Information Processing Letters 35 (1990) 167–170
North Holland

7 August 1990

PARALLEL ALGORITHM FOR GENERATING PERMUTATIONS ON LINEAR ARRAY

Chau-Jy LIN

Department of Applied Mathematics, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

Communicated by K. Ikeda
Received 31 January 1990
Revised 9 April 1990

Keywords: Parallel algorithm, linear array, permutations, ranking function

1. Introduction

Given n items, a parallel algorithm for generating the $n!$ permutations is presented. This algorithm is designed to run on a linear array consisting of n identical processing elements (PEs for short). These PEs are numbered and referred to as PE(i) for $1 \leqslant i \leqslant n$. Each PE is responsible for producing one component of each permutation. Every PE(i) contains six registers, namely: $C(i)$, $T(i)$, $K(i)$, $Y(i)$, $Z(i)$ and $X(i)$. A permutation takes constant time from the preceding one. For generating the required $n!$ permutations, our algorithm requires $n! + n - 1$ time steps in which $n - 1$ initial time steps are included. Here, a time step is defined as the maximal elapsed time (considering all PEs) to do the following three tasks: (1) PE(i) receives two data from PE($i + 1$); (2) PE(i) executes the design algorithm once; (3) PE(i) sends the ith component of a certain permutation to output terminal. This ith component is stored in the register $C(i)$.

In the literature, there are many methods used to produce the $n!$ permutations sequentially, e.g. the methods of lexicographic order, inversion vectors, rotations, minimal changes, random generations, and by reversing a certain suffix of current permutation, see [6,7]. A parallel algorithm is given in [2] to generate the permutations. However, this algorithm is not optimal (in the sense that the number of PE times the time complexity does not match the optimal sequential running time) and each PE needs a stack of size n. The parallel algorithm in [1] uses an arbitrary number of independent PEs, each producing a part of consecutive permutations. This algorithm is optimal, but each PE requires storage size O(n) and has to deal with large integers. The parallel algorithm in [3] running on a SIMD computer is complicated because it requires a table to handle a binary tree. An efficient parallel algorithm for generating all the $n!$ permutations is presented in [5] which is executed on vector processors. This algorithm requires a large memory size. In contrast, our algorithm requires O($n!$) time steps and runs on a linear array in which each PE contains only constant storage size and the elapsed time of a time step is independent of n.

2. The design for generating permutations

Without loss of generality, the n items are denoted by $1, 2, \ldots, n$. We write $per(n)$ to denote the set of the $n!$ permutations which are produced by our algorithm with the order as they are generated. The basic idea for designing our algorithm is the use of iterative method and adding modulo operation. The design consideration is described as below.

(1) For any B in $per(n-1)$, we append the number n to B to form a permutation A in $per(n)$. We denote $A = \{a_1, a_2, \ldots, a_n\}$ and assume that A is produced at time step t. For

$1 \leqslant j \leqslant n-1$, let $A_j = \{a_1^j, a_2^j, \ldots, a_n^j\}$ be the $n-1$ permutations in $per(n)$ to be produced at time steps $t+j$. For $1 \leqslant i \leqslant n$, PE($i$) calculates the ith components a_i^j of A_j with the statement: **if** $a_i + j > n$ **then** $a_i^j = a_i + j - n$ **else** $a_i^j = a_i + j$. These n permutations A and A_j are called in a *cycle-n-perms* with the *header* A. The register $K(i)$ keeps a counter indicating how many permutations has generated within a cycle-n-perms.

(2) During the generation of a current cycle-n-perms with the header $A = \{a_1, a_2, \ldots, a_{n-1}, a_n = n\}$, the next header (say A') for the next cycle-n-perms must be prepared in time. That is, a permutation B' in $per(n-1)$ is also produced before it can be used to form the header A'. The preparation of $B' = \{b_1, b_2, \ldots, b_{n-1}\}$ is considered as follows. Let $a_i^{(0)} = a_i$ for $1 \leqslant i \leqslant n$ and k be a variable with initial value 0. We perform the procedure

while $a_{n-k}^{(k)} = n-k$ **do**
  begin
    $k := k+1$;
    for $i = 1$ **to** $n-k$ **do** $a_i^{(k)} := a_i^{(k-1)} + 1$
  end.

