Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

Parallel Generation of Permutations

Phalguni Gupta and G. P. Bhattacharjee
Department of Mathematics, Indian Institute of Technology, Kharagpur, India

A parallel algorithm to generate all the ${}^nP_r$ distinct permutations of r items out of n distinct items is presented in this paper. The algorithm requires $O(\lceil {}^nP_r/P \rceil\, r \log_2 r)$ units of time when P processors are available in the SIMD computer.

INTRODUCTION

Parallel algorithms have been designed since the early sixties, when parallel computers were not even available.[1] The interest in designing parallel algorithms for various numerical and non-numerical problems has increased after the development of parallel computers like ILLIAC IV, Cm*, CD-STAR, etc. and implementation of parallel algorithms on these computers.[2–4] This has been sustained since the early seventies due to the realization that the growing demand for the computing power could be met only through parallel computers and parallel algorithms; survey papers[5–7] highlight the growing importance of parallel computers and parallel algorithms. Among the parallel computers, single-instruction-stream–multiple-data-stream (SIMD) computers have been studied most widely.[8] There are several models of SIMD computers. The MCC (mesh connected computer), the CCC (cube connected computer), the PSC (perfect shuffle computer) and the SMM (shared memory model) of the SIMD computer[8] have been extensively studied.

Parallel algorithms have been designed for problems in various fields and for various parallel computers. Hirschberg[9] and Preparata[10] have investigated the sorting problem on the SMM. Parallel algorithms to sort and perform data permutations on an MCC have been presented in Refs 11 and 12. Data permutations on a CCC and a PSC have been considered in Ref. 11. Parallel algorithms for certain graph theoretic and matrix problems on an SMM are available in Ref. 13. The parallel evaluation of arithmetic expressions on the SMM has been studied by Brent[14] and others whereas the parallel evaluation of polynomials has been considered by Munro and Paterson.[15,16]

Generation of permutations is one of the common combinatorial problems. Though parallel algorithms for a large number of numerical and non-numerical problems have been developed, no parallel algorithm to generate permutations has yet been designed. There exist a number of parallel operations in generating permutations; this parallelism may be exploited to design an efficient parallel algorithm for the generation of permutations.

A permutation generating algorithm (PGA) generates ${}^nP_r$ distinct permutations of r items out of n distinct items. Two permutations are considered distinct if they differ with respect to either the items or the order of the items. Several algorithms for serial computers have been designed for generating permutations. The first PGA for the serial computer seems to have been given by Tompkins[17] who has used mixed radix arithmetic in his algorithm. Wells[18] has designed a PGA and has shown the existence of transposition sequences. A PGA with the adjacent transposition sequence has been developed by Johnson.[19] However the fastest PGA based on transposition sequence is due to Trotter.[20] The algorithm due to Boothroyd[21] is an implementation of Wells' sequence, though the algorithm modifies Wells' rules. Ord-Smith[22] has designed a PGA which produces a pseudolexicographic sequence. Algorithms for generating lexicographic permutation sequences have been proposed by Ord-Smith[23] and Phillips.[24] Ehrlich[25] has presented a PGA having no loops. All the algorithms mentioned above have the time complexity of $O(n!)$.

In this paper a parallel algorithm to generate permutation sequences on a SIMD computer is proposed. The assumptions made here about the model of computation are so general that all the models of SIMD computers discussed in the literature[8] satisfy these assumptions. The algorithm is written in an Algol-like language. The paper also presents a modification of Knuth's algorithm[26] to search and insert in a balanced tree using a serial computer. This modified algorithm has been used in the parallel PGA. It has been shown that the parallel PGA has the time complexity of $O(\lceil {}^nP_r/P \rceil r \log_2 r)$ when P processors are available.

2. MODEL OF COMPUTATION

Consider a SIMD computer that consists of P processing elements (PE), a control unit (CU) and an interconnection network (cf. Ref. 27). Each PE consists of a processor with its own local memory (LM), an accumulator, a program counter and a flag indicator indicating whether or not the PE is active. The PEs are indexed 1 to P. The CU broadcasts a single instruction stream to all PEs. The PEs are synchronized and active PEs execute the same instruction under the control of the CU. The interconnection network permits the PEs to communicate among themselves. However no specific interconnection network is assumed for the model. So a SIMD computer with any interconnection network, e.g. the network used in the MCC, the CCC, the PSC, the SMM or other SIMD computers mentioned in the literature, satisfies the assumptions of the model. In fact the parallel algorithm presented in this paper does not require any PEs' communication. The model is illustrated in Fig. 1.

