整数集合の順列と組み合わせを生成する
並列アルゴリズムについて

NOTE ON A PARALLEL ALGORITHM FOR GENERATING
PERMUTATIONS AND COMBINATIONS

伊豆田義人・太田耕三・小嶋隆一・小針亮子
G. Izuta, K. Ohta, R. Kojima and R. Kobari

ABSTRACT

This paper suggests a recursive parallel algorithm that produces the permutations and combinations of a set of integers by using a table of appropriate size. The procedure for generating permutations that runs on the PRAM computer is optimal and adaptive. Furthermore it is a by product of the procedure for MIMD machine. The procedure for generating combinations is derived from the permutations one with minor changes, which is possible due to the use of the table and recursion. Finally, the key feature is that the algorithm presented here is concerned with the problem of an algorithm generating both the permutations and combinations.

I. INTRODUCTION

The study of the computational methods for generating permutations and combinations has a long history [1] and is currently almost a standard subject in the field of algorithms theory. There is a vast literature handling these problems in the sequential case. Among them, the use of the combination generator and the numbering system proposed by Mifsud [2] and Knott [3, 4], respectively are quite known. On the other hand, it has been pointed out that reports on researches related to the parallel algorithms for these problems are not as numerous as their sequential counterpart [5].

In the parallel setting, Akl [6] proposed an algorithm on the grounds of the numbering system, which is optimal but can not be used to generate both the permutations and combinations; Gupta and Bhattacharjee [7] suggested a non-optimal algorithm for PRAM computer; Mor and Frankel [8] designed a strongly machine architecture dependent algorithm for vector computing type; and Lawrie [9], Lenfant [10], Nassimi and Sahni [11] and others researched algorithms for special purpose architectures, which either are based on a hard wired interconnection or produce only a subset of the permutations or are not optimal. On the basis of the discussions above, this paper deals with the generation of permutations and combinations in parallel and aims at:

(1) Presenting a simple parallel algorithm for generating permutations and combinations in lexicographic order. The algorithm is on the basis of what is named here in
core generation due to the fact that the m-tuples are output in a matrix and their values are to be known only at the end of the processing.

(2) Spotting the problem of generating permutations and combinations in parallel with respect to cost optimality and adaptability by focusing on the MIMD and PRAM models of computation. Note that the cost is evaluated by comparing the parallel algorithm with an optimal sequential one.

(3) Giving some insights into the problem of making the m-permutation generation algorithm to generate efficiently m-combinations by means of the intensive use of recursive procedures.

The paper is organized as follows. In section II, the basic concepts and facts are reviewed; in section III, the generation of the permutations is focused on, and the general idea of the algorithm is outlined. Also, the procedures for MIMD and PRAM computers are presented. In section IV, it is dealt with the generation of the combinations following the same lines of reasoning of the previous section. In section V, concluding remarks are given.

II. BRIEF REVIEW OF SOME BASIC CONCEPTS AND FACTS

In this section a brief review of the basic concepts and notations used in this paper are presented.

Assumption 1: Let the set $S$ of items to be handled with consists of the first $N$ integers, i.e., $S = \{1, 2, ..., N\}$.

Definition 1: An m-permutation of $S$ is obtained by selecting $M$ distinct integers out of $N$ and arranging them in some order.

Fact 1: The number of distinct m-permutations of $N$ items is denoted by $P_{nm}$, where

$$P_{nm} = \frac{n!}{(n-m)!}.$$  

(1)

Definition 2: Let $x = (x_1, x_2, ..., x_m)$ and $y = (y_1, y_2, ..., y_m)$ be two m-permutation of $S$. Then, $x$ precedes $y$ in lexicographic order if there exists an index $i$, for $1 \leq i \leq m$, such that $x_i = y_i$ for all $j < i$ and $x_i < y_i$.

Definition 3: The overall running time of an optimal sequential algorithm for generating permutations in lexicographic order is given by $\Theta(P_{nm} \times m)$.

Definition 4: An m-combination of $S$ is an m-permutation, in which two different m-combinations differ with respect to the items they contain.