After executing this while-do loop, the required B' is ready with its components $b_i = a_i^{(k)}$ for $1 \leqslant i \leqslant n-k$ and $b_i = i$ for $n-k+1 \leqslant i \leqslant n-1$.

Now we present the consideration to prepare the header A' in our algorithm. Once the header A is produced, a signal for preparing new header A' is issued from PE(n) to PE(i) for $1 \leqslant i \leqslant n-1$. This signal causes two data to be sent out from PE(n). One is an updating information with the number $n-1$ which is put in $Y(n)$ and then it should be propagated to the register $Y(i)$ for $1 \leqslant i \leqslant n-1$. The other is a flag which indicates that PE(n) has sent out an updating information by assigning 1 to register $Z(n)$. For $1 \leqslant i \leqslant n-1$, when PE($i$) receives an updating information from PE($i+1$) by receiving $Y(i+1)$, PE(i) begins to evaluate the ith component of A' and stores this component in register $T(i)$. Then PE(i) determines whether it should issue another signal to PE($i-1$). The condition for PE(i) to issue a signal is that PE($i+1$) has issued its signal and PE(i) has the value i in its $T(i)$. Once this condition is true, PE(i) sends its updating information with the number $i-1$ in $Y(i)$ and its flag in $Z(i)$ within two consecutive time steps. Thus, in order to delay this time step of sending out the flag, PE(i) needs an assignment statement to translate its flag from $Z(i)$ to register $X(i)$.

(3) We continue the above two design considerations until PE(1) has issued an updating information.

3. The parallel algorithm

From the previous description, a parallel algorithm for generating the $n!$ permutations is presented with name GENPER(n). Here, we assume that the registers with names $Y(n+1)$ and $X(n+1)$ are in the memory of the host computer. We have $Y(n+1) = 0$ at all time steps and we also assume $X(n+1) = 1$ at the time step $t = q * n$ for $0 \leqslant q \leqslant (n-1)!$ and $X(n+1) = 0$ at the remaining time steps. The algorithm GENPER(n) is executed in a skew form, that is, PE(i) has $i-1$ time steps delay before it begins to evaluate the ith component of the first permutation.

Algorithm GENPER(n) ≡
[*Initial state*]
  For $1 \leqslant i \leqslant n$, set $C(i) = i$, $K(i) = n$, $T(i) = i$, $X(i) = Z(i) = Y(i) = 0$ in PE(i) at time step $n-i$.

[*Execution state*]
  begin
    repeat / * **do parallel for all** PE(i). * /
      if $K(i) < n$ **then**
        evaluating-perms-after-header
      else *evaluating-header*;
      output $C(i)$;
      $X(i) := Z(i)$;
      if $Y(i+1) \neq 0$ **then**
        evaluating-next-header
      else $Y(i) := 0$;
      if $X(i+1) = 1$ and $T = i$ **then**
        issuing-updating-signal
      else $Z(i) := 0$
    until $X(1) = 1$
  end.

evaluating-perms-after-header ≡
begin
if $C(i) < n$ **then**
$C(i) := C(i) + 1$
else $C(i) := 1$; $K(i) := K(i) + 1$
end
evaluating-header ≡
begin $C(i) := T(i)$; $K(i) := 1$ **end**
evaluating-next-header ≡
begin
if $T(i) < Y(i+1)$ **then**
$T(i) := T(i) + 1$
else $T(i) := 1$; $Y(i) := Y(i+1)$
end
issuing-updating-signal ≡
begin $Y(i) = i - 1$; $Z(i) := 1$ **end**

4. The index of *per(n)*

We give the ranking and the unranking functions of $per(n)$. Let A_1, A_2 be two permutations in $per(n)$ which are generated at time steps t_1, t_2 respectively. We define $A_1 < A_2$ if and only if $t_1 < t_2$. The one-to-one correspondence f between the $per(n)$ and the set of integers $\{0, 1, 2, \ldots, n! - 1\}$ has the property: $A < B$ if and only if $f(A) < f(B)$. The function f and its inverse function are called the ranking (or index) and unranking function of $per(n)$, respectively. Their corresponding evaluations are derived as follows.