An Algol-like language is used to describe the algorithms. Besides the usual Algol statements the

CCC–0010–4620/83/0026–0097 $04.50

© Wiley Heyden Ltd, 1983

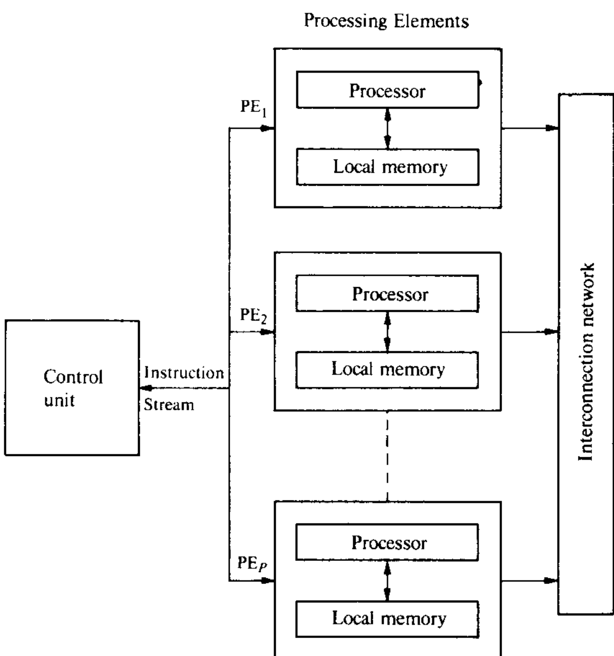


Figure 1. The SIMD computer.

following additional notations and statements are used in describing the algorithms.

(i) Symbols $\lceil x \rceil$ and $\lfloor x \rfloor$ are used to denote the upper integer and the lower integer of the variable x, respectively.
(ii) The symbol $\Leftarrow$ is used to indicate that a variable is taken off a stack. For example, in order to set a link variable L to the address of a new node and to reserve that node for future use, $L \Leftarrow AVAIL$ is used where $AVAIL$ has its usual meaning, i.e. $AVAIL$ refers to the top node of a list of available spaces.
(iii) The symbol $\wedge$ is used to denote the null link.
(iv) Remarks enclosed within braces are considered as comments.
(v) The symbol α refers to the processor index.
(vi) The statement

pal-if ⟨*condition*⟩ **then** ⟨*statement*⟩;

is used to indicate that all processors satisfying ⟨*condition*⟩ are active and execute ⟨*statement*⟩ in parallel. The ⟨*condition*⟩ refers to a Boolean expression involving the processor index α and the ⟨*statement*⟩ refers to an Algol statement or a sequence of statements enclosed by **pal-begin** and **pal-end**.
(vii) A loop of the form

pal-for $\alpha = n_1$ **to** n_2 **do** ⟨*statement*⟩;

is used to indicate that processors with indices $\alpha = n_1, n_1 + 1, \ldots, n_2$ are active and execute ⟨*statement*⟩ in parallel. The ⟨*statement*⟩ has the same meaning as defined in (vi).

3. A MODIFICATION OF KNUTH'S ALGORITHM FOR BALANCED TREE SEARCH AND INSERTION

Table search and insertion is one of the common problems in information processing. Usually a table of records containing distinct keys is to be searched for a given argument k; if k is not found in the table a record containing k is to be inserted into the table. A number of efficient serial algorithms[26] exist for this purpose. Knuth's Algorithm A[26] for balanced tree search and insertion is one such algorithm. The algorithm is based on a balanced binary tree representation of the table of records. Each node of the tree is assumed to contain four fields, namely the key (KEY) of the node, the pointer (LL) to the left subtree of the node, the pointer (RL) to the right subtree of the node and the balance factor (B) that is equal to the height of the left subtree minus the height of the right subtree of the node. A special header node appears at the top of the tree in location h; the pointer $RL(h)$ points to the root of the tree and $LL(h)$ is used to keep track of the overall height of the tree. The tree is assumed to be non-empty.

In parallel generation of permutations a problem similar to the table search and insertion arises. The problem may be stated as follows:

Problem 1

Given a table of records which form a balanced binary tree and argument k, determine a non-negative integer u and insert into the tree a node containing $k + u$ such that the rank of the inserted node when records are arranged in increasing order of their keys is $u + 1$ and rebalance the tree, if necessary, to make the tree balanced.

In Algorithm 1 we present a serial algorithm named Algorithm BAL-SRH-INT that solves this problem.

Algorithm BAL-SRH-INT is a modification of Knuth's Algorithm A.[26] As in Algorithm A the table of records is represented by a balanced binary tree. Each node is assumed to have six fields: four as defined by Knuth, namely KEY, LL, RL and B and two additional fields, namely, the right information (RI) and the left information (LI) which are equal to the number of nodes in the right subtree and the number of nodes in the left subtree of the node, respectively. Thus a node p of the tree is of the form

$LL(p)$	$KEY(p)$	$RL(p)$
$LI(p)$	$B(p)$	$RI(p)$

The total number of nodes in the subtree $TREE(p)$ with root p is $LI(p) + RI(p) + 1$. The header node h is defined as in Knuth.

The inputs to the algorithm are:

(i) m, the total number of records in the table
(ii) k, the given argument
(iii) $T(1:m)$, the table of records; records are represented in the storage by a balanced binary tree with header h, i.e. for any node p, other than the header h, of the tree we have

(1) $KEY(LL(p)) < KEY(p) < KEY(RL(p))$;

and

(2) $B(p) = -1, 0$ or 1.