Fact 2: The total number of combinations is denoted by $C_{nm}$ and is determined by

$$C_{nm} = \frac{n!}{(n-m)! \cdot m!}.$$  

(2)

III. PERMUTATIONS AND FACTS

In this section only the generation of m-permutations is the subject of study. The relationship between these procedures with that of generating combinations one will be studied in the next section.
III.1 In Core Generation Algorithm

An Algorithm for generating in parallel the m-permutations of a set S is presented in this subsection. The result appears at the end of the operations in the form of a \( P_{nm} \times m \)-dimensional matrix. Each row of the matrix corresponds to a permutation and the rows are in lexicographic order from top to bottom. In what follows, the general philosophy of the algorithm is given and then procedures for MIMD and PRAM models of computation are yielded. For this purpose the next definition will be useful.

**Definition 5:** A set of \( P_{nm}/(n(n-1)\ldots(n-j)) \) rows from the \([i \cdot P_{nm}/n(n-1)\ldots(n-j)] + 1\) \(^{th}\) to 
\([(i+1) \cdot P_{nm}/n(n-1)\ldots(n-j)]^{th}\) row is called block \( j \), where \( 0 < j < m \) and \( 0 < i < (n-1) \). And when \( j = 0 \) and no ambiguities arise the term block is used in spite of block 0.

III.2 Philosophy of the Algorithm

Assume that \( P_{nm} \times m \) positions of memory are assigned to the generation of \( P_{nm} \) m-permutations is assigned. Clearly, this memory space forms a \( P_{nm} \times m \) matrix. A typical configuration after the computations is shown in figure 1. Note that there are \( N \) blocks with the integer \( i \) poked in the memory position \( i^{th} \) block, 1). Here, * means an appropriate element. The process of how the matrix is filled up is as follows.

**Step 1:** Let the sequence S be expressed by \( (x_1, x_2, \ldots, x_n) \).

**Step 2:** Put the integer \( xi \) in memory position \( j^{th} \) block, \( j+1 \), for all \( 0 < j < m-1 \), and make sure that once written this integer will not be used again in the same row. For \( j=m-1 \); namely, column m, write the \( n \cdot m \) remaining elements of the sequence S in increasing order, taking care to increment the row number before writing the next element. At this point, as shown in figure 1, the m-permutations \( (x_1, \ldots, x_{m-1}, x_m), (x_1, \ldots, x_{m-1}, x_{m+1}), \ldots, (x_1, \ldots, x_{m-1}, x_n) \) are generated.

**Step 3:** Go back to column \((m-1)\). Let the beginning of the next writing operation be the next block on that column. Write on this block the next least integer greater than the one used in the previous block. Handle the column m as in step 2. Thus at the end of this step, the m-permutations \( (x_1, \ldots, x_{m-2}, x_{m+1}, x_m), (x_1, \ldots, x_{m-2}, x_{m+2}, x_{m+1}), (x_1, \ldots, x_{m-2}, x_{m+2}, x_{m+3}), \ldots, (x_1, \ldots, x_{m-2}, x_{m+2}, x_n) \) are obtained.

**Step 4:** Repeat step 3 for all the remaining elements. In other words, interchange \( x_{m+1} \) with \( x_{m+2} \) and then \( x_{m+2} \) with \( x_{m+3} \), and so forth until the interchange of \( x_{m+1} \) with \( x_n \) is carried out. This operation leads to the m-permutations \( (x_1, \ldots, x_{m-2}, x_{m+2}, x_m), (x_1, \ldots, x_{m-2}, x_{m+3}, x_m), (x_1, \ldots, x_{m-2}, x_{m+3}, x_{m+1}), \ldots, (x_1, \ldots, x_{m-2}, x_{m+3}, x_{m+2}, x_n), \ldots, (x_1, \ldots, x_{m-2}, x_{m+3}, x_n, x_m), (x_1, \ldots, x_{m-2}, x_{m+3}, x_n, x_{m+1}), \ldots, (x_1, \ldots, x_{m-2}, x_{m+3}, x_n, x_{m+2}, x_{n+1}) \).