For ranking, let $A = \{a_1, a_2, \ldots, a_n\}$ be any given permutation in $per(n)$ with the initial index$(\{1\}) = 0$. The index of A will be evaluated recursively. If $a_n = n$, we have the formula: index$(A) = n *$ index$(\{a_1, a_2, \ldots, a_{n-1}\})$. If $a_n < n$, first we compute the permutation in $per(n-1)$ to obtain $B = \{a_1 - a_n, a_2 - a_n, \ldots, a_{n-1} - a_n\}$ with subtraction modulo n. Then we have the formula: index$(A) = n *$ index$(B) + a_n$.

As for unranking, given any integer γ such that $0 \leqslant \gamma \leqslant n! - 1$. We evaluate a sequence $C = \{c_1 = 0, \ldots, c_{j-1} = 0, c_j, c_{j+1}, \ldots, c_n\}$ by the following process. There exist two nonnegative integers q_n, c_n such that $\gamma = n * q_n + c_n$ for $0 \leqslant c_n \leqslant n - 1$. If $q_n = 0$, then the evaluation of the sequence C stops; otherwise we continue to find two nonnegative integers q_{n-1}, c_{n-1} such that $q_n = (n-1) * q_{n-1} + c_{n-1}$ for $0 \leqslant c_{n-1} \leqslant n - 2$. We continue this process until there exists an integer j such that $q_j = 0$ and $q_{j+1} = c_j$ for $0 \leqslant c_j \leqslant j - 1$. The minimum integer of j is 2 because of $\gamma \leqslant n! - 1$.

Now we calculate the desired permutation A with index γ. Initially, let $A_{j-1} = \{1, 2, \ldots, j-1\}$. Suppose that there exists an integer i for $1 \leqslant i \leqslant n - 1$ such that the $A_i = \{a'_1, a'_2, \ldots, a'_i\}$ is determined. For $1 \leqslant k \leqslant i$ if $c_{i+1} \neq 0$, we assign $a'_k + c_{i+1} (\mathrm{mod}\ i+1)$ to a_k and let $a_{i+1} = c_{i+1}$ to form $A_{i+1} = \{a_1, a_2, \ldots, a_{i+1}\}$. If $c_{i+1} = 0$, then we get A_{i+1} by appending the number $i+1$ to A_i, i.e., $a_k = a'_k$ for $1 \leqslant k \leqslant i$ and $a_{k+1} = i + 1$. We continue this evaluation until the final permutation $A_n = A$ is obtained.

5. The correctness proof

The essential part in our algorithm is that during the execution of current cycle-n-perms with a header (say A), the next header for the next cycle-n-perms must be prepared in time. Suppose the header of the current cycle-n-perms is $A = \{a_1, a_2, \ldots, a_{n-1}, a_n = n\}$, then the next header $A' = \{a'_1, a'_2, \ldots, a'_{n-1}, a'_n = n\}$ will be prepared within the time steps that the current cycle-n-perms are generating.

Proposition 1. PE(i) *sets* $Z(i) = 1$ *only if* PE$(i+1)$ *has set* $Z(i+1) = 1$.