As in Algorithm A, the pointer variable p denotes the currently searched node; the auxiliary variable s represents the root of a subtree; the variables t and r always

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

Algorithm 1. Algorithm BAL-SRH-INT

```
              algorithm BAL-SRH-INT (k, m, h, AVAIL);
                begin {inputs: argument k, the number of records m and the table of records T(1:m); the records are
                               represented in the form of a balanced tree with the header node h};
                      {output: updated table T(1:m) after the insertion of a new node with key k + u at the
                               (u + 1)th position, u being an integer determined by the algorithm};

INITIALIZE:           {initialize the pointer variables t, s, p and set the variable u equal to the number of records
                      whose keys are less than the key of the root of the tree}
                      t = h; s = p = RL(h); u = LI(p); Z = 1;

SEARCH:               while Z ≤ log2 m + 1
                        do {top-down search for finding u}

S1:                       if KEY(p) ≤ k + u
                            then {search the right link q and update u}
                              begin q = RL(p); RI(p) = RI(p) + 1; Z = Z + 1;
                                if q ≠ ∧
                                  then {move down the tree}
                                    begin if B(q) ≠ 0 then
                                                          begin t = p; s = q end;
                                          p = q
                                    end;
                                  else {q = ∧ and set q to the address of the new node to be inserted}
                                    begin q ⇐ AVAIL; RL(p) = q end;
                                u = u + LI(p) + 1
                              end {right link search};

S2:                         else {search the left link q and update u}
                              begin q = LL(p); LI(p) = LI(p) + 1; Z = Z + 1;
                                if q ≠ ∧
                                  then {move down the tree}
                                    begin if B(q) ≠ 0 then
                                                          begin t = p; s = q end;
                                          u = u − LI(p) + LI(q); p = q
                                    end;
                                  else {q = ∧ and set q to the address of the new node to be inserted}
                                    begin q ⇐ AVAIL; LL(p) = q end;
                              end {left link search};
                     end-while {top-down search};

INSERT:            begin {insert fields in the new node q}
                          k = k + u; KEY(q) = k; LI(q) = RI(q) = 0;
                          LL(q) = RL(q) = ∧ ; B(q) = 0;

ADJUST:                 if KEY(s) > k then r = p = LL(s); else r = p = RL(s);
                        while p ≠ q do {adjust the balance factors of nodes between s and q}
                          if KEY(p) > k
                            then {adjust the balance factor of p by −1}
                              begin B(p) = −1; p = LL(p) end;
                            else if KEY(p) < k
                                   then {adjust the balance factor of p by 1}
                                     begin B(p) = 1; p = RL(p) end;
                        end-while {balance factors adjust};
                   end {insert};

REORGANIZE:        begin {reorganize the tree to make it balanced}
                     if KEY(s) > k then a = −1; else a = 1;
                     if B(s) ≠ a
                       then {change only the balance factor of s; the new tree is also balanced}
                         if B(s) = −a then B(s) = 0;
                                           else {B(s) = 0 and increase the height of the tree}
                                             begin B(s) = a; LL(h) = LL(h) + 1 end;
                       else {B(s) = a and rebalance the tree}
                         if B(r) = a
                           then {use single rotation}
```

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

Algorithm 1. *continued*

```
                begin p = r; LINK(a, s) = LINK(-a, r);
                      ILINK(a, s) = ILINK(-a, r); LINK(-a, r) = s;
                      ILINK(-a, r) = LI(s) + RI(s) + 1;
                      B(s) = 0; B(r) = 0;
                      if s = RL(t) then RL(t) = p;
                                   else LL(t) = p;
                end {single rotation use};
              else {use double rotation}
                begin p = LINK(-a, r); LINK(-a, r) = LINK(a, p);
                      ILINK(-a, r) = ILINK(a, p); LINK(a, p) = r;
                      ILINK(a, r) = LI(r) + RI(r) + 1;
                      LINK(a, s) = LINK(-a, p);
                      ILINK(a, s) = ILINK(-a, p); LINK(-a, p) = s;
                      ILINK(-a, p) = LI(s) + RI(s) + 1;
                      if B(p) = a
                        then {change the balance factors}
                          begin B(s) = -a; B(r) = 0 end;
                        else B(p) ≠ a
                          begin if B(p) = 0 then B(r) = 0;
                                            else B(r) = a;
                                B(s) = 0
                          end;
                        B(p) = 0;
                        if s = RL(t) then RL(t) = p;
                                     else LL(t) = p;
                end {double rotation use}
          end {the tree reorganize};
        end {algorithm BAL-SRH-INT};
```

point to the father and the son of s, respectively; $LINK(a, p)$ stands for

$$LINK(a,p) = \begin{cases} LL(p) & \text{if } a = -1 \\ RL(p) & \text{if } a = 1 \end{cases}$$

In addition to the notations used in Algorithm A the variable $u = u(x)$ is used to denote the number of nodes whose keys are less than the key of the currently visited node x during the tree search and $ILINK(a, p)$ is defined by

$$ILINK(a,p) = \begin{cases} LI(p) & \text{if } a = -1 \\ RI(p) & \text{if } a = 1 \end{cases}$$

