positive integer t, let r denote the smallest positive integer such that $t|r!$. Then $r \leq t$ and equality holds if and only if t is prime or $t = 1$ or $t = 4$.

Conclusion. For any composite integer $t > 1$, $t = \prod_{i=1}^{n} q_i^{\alpha_i}$, the smallest integer r such that $t|r!$ can be determined as follows: Choose the smallest r_1 such that $\alpha_{m_1} \leq \sum_{k>0} \lfloor r_1/q_{m_1}^k \rfloor$, where m_1 is determined by $\alpha_{m_1} q_{m_1} = \max_{j=1,\ldots,n} \{\alpha_j q_j\}$. If $r_1 \geq \alpha_j q_j$, $j = 1, \ldots, n$, $j \neq m_1$ then $r = r_1$. Otherwise repeat the process for $\{\alpha_j q_j\} - \{\alpha_{m_1} q_{m_1}\}$, to obtain r_2 etc. until $r_i > \alpha_j q_j$, $2 \leq i \leq n, j \neq m_1, \ldots, m_{i-1}$, and set $r = \max\{r_1, r_2, \ldots, r_i\}$.

The proofs are based on simple arguments from number theory. They are omitted here but can be found in Ref. 14.

For all practical r, say for $r \leq 20$ $(21! > 2^{64} > 20!)$ a table of $r!$ can be constructed, and then after at most 20 steps the smallest r such that $t|r!$ can be found. Another practical way, suggested by the referee, is to express 20! in terms of its prime decomposition: $20! = 2^{18} \cdot 3^8 \cdot 5^4 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$, and to compare this with the prime decomposition of t. Simple inspection will indicate immediately whether or not $t|20!$. Then divisions by 20, 19, 18, 17, ... will reveal the minimum r such that $t|r!$.

If t is not a prime number or a multiple of large primes, $r \ll t$. In actual SIMD-machines t is usually a power of 2 (or of small prime factors), and never a large prime. Therefore usually a small r as requested can be found.

An obvious extension is the following: given any $t \neq s!$, let $\tau \in [t - \Delta, t]$ be the actual number of processors used such that r is small, where Δ is a small integer. For example, for $t = 50$, $r = 10$, for $t = 49$, $r = 14$ but for $t = 48$, $r = 6$. The reason for this is clearly because 48 has only small prime factors. Therefore by ignoring two processors, the active processors must be initialized only $6!/48 = 15$ times instead of $10!/50 = 72576$ times (but in every step only 48 instead of 50 permutations are generated).

REFERENCES

1. R. Sedgewick, Permutation generation methods. *Computing Surveys* **9**, 137–164 (1977).
2. M. K. Roy, Evaluation of permutation algorithms. *The Computer Journal* **21**, 296–301 (1978).
3. M. J. Flynn, Some computer organizations and their effectiveness. *IEEE Transactions on Computers* **C-21**, 948–960 (1972).
4. H. T. Kung, The structure of parallel algorithms, in *Advances in Computers*, Vol. 19, ed. by M. C. Yovits, pp. 65–112. Academic Press, New York (1980).
5. R. M. Russel, The CRAY-1 computer system. *Communications of the ACM* **21**, 63–72 (1978).
6. E. W. Kozdrowicki and D. J. Theis, Second generation of vector supercomputers. *Computer* **13** (11), 71–83 (1980).
7. R. J. Ord-Smith, Generation of permutation sequences, part 2. *The Computer Journal* **14**, 136–139 (1971).
8. M. B. Wells, *Elements of Combinatorial Computing*, Pergamon Press, New York (1971).
9. J. K. Lenstra, Recursive algorithms for enumerating subsets, lattice points, combinations and permutations. *Report BW 28/73*, Matematisch Centrum, Amsterdam.
10. D. E. Lehmer, The machine tools of combinatorics, in *Applied Combinatorial Mathematics*, ed. by E. F. Beckenbach, pp. 5–31. Wiley, New York (1964).
11. S. M. Johnson, Generation of permutations by adjacent transpositions. *Mathematics of Computation* **17**, 282–285 (1963).
12. H. F. Trotter, Algorithm 115. *Communications of the ACM* **5**, 434–435 (1962).
13. S. Even, *Algorithmic Combinatorics*, MacMillan Company, New York (1973).
14. M. Mor and A. S. Fraenkel, Permutation generation on vector processors, *Technical Report CS81-10*, The Weizmann Institute of Science, Rehovot (June 1981).
15. F. M. Ives, Permutation enumeration, *Communications of the ACM* **19**, 68–72 (1976).
16. D. J. Kuck, A survey of parallel machine organization and programming. *Computing Surveys* **9**, 29–59 (1977).

Received June 1981