Proof. By the two statements "$X(i) := Z(i)$" and "**if** $X(i+1) = 1$ and $T(i) = i$ **then** *issuing-updating-signal* **else** $Z(i) := 0$". □

Proposition 2. *The components of the header in* $per(n)$ *for the next cycle-n-perms are prepared in time.*

Proof. Let $A = \{a_1, a_2, \ldots, a_n\}$ be the header of the current cycle-n-perms and $A' = \{a'_1, a'_2, \ldots, a'_n\}$ be the next header of the next cycle-n-perms. We wish to show that the a'_i is ready at time step $t = t_0 + 2(n - i)$, where t_0 is the time step such that GENPER(n) produces the nth component of header A. From the discussion in

the previous section and the use of mathematical induction, we obtain $t_0 = q * n + 1$ for $q = 0, 1, 2, \ldots, (n-1)!$. For $1 \leqslant i \leqslant n-1$, PE($i$) calculates the content of its $T(i)$ from $t = t_0 + (n - i)$ to $t = t_0 + 2(n - i)$. The final value of $T(i)$ (i.e., a_i') is ready before the time step $t = t_0 + 2(n - i)$. Since the generation of ith components of any fixed permutation has $n - i$ time steps delay in GENPER(n) (because of a skew form for the first permutation to be generated), and a cycle-n-perms has n permutations to be produced, a_i' is retrieved at $t_0 + (n - i) + n = t_0 + 2n - i$. Therefore, A' has prepared before it is generated by the fact of $t_0 + 2n - i > t_0 + 2(n - i)$. □

From the adding modulo operation in GENPER(n), we have the following propositions.

Proposition 3. *Within any cycle-n-perms, the n permutations are different.*

Proposition 4. *Any two permutations being generated within any two different cycle-n-perms are distinct.*

Theorem 5. *The parallel algorithm GENPER(n) generates $n!$ permutations correctly.*

Proof. By induction on n under the use of the ranking function as shown in the previous section. □

Since we have $n - 1$ time steps delay before the components of the first permutation $\{1, 2, 3, \ldots, n\}$ to be produced, and there are $n!$ permutations to be generated, the execution of GENPER(n) requires $n! + (n - 1)$ time steps.

6. Conclusion

In this paper we have presented a parallel algorithm to generate all the $n!$ permutations of n given items. The computational model used is a linear array consisting of n PEs. Since each PE is identical and executes the same program, it is suitable for VLSI implementation. In [4] a parallel algorithm to generate all the $\binom{m}{n} = m!/(n!(m-n)!)$ combinations is presented. It seems that these two algorithms can be used to generate all the $p_{mn} = m!/(m-n)!$ permutations. Note that for any $B = \{a_1, a_2, \ldots, a_{n-1}\}$ in $per(n-1)$ and $A = \{a_1, a_2, \ldots, a_{n-1}, n\}$, we can modify our linear array so that it gives the permutations after A by loading some adequate initial values. That is, the initial values of $C(i) = i$ of PE(i), $1 \leqslant i \leqslant n$, are replaced by $C(i) = a_i$. Then during the execution of the modified algorithm, the permutation A comes out at the first time step, and any m required permutations following A will be generated within the following m time steps. Furthermore, there exist many combinatorial enumeration problems for which efficient parallel algorithms are yet to be developed. We hope that this consideration of designing parallel algorithm can be used to solve some of these problems in the near future. We are also interested to investigate the parallel algorithms which will be run on a computational model where the storage in any PE and the elapsed time in a time step are independent of the problem size.

References

[1] S.G. Akl, Adaptive and optimal parallel algorithms for enumerating permutations and combinations, *Comput. J.* **30** (1987) 433–436.

[2] G.H. Chen and M.S. Chern, Parallel generation of permutations and combinations *BIT* **26** (1986) 277–283.

[3] P. Gupta and G.P. Bhattackerjee, Parallel generation of permutations, *Comput. J.* **26** (1983) 97–105.

[4] C.J. Lin, A parallel algorithm for generating combinations, *Comput. Math. Appl.* **12** (1989) 1523–1533.

[5] U. Mor and A.S. Fraenkel, Permutation generation on vector processors, *Comput. J.* **25** (1982) 423–428.

[6] R. Sedgewick, Permutation generation method, *Comput. Surveys* **9** (1977) 137–164.

[7] S. Zals, A new algorithm for generation of permutations, *BIT* **24** (1984) 196–204.