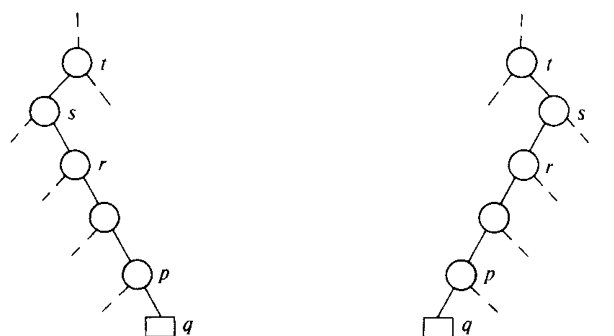
Algorithm BAL-SRH-INT uses a top-down search of the tree (cf. the while loop at label SEARCH) to find an appropriate position $(u + 1)$ for the new node $k + u$ to be inserted, inserts the new node (cf. statements at label INSERT), adjusts the balance factors (cf. statements at label ADJUST) and finally reorganizes the tree, if necessary, to make it balanced (cf. statements at label REORGANIZE). In Algorithm BAL-SRH-INT as in Algorithm A, only one comparison at each level of the tree is made. But Algorithm A compares the key of a node with the argument k whereas Algorithm BAL-SRH-INT compares the key with $k + u$ where u is the number of nodes whose keys are less than the currently compared key, i.e. for each node visited during the search, Algorithm BAL-SRH-INT computes u in addition to the work that it does like Algorithm A. If q denotes the node to be visited and p the node currently visited then

$$u(q) = \begin{cases} u(p) + LI(q) + 1 & \text{if } KEY(p) \le k + u(p) \\ \quad \text{(cf. label S1)} & \\ u(p) - LI(p) + LI(q) & \text{if } KEY(p) > k + u(p) \\ \quad \text{(cf. label S2)} & \end{cases}$$

$$q = \begin{cases} RL(p) & \text{if } KEY(p) \le k + u(p) \text{ (cf. label S1)} \\ LL(p) & \text{if } KEY(p) > k + u(p) \text{ (cf. label S2)} \end{cases} \quad (1)$$

The search is discontinued as soon as a terminal node p, satisfying $LL(p) = RL(p) = \wedge$, is visited and either case (i) $KEY(p) \le k + u(p)$ or case (ii) $KEY(p) > k + u(p)$ occurs.

In case (i) (or case (ii)) there exists a subtree $TREE(s)$ having the maximum height such that either p is the rightmost (or the leftmost) terminal node of $TREE(s)$ or the height of $TREE(s)$ is equal to one and $s = p$. Let t and r be the father and son of s, respectively. The situations are illustrated in Fig. 2.



Case (i) $KEY(p) \le k + u(p)$ Case (ii) $KEY(p) > k + u(p)$

Figure 2

Consider case (i). Using the definition of information fields of a node and the statements at label SEARCH of Algorithm BAL-SRH-INT it is observed that

$$KEY(t) > k + u(t) \tag{2}$$

$$u(s) = u(t) - LI(t) + LI(s) = u(t) - RI(s) - 1$$

and

$$u(r) = u(s) + LI(r) + 1 = u(s) + RI(s) - RI(r) = u(t) - RI(r) - 1 \tag{3}$$

By the method of induction it is easy to show that for any node x lying on the search path from s to p

$$u(x) = u(t) - RI(x) - 1 \tag{4}$$

Since $RI(p) = 0$, $u(p) = u(t) - 1$. So there does not exist a node in the tree whose key lies between $KEY(p)$ and $KEY(t)$.

Let q be the inserted node in the tree. Statements at labels S1 and INSERT of Algorithm BAL-SRH-INT indicate that

$$u = u(q) = u(p) + 1 \quad \text{and} \quad KEY(q) = k + u. \tag{5}$$

Hence from (2) and (5)

$$KEY(p) < KEY(q) < KEY(t).$$

Proceeding in the similar way it can be shown that, in case (ii)

$$u(p) = u(t) + 1 \qquad u = u(q) = u(p), \qquad KEY(q) = k + u$$

and

$$KEY(t) < KEY(q) < KEY(p).$$

Hence we have

Theorem 1

The key $(k + u)$ of the inserted node is unique and its rank is $u + 1$.

Due to the insertion of the node, the new tree may loose its balance property. Following Algorithm A, a single or double rotation technique,[26] depending on the new structure of the tree, is used at label REORGANIZE to make the tree balanced. The balance factors and the information fields of all the affected nodes are adjusted at label SEARCH or ADJUST or REORGANIZE.

Assuming that each operation requires one unit of time, statements at labels INITIALIZE, INSERT, ADJUST, REORGANIZE and at the Zth search of label SEARCH are executed in constant unit of time. Since the while loop at label SEARCH consists of $\lfloor \log_2 m \rfloor + 1$ searches, so

Theorem 2

Algorithm BAL-SRH-INT requires $O(\log_2 m)$ units of time.

Algorithm BAL-SRH-INT is illustrated with an example. Consider a table of 10 records with keys

1 2 4 5 7 8 10 11 13 14

Let the table be stored in the form of the balanced binary tree T of Fig. 3(a); the tree T consists of 4 levels.

In order to find u for the argument $k = 2$ (say), KEY of the node A, i.e. 5, is first compared with $k + u$ where $k = 2$ and u, the number of nodes whose keys are less than 5, is 3. Since $KEY(A)$ is equal to $k + u$, the node to be inserted is in the right subtree of A. So the content of the right information field of A is increased by 1 and the root of the right subtree of A is considered for the next comparison. The procedure is repeated until KEY of a terminal node is compared. The search-path for the argument $k = 2$, and the changed values of information fields, balanced factors and the computed value of u at nodes A, B, C and D are shown in Fig. 3(b). It is found that the value of u at the terminal node D is 4 and $KEY(D) = 7$. Since $KEY(D) > k + u$, a new node E is inserted to the left of D.

From Fig. 3(b) and Ref. 26 it is seen that double rotation is necessary to make the new tree balanced. The balanced tree is in Fig. 3(c).