**Step 5:** Back to column \((m-2)\). Fill up the next block with the least element greater than the used in the previous block. Consider the element of S not used in the blocks to the left as unhandled. Make the block of column \((m-1)\) with the least element unhandled yet. Work on the column m as in step 2. Execute the step 3. Repeat this step while possible, i.e., making blocks with elements greater that used in the previous operation.

**Step 6:** Back to \( j^{th} \) column for \( (m-3) \geq j \geq 0 \) in decreasing order. For each \( j \), one block is made at each time the blocks to the right are built up in a recursive fashion.
by using the previous step. Thus, if \( j = m - 3 \) and a block is to be constituted, then after writing the correct element, the blocks of the next columns are yielded by the procedure of step 5.

![Figure 1. Memory space and its contents](image-url)
III.3 MIMD Algorithm

Consider a CREW SM MIMD computer with N processors, where N \leq n. Thus the algorithm above for this machine creates for each element of the sequence n process for column 1 and n(n-1)...(n-j) processes for each column j, where 2 \leq j \leq m. These processes are executed by the N processors asynchronously. Figure 2 shows the processes tree for one element of S, where the root process is created by the main procedure and the leaves are triggered by the upper level processes.

Figure 2. Tree of processes

The algorithm is given below as a procedure. The array, in which the m-permutations are to be stored, is called PERM-mem. The symbols ← and → stand for assignment and interchange, respectively. The inner part of ‘( )’ means comments, and the numbers between ( ) are the step numbers.

procedure MIMD INCORE(n,m)

(This procedure uses one variable, which is called StartWriteAt, to provide the initial position of the writing operations and one variable, namely S1, that holds the integer to be written and an array S2end with elements of S.)

Step 1: (1.1) S1 1
(1.2) S2end 2, 3, ..., N

Step 2: (Create a process for the first block)
(2.1) Calculate the value of StartWriteAt
(2.2) INCORE-FILL(S1, S2end, 1, StartWriteAt)
Step 3: (Tackle the 2\textsuperscript{nd} to N\textsuperscript{th} blocks.)
for BlkN=2 to N do
  (3.1) S1S2end[BlkN]
  (3.2) calculate the value of StartWriteAt
  (3.3) INCORE-FILL(S1, S2end, 1, StartWriteAt)
end for

process INCORE-FILL(S1, S2end, ColToFill, StartWriteAt)
{The process uses the variables passed by its creator and uses as local variables newS1 and newS2end, which are used to create sub processes and play the same role as S1 and S2end.}
if ColToFill = m then
  (1) PERM-mem[pos][m] ← S1
else
  (write on memory position (block \(j, j+1\), \(j < M-1\).
for \(i=1\) to \(Pn,m/n...(n-ColToFill)\) do
  (2.1) PERM-mem[pos][ColToFill] ← S1
end for.
{Begin to consider the next column.}
  (2.2) newS1 ← S[n1]
  (2.3) newS2end ← S-newS1
  (2.4) INCORE-FILL(newS1, newS2end, ColToFill+1, StartWriteAt)
{Poke the remaining elements of newS2end for the same column.}
for \(i=1\) to size of newS2end end do
  (2.5) newS1 ← newS2end[i]
  (2.6) update the value of StartWriteAt.
  (2.7) INCORE-FILL(newS1, newS2end, ColToFill+1, StartWriteAt)
end for.
end if

Analysis of the algorithm
The procedure above is such that
(a) The total number of processes denoted by TotP is given by

\[
Totp = \sum_{i=1}^{m} n(n-1)...(n-i+1) = \sum_{i=1}^{m} \frac{(n-m)!}{(n-i)!}\cdot
\]

Thus the total number of processes is expressed by

\[
Totp = O(P_{num}).
\]

Since there are \(N\) processors, the number of processes executed theoretically by each processor is given by

\[
\frac{Process}{processor} = \frac{TotP}{\text{no. of processors}} = O\left(\frac{P_{num}}{N}\right).
\]
Since the processes are asynchronous and the time required to execute the processes are not constant, this value hold in practice. An interesting interpretation for equation (4) is that since there are $P_{nm}$ rows in the array, each process produces in average one row.