4. A PARALLEL ALGORITHM FOR GENERATING PERMUTATIONS

Consider the problem of generating all the permutations of r items out of n distinct items in lexicographic order. Without any loss of generality, let the items be represented

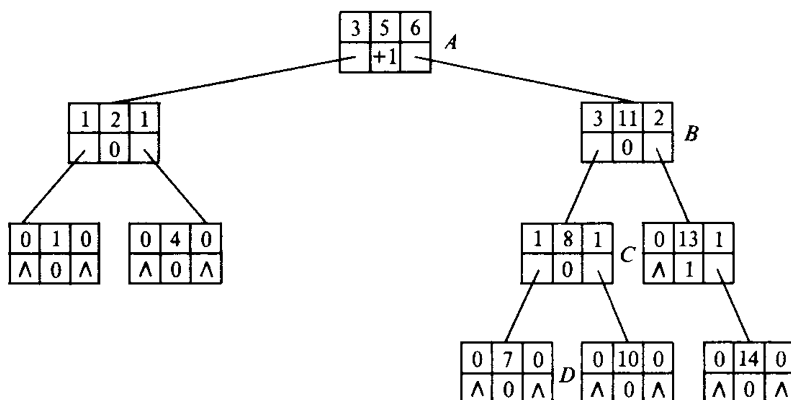


Figure 3(a). Balanced tree T of 10 nodes.

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

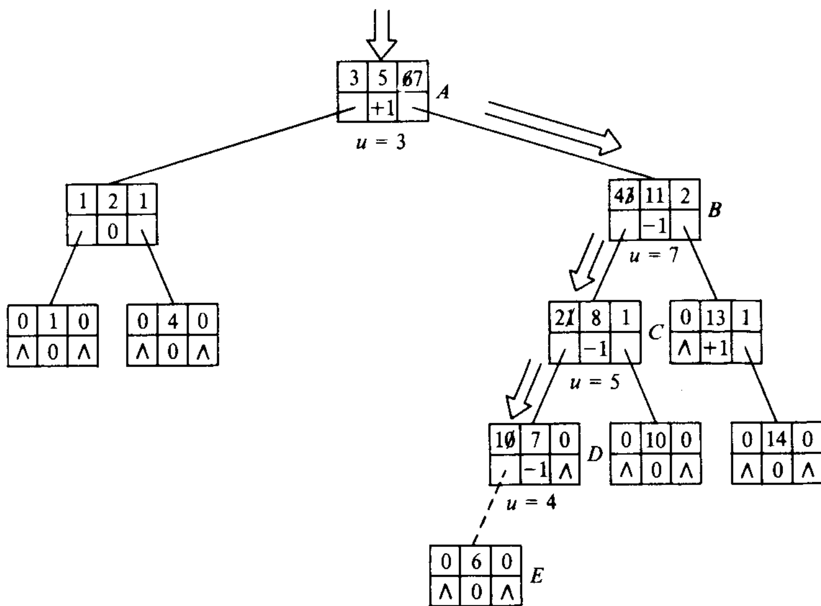


Figure 3(b). Structure of the tree after the insertion of the new node E. The search path is marked by the arrow ⇒.

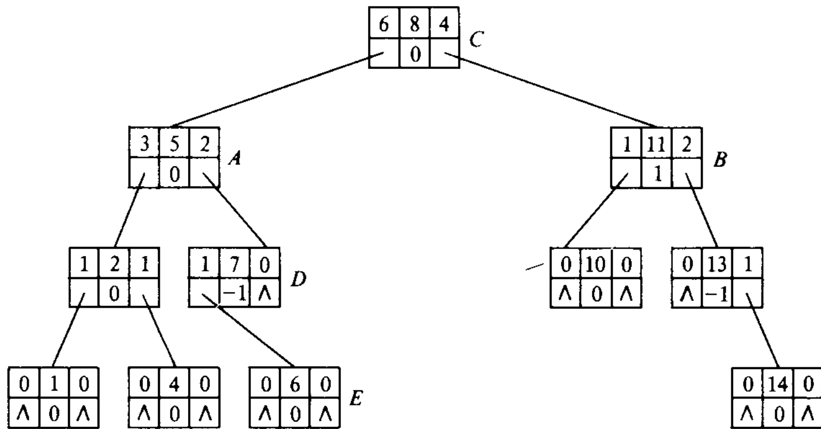


Figure 3(c). New balanced tree.

by the integers 1, 2, . . ., n. The total number of permutations, $\mathbb{N}_0$, satisfies

$$\mathbb{N}_0 = {}^nP_r = n \times {}^{n-1}P_{r-1} \quad (6)$$

A sequential computer requires $O(r)$ multiplications to compute $\mathbb{N}_0$ whereas any model of SIMD computer having $O(r)$ processors can compute it using $O(\log_2 r)$ multiplications.

Let each permutation be represented by a vector of r elements; elements are distinct and belong to the set S of integers 1, 2, . . ., n. If X: $x(i)$, $i = 1, 2, \ldots, r$ and X': $x'(i)$, $i = 1, 2, \ldots, r$ are two permutations, then X precedes X' lexicographically iff for some $j \geq 1$

$$x(i) = x'(i) \text{ for all } i < j \text{ and } x(j) < x'(j) \quad (7)$$

The relation (7) defines a unique ordering among the permutations. Let $X(\mu)$: $x(\mu, i)$, $i = 1, 2, \ldots, r$ denote the μth permutation in lexicographic order, $\mu = 1, 2, \ldots, \mathbb{N}_0$.

Consider an arbitrary permutation, say $X(\gamma)$. Assume that the first m elements of $X(\gamma)$, $0 \leq m < r$, are known. Let A denote the set of m known elements and B denote the set of $(n - m)$ elements of S which are not in A.

There is a group of $\mathbb{N}_m$ permutations including $X(\gamma)$, where

$$\mathbb{N}_m = {}^{n-m}P_{r-m} \quad (8)$$

whose first m elements are equal to the corresponding m elements of $X(\gamma)$. Assume that these $\mathbb{N}_m$ permutations are serially numbered 1 to $\mathbb{N}_m$. Let $\delta(\gamma, m)$ denote the serial number of $X(\gamma)$; note that $\delta(\gamma, 0) = \gamma$. Since there are $(n - m)$ groups of $\mathbb{N}_{m+1}$ permutations each where

$$\mathbb{N}_{m+1} = \mathbb{N}_m/(n - m) \quad (9)$$

and the permutations are in lexicographic order, the kth group, $1 \leq k \leq n - m$, consists of permutations whose serial numbers are $(k - 1)\mathbb{N}_{m+1} + j$, $j = 1, 2, \ldots, \mathbb{N}_{m+1}$. The $(m + 1)$th element of any permutation in the kth group is b_k, the kth smallest element of B. The permutation $X(\gamma)$ belongs to the kth group where

$$k = \delta(\gamma, m)/\mathbb{N}_{m+1} \quad (10)$$

and its serial number within the group satisfies

$$\delta(\gamma, m + 1) = \delta(\gamma, m) - (k - 1)\mathbb{N}_{m+1} \quad (11)$$

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

Hence $x(\gamma, m+1)$ can be found by searching B. However, it can also be found by searching A.

Since S is the set of the first n natural numbers and b_k belongs to S, there are $(b_k - 1)$ elements of S which are less than b_k. Let u be the number of elements in A which are less than b_k. From the definition of b_k, the number of elements in B which are less than b_k is $(k-1)$. But $S = A \cup B$. Hence $b_k - 1 = u + (k-1)$, i.e. $b_k = u + k$ and

Lemma 1

The $(m+1)$th element of $X(\gamma)$ is $k + u$ where u is to be determined in such a way that the rank of $(k+u)$ when inserted into A is $(u+1)$.

Thus, for given k and A, the problem of finding $x(\gamma, m+1)$ is equivalent to Problem 1 and Algorithm BAL-SRH-INT can be used to find $x(\gamma, m+1)$. Procedure 1 is used to find $X(\gamma)$, for an arbitrary γ.

Procedure 1

Given n, r, γ the procedure finds $X(\gamma)$, the γth permutation when the permutations are in lexicographic order.

Step 1. Initialize: $m = 0$, $\mathbb{N}_m = {}^nP_r$, $\delta(\gamma, m) = \gamma$;
Step 2. Repeat Step 3 and Step 4 r times
Step 3. Given the first m distinct elements of $X(\gamma)$ and the integers $\mathbb{N}_m$, $\delta(\gamma, m)$ defined in (8) and (11), respectively, obtain the next elements $x(\gamma, m+1)$ as follows:
 (i) Compute $\mathbb{N}_{m+1}$ and k, defined in (9) and (10), respectively;
 and
 (ii) Find $x(\gamma, m+1)$ using lemma 1;
Step 4. Update $\mathbb{N}_m$ to $\mathbb{N}_{m+1}$, $\delta(\gamma, m)$ to $\delta(\gamma, m+1)$ and m to $m+1$.

Since Procedure 1 generates r elements for any arbitrary γ, it can be used simultaneously (i.e. in parallel) for $\gamma = 1, 2, \ldots, \mathbb{N}_0$ to get all the desired permutations. Step 3 and Step 4 of Procedure 1 are repeated for r times and in each repetition one element of the permutation is generated. It follows that

Lemma 2

Elements of each permutation generated by Procedure 1 are distinct and lie between 1 and n.

Theorem 3

Permutations generated by Procedure 1 are in lexicographic order.

Proof. Theorem 3 is proved if it is shown that for any arbitrary β and γ with $\beta < \gamma$, $X(\beta)$ precedes $X(\gamma)$ lexicographically. Without any loss of generality assume that there exists an integer m, $m \geq 0$ such that the first m elements of $X(\beta)$ and $X(\gamma)$ are equal, i.e.