(b) The memory space utilized by the algorithm is basically equal to the size of the matrix of m-permutations, which is expressed by

$$\text{memory space} = O(mxP_{nm}).$$  \quad (6)

(c) Since the analysis of the running time and cost of asynchronous algorithms is quite difficult and in general leads to unreliable results when compared with that yielded by an actual implementation, this kind of analysis is not accomplished here.

(d) It was assumed CREW SM model, i.e., the adoption of the concurrent read policy, which is useful only in broadcasting the value of the size of the sequence and the value of the value of m. This restriction can be singled out to allow EREW if the n and m are spread to all processes as variables at the time of the process creation.

(e) The permutations are not produced in sequence from top to bottom due to the asynchronous nature of the processes. To overcome this kind of abnormality a scheduling scheme for the procedures, which is not an issue in this paper, should be taken into consideration.

From these comments, it is straightforward that the following assertion holds.

**Proposition 1:** Let $S$ be a set of integers with $n$ elements. Consider CREW SM MIMD model of computation with $N$ ($N \leq n$) processors and the procedure MIMD INCORE for generating m-permutations. Then the number of processes executed by each processor and the memory space required to execute the algorithm are given by equations (5) and (6) respectively.

### III.4 SIMD Version

Consider an EREW SM SIMD computer with $N$ processors, with each processor in charge of the generation of $\lceil n/N \rceil$ blocks. Thus, to implement the algorithm given in section III.3 on an SIMD computer, let the procedure MIMD INCORE be called procedure SIMD INCORE, and the let the process INCORE-FILL be called procedure INCORE-FILL as below. To let all the processors know the values of $n$ and $m$ either let them be variables passed as parameters of the procedure or apply any algorithm available for broadcasting efficiently these values before starting the generation of the m-permutations.

**procedure SIMD INCORE(n,m)**

**Step 1:**

1.1) $S_{11} = (i\cdot 1) \times \lceil n/N \rceil + 1$

1.2) $S_{2\cdot \text{end}}, S_{1i}, \quad 1 \leq i \leq N$
Step 2:
for i=1 to N do in parallel
    (2.1) calculate the value of StartWriteAti
    (2.2) INCORE-FILL(S1i, S2endi, ColToFill, StartWriteAti)
end for

Step 3:
for i=1 to N do in parallel
    for BlkN=2 to ⌊ceil n/M⌋ do
        (3.1) S1i=S2endi[BlkN]
        (3.2) calculate the value of StartWriteAti.
        (3.3) INCORE-FILL(S1i, S2endi, ColToFill, StartWriteAti)
    end for
end for

Analysis of the algorithm
The analysis related to the total number of procedure calls and memory space required by the algorithm is similar to the previous section ones. For this, here the word procedure is used in place of process. The analysis of the running time is as follows. For the sake of simplicity, only one block is considered first.
(a) Procedure SIMD INCORE: Steps 1, (2.1), (3.1) and (3.2) take constant time.
b) Procedure SIMD-INCORE: Steps (2.2), (2.3), (2.5) and (2.6) take constant time.
c) For the recursion, the running time of the operation on the i-th column (2≤i≤m-1) is given by steps (2.1) and (2.7). Step (2.1) is executed Pnm/n(n-1)...(n-i) times. Taking into account the iteration of step (2.7), the running time reads O(Pnm/n).
d) For i = m, the running time is given by step 1 and (2.7). As in the previous case, the running time is O(Pnm/n). Thus, since there are m columns, the overall running time T_pb for one block is
\[ T_{pb} = O \left( \frac{mxP_{nm}}{n} \right). \]  
and as long as each processor generates ⌊n/N⌋ blocks, the overall running time T_p of the algorithm is given by
\[ T_p = O \left( \frac{mxP_{nm}}{n} \times \frac{n}{N} \right) = O \left( \frac{mxP_{nm}}{n} \right). \]  
Also as far as the cost is concerned, the cost C_p of the parallel algorithm is given by
\[ C_p = T_{pxN} = O \left( mxP_{nm} \right). \]  
Now, recall that in the optimal sequential case, the running time is given by
\[ T = \Theta \left( mxP_{nm} \right). \]  
Hence the parallel algorithm is optimal cost with maximum speed up. Since the number of processors available is N≤n, the algorithm is adaptive with respect to the number of processors. Also, from equation (8) it is clear that T_p varies inversely with the number of processors, within the bounds for N, which means that the algorithm is
time adaptive. In summary

*Proposition 2*: Let S be a set of integers with n elements. Consider CREW SM MIMD model of computation with N (N \leq n) processors and the procedure MIMD INCORE for generating m-permutations. Then procedure MIMD INCORE can be used to produce an algorithm for EREW SM SIMD computer with N processors. Moreover the derived algorithm is of optimal cost with maximum speed up, and adaptive with respect to the number of processors.

**Remarks**

(1) Nevertheless the algorithm is optimal cost it may happen that a processor be idle for some time. This occurs when n is not a multiple of N. For example, for N=3, n=7, the processor number 1 is in charge of the generation of the 1\textsuperscript{st}, 2\textsuperscript{nd} and 3\textsuperscript{rd} blocks; the processor number 2 of 4\textsuperscript{th}, 5\textsuperscript{th} and 6\textsuperscript{th} blocks while the processor number 3 generates only the 7\textsuperscript{th} block.

(2) At the beginning of the algorithm the processors must calculate the values of \(P_{num}\) which takes \(O(m)\) time. This term is excluded from the analysis due to the fact that it is insignificant when compared to that determined above.

**IV. COMBINATIONS**

As mentioned previously, it has been pointed out that the generation of combinations only by using algorithms for permutations is an issue in the parallel algorithm design. In this section, it is shown how the previous algorithm can generate the combinations with a few changes for an EREW SM MIMD computer. In fact, this is possible due to the similarities in the memory configurations after the generation of combinations and permutations as can be seen by comparing figure 1 with a sketch for combinations shown in figure 3. Note that in figure 3, the terms between parentheses mean the number of rows.

The algorithm builds the table in the following way.

Step 1: The main procedure produces the processes that generate the leftmost column.

Step 2: The processes thus created fill up the 1\textsuperscript{st} column and trigger sub processes to write on the right column. These sub processes do their processing and again trigger new sub processes to store new elements in the next column. This continues until the last column is yielded.
<table>
<thead>
<tr>
<th>Column</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>\ldots</th>
<th>\text{n-column}</th>
<th>\text{n-column+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{1n}</td>
<td>\text{C_{1-1,m-1}}</td>
<td>\text{C_{1-2,m-2}}</td>
<td>(C_{1-3,m-3})</td>
<td>\ldots</td>
<td>(C_{1-column,1})</td>
<td>\ldots</td>
</tr>
<tr>
<td>\text{n-2n}</td>
<td>\text{C_{n-2,m-2}}</td>
<td>2</td>
<td>(C_{n-3,m-3})</td>
<td>\ldots</td>
<td>\text{n-1}</td>
<td>\text{n}</td>
</tr>
<tr>
<td>\text{n-3n}</td>
<td>3</td>
<td>2</td>
<td>(C_{n-3,m-3})</td>
<td>\ldots</td>
<td>\text{n-1}</td>
<td>\text{n}</td>
</tr>
<tr>
<td>\text{n-4n}</td>
<td>(C_{n-4,m-3})</td>
<td>3</td>
<td>(C_{n-4,m-3})</td>
<td>\ldots</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
<tr>
<td>\text{n-5n}</td>
<td>(C_{n-5,m-4})</td>
<td>4</td>
<td>(C_{n-5,m-4})</td>
<td>\ldots</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
<tr>
<td>\text{n-6n}</td>
<td>(C_{n-6,m-5})</td>
<td>5</td>
<td>(C_{n-6,m-5})</td>
<td>\ldots</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
<tr>
<td>\text{n-7n}</td>
<td>(C_{n-7,m-6})</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
<tr>
<td>\text{n-8n}</td>
<td>(C_{n-8,m-7})</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
<tr>
<td>\text{n-9n}</td>
<td>(C_{n-9,m-8})</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
<tr>
<td>\text{n-10n}</td>
<td>(C_{n-10,m-9})</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
<td>\text{...}</td>
</tr>
</tbody>
</table>