$$x(\beta, i) = x(\gamma, i), \quad i = 1, 2, \ldots, m, m \neq 0 \quad (12)$$

Let the values of k obtained by (10) be k_1 for $X(\beta)$ and k_2 for $X(\gamma)$. Since $\beta < \gamma$ we have from (11)

```
    algorithm PAL-PRM-GEN (n, r, h, AVAIL, x(1: ⁿP_r, 1: r));
      begin {input: n, r, header h, AVAIL and ℕ₀ to each processor};
            {output: r elements x(1), x(2), ..., x(r) of the ℕ₀ permutations; P processors are assumed to be available};
        pal-for α = 1 to P do
          pal-begin {execute in parallel using P processors}
            γ = α − P;
L1:         for L = 1 step 1 until ⌈ℕ₀/P⌉ do
              begin {modify γ}
L2:             γ = γ + P;
                pal-if γ ≤ ℕ₀
                  then {execute in processor α}
                    pal-begin {initialize}
L3:                   δ = γ; N = ℕ₀;
L4:                   for m = 0 step 1 until r − 1 do
                        begin {generate the (m + 1)th element of X(γ)}
L5:                       N = N/(n − m); k = ⌈δ/N⌉;
                          δ = δ − (k − 1)*N;
L6:                       if m = 0
                            then {form a non-empty tree by inserting a node of RL(h) with key k}
                              begin q ⇐ AVAIL; RL(h) = q;
                                    KEY(q) = k; B(q) = 0; LL(h) = 1;
                                    LL(q) = RL(q) = ∧; LI(q) = RI(q) = 0
                              end {a non-empty tree form};
                            else {m ≠ 0, the tree is non-empty}
                              call Alg. BAL-SRH-INT (k, m, h, AVAIL);
L7:                       x(γ, m + 1) = k
                        end {the (m + 1)th element generate};
                    end-pal {processor α execute};
              end;
          end-pal {parallel execute};
      end {algorithm PAL-PRM-GEN};
```

Algorithm 2. Algorithm PAL-PRM-GEN.

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

$$\delta(\beta, m) < \delta(\gamma, m)$$
$$\Rightarrow \delta(\beta, m)/\mathbb{N}_{m+1} \leq \delta(\gamma, m)/\mathbb{N}_{m+1}$$
$$\Rightarrow k_1 \leq k_2 \Rightarrow b_{k_1} \leq b_{k_2} \Rightarrow x(\beta, m+1) \leq x(\gamma, m+1) \quad (13)$$

Case (i). $x(\beta, m+1) < x(\gamma, m+1)$: It follows from (7) and (12) that $X(\beta)$ precedes $X(\gamma)$ lexicographically.

Case (ii). $x(\beta, m+1) = x(\gamma, m+1)$: So there exists an integer $m+1 \geq 0$ such that (12) is satisfied.

Hence, by induction, it follows that either $X(\beta)$ precedes $X(\gamma)$ lexicographically or $x(\beta, i) = x(\gamma, i)$, $i = 1, 2, \ldots, r$. But for $m = R$ where

$$R = \begin{cases} r-1 & \text{if } r \neq n \\ n-2 & \text{if } r = n \end{cases}$$

$\mathbb{N}_{m+1} = 1$ and from (13) it follows that $x(\beta, m+1) < x(\gamma, m+1)$.

Hence if $\beta < \gamma$, there exists some $j, j > 0$ such that

$$x(\beta, i) = x(\gamma, i) \text{ for all } i < j \text{ and } x(\beta, j) < x(\gamma, j)$$

i.e. $X(\beta)$ precedes $X(\gamma)$ lexicographically. Hence the proof.

Procedure 1 for generation of permutations may be implemented in different models of the SIMD computer. We assume that P processors are available on the model of the SIMD computer described in Section 2. Algorithm 2, namely, PAL-PRM-GEN, has been designed following Procedure 1 to generate the permutations in parallel. In Algorithm PAL-PRM-GEN, one processor is assigned to generate one permutation using Procedure 1, so P permutations are generated at a time by P processors. In order to get all the ${}^nP_r$ distinct permutations, the procedure is required to be repeated $\lceil {}^nP_r/P \rceil$ times. During the Lth repetition of Procedure 1, $L = 1, 2, \ldots, \lceil {}^nP_r/P \rceil$, the processor with index α, $\alpha = 1, 2, \ldots, P$, generates the γth permutation where $\gamma = \alpha + (L-1)P \leq {}^nP_r$. Assume that each operation requires one unit of time. For computing the $(m+1)$th element of the γth permutation, $m < r$, Step 3 of Procedure 1 uses Algorithm BAL-SRH-INT described in the previous section which requires $O(\log_2 m)$ units of time while a constant unit of time is required for Steps 1, 2 and 4. Since each processor uses $O(m)$ spaces of its local memory to keep a table of m records which form a balanced tree and Steps 2, 3 and 4 of Procedure 1 are repeated for r times and Algorithm PAL-PRM-GEN repeats the procedure $\lceil {}^nP_r/P \rceil$ times to generate ${}^nP_r$ permutations, so it follows that

Theorem 4

Using P processors and $O(r)$ space in the local memory of each processor, Algorithm PAL-PRM-GEN generates all the ${}^nP_r$ distinct permutations of r items out of n distinct items in $O(\lceil {}^nP_r/P \rceil \log_2 (r-1)!) \simeq O(\lceil {}^nP_r/P \rceil r \log_2 r)$ units of time.

The binary search and deletion technique[26] could also have been used to find $x(\gamma, m+1)$, the $(m+1)$th element when the first m elements of the permutation $X(\gamma)$ have been generated. The reason for not using the technique is that it increases the time complexity of the algorithm to $O(\lceil {}^nP_r/P \rceil r \log_2 n)$ units of time.