Figure 3. Generation of the combinations

Hence, the procedure PERM-MIMD can be used with a few changes as shown below.

**procedure MIMD COMB(n,m)**

**Step 1:**

(1.1) S1 = 1
(1.2) $S_{2\text{end}} \leftarrow 2, 3, \ldots, n$

Step 2:
(2.1) calculate the value of $StartWriteAt$
(2.2) COMB-FILL($S_1$, $S_{2\text{end}}$, 1, $StartWriteAt$)

Step 3: for BlkN=2 to (n-m)+1 do
(3.1) $S_1 \leftrightarrow S_{2\text{end}}[1]$
(3.2) $S_{2\text{end}} \leftarrow S_{2\text{end}}\cdot S_1$
(3.3) COMB-FILL($S_1$, $S_{2\text{end}}$, 1, $StartWriteAt$)

process COMB-FILL($S_1$, $S_{2\text{end}}$, ColToFill, $StartWriteAt$)
if ColToFill = m then
    (1) COMB-mem[pos][m] $\leftarrow$ S1
else
    [write on memory, but first calculate the number of times to do it.]
    NumberOfWriteValue calculated
    for i=1 to NumberOfWrite do
        (2.1) COMB-mem[pos+1][ColToFill] $\leftarrow$ S1
    end for
    (2.2) $S_1 \leftarrow S[1]$
    (2.3) $S_{2\text{end}} \leftarrow S\cdot S_1$
    (2.4) COMB-FILL($S_1$, $S_{2\text{end}}$, ColToFill+1, $StartWriteAt$)
    for i=1 to SizeOfS - m+1 do
        (2.5) $S_1 S_{2\text{end}}[1]$
        (2.5) $S_{2\text{end}} S_{2\text{end}} \cdot S_1$
        (2.6) update the value of $StartWriteAt$
        (2.7) COMB-FILL($S_1$, $S_{2\text{end}}$, ColToFill+1, $StartWriteAt$)
    end for
end if

Analysis of the algorithm
The procedure for generating combinations thus obtained is such that
(a) It is structurally similar to the permutations one, but differs in the number of
writing operations it performs.
(b) The definition of block is meaningless here, since on the same column there is no
uniformity in the size of each piece.
(c) the memory space required is obviously

\[ \text{memory space} = O(mx_{\text{cm}}). \]  \hspace{1cm} (1)

(d) An SIMD version would be quite difficult, since it is not just a matter of letting the
processes be procedures. The problem is that \( \lfloor n/N \rfloor \) blocks per processor no
longer is a good policy.
From these, the following theorem is yielded.
Theorem 1: Let $S$ be a set of integers with $n$ elements. Consider EREW SM MIMD model of computation with $N$ ($N \leq n$) processors and the procedure MIMD INCORE for generating $m$-permutations. Then, basically, the same algorithm can be used to generate $m$-combinations of the set $S$.

V. CONCLUDING REMARKS

In this paper a recursive procedure for generating both the $m$-permutations and $m$-combinations of a set $S$ in parallel were suggested on the basis of what was named here in core philosophy. Procedures for generating permutations on the MIMD and PRAM computers were presented, and it was shown that the procedure for the PRAM computer is both adaptive and optimal. Finally, a procedure for generating $m$-combinations was obtained from the algorithm for generating $m$-permutations with a few modifications, which is a challenge as far as this kind of computational problem is an issue.

VI. REFERENCES