Algorithm PAL-PRM-GEN is illustrated with an example. Let there be 4 items and the ith item be represented by i, $i = 1, 2, 3, 4$. The number of permutations of 3 items out of 4 items is $\mathbb{N}_0 = ({}^4P_3) = 24$. Let 15 processors be available. Table 1 illustrates the generation of permutations by the processor with index 2. The processor with index 2 generates $X(2) = 1, 2, 4$, and $X(17) = 3, 4, 1$.

Table 1. Generation of permutations by processor with index 2

	L	γ	δ	m	N	k	δ	Tree	$x(\gamma, m+1)$
Ref. Label No.	L1	L2	L3	L4	L5	L5	L5	L6	L7
	1	2	2	0	6	1	2	h; [0 1 0 / ∧ 0 ∧]	1
				1	2	1	2	h; [0 1 1 / ∧ 1]; [0 2 0 / ∧ 0 ∧]	2
				2	1	2	1	h; [1 2 1 / 0]; [0 1 0 / ∧ 0 ∧], [0 4 0 / ∧ 0 ∧]	4
	2	17	17	0	6	3	5	h; [0 3 0 / ∧ 0 ∧]	3
				1	2	3	1	h; [0 3 1 / ∧ 1]; [0 4 0 / ∧ 0 ∧]	4
				2	1	1	1	h; [1 3 1 / 0]; [0 1 0 / ∧ 0 ∧], [0 4 0 / ∧ 0 ∧]	1

Acknowledgement

Authors are thankful to the referee for his valuable suggestions and comments on an earlier version of the paper. The first author wishes to acknowledge the financial support of the Department of Atomic Energy, Government of India.

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022

REFERENCES

1. W. L. Miranker, A survey of parallelism in numerical analysis. *SIAM Review* **13**, 524 (1971).
2. G. H. Barnes, *et al.*, The ILLIAC IV computer. *IEEE Trans. Computers* **C-17**, 776 (1968).
3. R. J. Swan, *et al.*, Cm* a moduler multi-microprocessor. *Proc. AFIPS* **46**, 637 (1977).
4. R. G. Hentz and D. P. Tate, Control Data STAR-100 processor design. *Compcon 72, IEEE Comput. Soc. Conf. Proc.*, p. 1 (1972).
5. H. S. Stone, Parallel computers. In *Introduction to Computer Architecture*, edited by H. S. Stone, Science Research Associates, Chicago, p. 318 (1975).
6. H. T. Kung, Synchronized and asynchronous parallel algorithms for multiprocessors. In *Algorithms and Complexity: New Directions and Recent Results*, edited by J. F. Traub, Academic Press, N.Y., p. 153 (1976).
7. D. Heller, A survey of parallel algorithms in numerical linear algebra. *SIAM Review* **20**, 740 (1978).
8. D. Nassimi and Sartaj Sahni, Parallel permutation and sorting algorithms and a new generalized connection network. University of Minnesota, TR 79-8, (1979).
9. D. S. Hirschberg, Fast parallel sorting algorithms. *CACM* **21**, 657 (1978).
10. F. P. Preparata, New parallel-sorting schemes. *IEEE Trans. Computers* **C-27**, 669 (1978).
11. C. D. Thompson and H. T. Kung, Sorting on a mesh-connected parallel computer. *CACM* **20**, 263 (1977).
12. D. Nassimi and S. Sahni, Bitonic sort on a mesh-connected parallel computer. *IEEE Trans Computers* **C-28**, 2 (1979).
13. C. Savage and Joseph Jaja, Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput.* **10**, 682 (1981).
14. R. P. Brent, The parallel evaluation of general arithmetic expression. *JACM* **21**, 201 (1974).
15. I. Munro and M. Paterson, Optimal algorithms for parallel polynomial evaluation. *JCSS* **7**, 189 (1973).
16. K. Maruyama, On the Parallel Evaluation of Polynomials. *IEEE Trans. Computers* **C-22**, 22 (1973).
17. C. B. Tompkins, Machine attacks on problems whose variables are permutations. *Numerical Analysis*, American Mathematical Society, 198 (1956).
18. M. B. Wells, Generation of permutations by transposition. *Math. Comp.* **15**, 192 (1961).
19. S. M. Johnson, Generation of permutations by adjacent transpositions. *Math. Comp.* **17**, 282 (1963).
20. H. F. Trotter, Algorithm 115, *CACM* **5**, 434 (1962).
21. J. Boothroyd, Algorithm 29, Permutation of the elements of a vector. *Comp. J.* **10**, 311 (1967).
22. R. J. Ord-Smith, Algorithm 323: Generation of permutations in lexicographic order. *CACM* **11**, 117 (1968).
23. R. J. Ord-Smith, Generation of permutation sequences. *Comp. J.* **13**, 152 (1970).
24. J. P. N. Phillips, Algorithm 28: permutations of the elements of a vector in lexicographic order. *Comp. J.* **10**, 311 (1967).
25. G. Ehrlich, Loopless algorithms for generating permutations, combinations and other combinatorial configurations. *JACM* **20**, 500 (1973).
26. D. E. Knuth, *The Art of Computer Programming Vol. 3*. Addison-Wesley, Reading, Mass. (1973).
27. H. J. Seigel, A model of SIMD machines and a comparison of various interconnection networks. *IEEE Trans. Computers* **C-28**, 907 (1979).

Received October 1981

Downloaded from https://academic.oup.com/comjnl/article/26/2/97/309905 by U.S. Department of Justice user on 16 August 2022