BIT 30 (1990), 424–436

AN OPTIMAL ALGORITHM FOR GENERATING EQUIVALENCE RELATIONS ON A LINEAR ARRAY OF PROCESSORS

IVAN STOJMENOVIĆ[1]

Computer Science Department, University of Ottawa, Ottawa, Ontario, Canada K1N 9B4, and Institute of Mathematics, University of Novi Sad, Yugoslavia

Abstract.

We describe a cost-optimal parallel algorithm for enumerating all partitions (equivalence relations) of the set $\{1,\ldots,n\}$, in lexicographic order. The algorithm is designed to be executed on a linear array of processors. It uses n processors, each having constant size memory and each being responsible for producing one element of a given set partition. Set partitions are generated with constant delay leading to an $O(B_n)$ time solution, where B_n is the total number of set partitions. The same method can be generalized to enumerate some other combinatorial objects such as variations. The algorithm can be made adaptive, i.e. to run on any prespecified number of processors. To illustrate the model of parallel computation, a simple case of enumerating subsets of the set $\{1,\ldots,n\}$ having at most m $(\leq n)$ elements is also described.

CR categories: C.1, F.2.

Keywords and phrases: parallel algorithm, linear array, subsets, equivalence relations.

1. Introduction.

Recently a number of parallel algorithms for generating combinatorial objects have been proposed. The types of objects studied are permutations, combinations, subsets, set partitions (i.e. equivalence relations), derangements, etc. In order to characterize known approaches, we list some desirable properties of generation techniques:

PROPERTY 1: The objects are listed in the lexicographic order.

PROPERTY 2: The algorithm is cost-optimal (i.e. the number of processors it uses multiplied by its running time matches – up to a constant factor – a lower bound on the number of operations required to solve the problem).

[1] The research is partialy supported by NSERC operating grant OGPIN 007.
Received October 1989. Revised February and March 1990.

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

a) the time to generate objects without need to produce them is counted. Optimal sequential algorithms in this sense generate objects in time proportional to the number of objects to generate.
b) the time to actually output the objects is counted. Here, optimal sequential algorithms run in time proportional to the total size of output, i.e. the product of the number of objects to generate and the size of one combinatorial object. In this paper we adopt this measure.

The latter approach can be further classified according to the i/o architecture. We assume that each processor is connected to a distinct output port. This approach is advantageous in case when the distributed representation of the objects can be used directly as input to another parallel program. A serial algorithm running in one of the processors would need $O(n)$ time to distribute each object over the n processors, whereas the parallel algorithm produces the distributed object in constant time.

Alternatively, one could consider a serial computer with a number of parallel output ports, each connected to a distinct address in main memory, and an instruction that outputs the contents of these memory cells in parallel. Conversely, one could well imagine a parallel computer where the output takes place over a serial port. This machine would require an output time proportional to the length of the object, despite the parallel computational capability.

PROPERTY 3: The time required by the algorithm between any two consecutive objects it produces is constant. A constant time delay between outputs is particularly important in systolic applications [1].

PROPERTY 4: The model of parallel computation should be as simple as possible.. In order of simplicity, frequently used models are:

k independent processors (no interconnection network links the processors except a connection to a master processor which distributes the job);

linear array of m processors, indexed 1 through m, where each processor i ($1 \leq i \leq m$) is connected by bidirectional links to its immediate left and right neighbors ($i - 1$ and $i + 1$) if they exist; this model is practical, as it is amenable to VLSI implementation [1];

SM EREW PRAM (shared memory exclusive read exclusive write parallel random access machine); in this model, a number of processors share a common memory and execute the same instruction of an algorithm on different data synchronously; however, no two processors are allowed to read from or write into the same memory location simultaneously. This model is mainly of theoretical interest.

PROPERTY 5: Each processor neeeds as little memory as possible, preferably a constant number of words, each of $\log n$ bits capable of storing an integer no larger

than n (where n is the size of one object). This implies that no processor can store an array of size n, or a counter like $n!$.

PROPERTY 6: The algorithm should produce *all* objects of given type.

PROPERTY 7: The algorithm is *adaptive* (an algorithm is adaptive if it can run on any number of processors).

We now review some existing parallel algorithms for generating certain types of combinatorial objects and provide a brief assessment of each in light of the properties above.

Two approaches for designing parallel generation algorithms are known in literature. In the first approach used there are an arbitrary number (k) of processors available; each of them produces an interval of S/k objects, where S is total number of objects to be generated (for example, 1024 subsets of 10-elements set are produced by 4 processors generating subsets 1–256, 257–512, 513–768 and 769–1024, respectively). The best known technique is to apply a sequential algorithm on each interval (i.e. for each processor), and is used in [2] for permutations and combinations and in [7] for subsets and equivalence relations. This gives cost-optimal algorithms under measure 2a); however, the properties 3 and 5 are never satisfied. The latter applies also to the other algorithms appearing in the literature. Permutations are generated in [5, 10, 14] and others references given in [1, 2] while derangements are generated in [11] using a theoretical model more powerfull than EREW PRAM. In addition, property 2 is not satisfied for all mentioned algorithms (except [5] which does not meet property 1). Although more sophisticated approaches are used in these papers, surprisingly no better results are achieved compared to the simple one above, no matter which of the listed properties is taken into consideration.

In the second approach (that is adopted in the paper) m processors produce an m-element combinatorial object $a_1a_2 \ldots a_m$ such that processor i is responsible for producing element a_i. For example, the subset $\{2, 3, 5\}$ is produced in the following way: processor 1 produces 2; processor 2 produces 3; processor 3 produces 5. The algorithms are made adaptive by combining this approach with the former one. This approach enabled designing an algorithm to satisfy properties 3 and 5, for which the former approach fails.

Using this approach, optimal algorithms (under measure 2b) that satisfy all properties 1–7 listed above are designed to generate combinations m out of n elements in [3] and [12], and permutations of n out of n elements [4]. The first known algorithms to solve these two problems with this approach [6] and [2] were not cost-optimal and were designed for the more powerful EREW PRAM model.

In this paper, we consider the problem of generating m-subsets (subsets having at most m elements) and equivalence relations (set partitions) of the set $\{1, 2, \ldots, n\}$, in lexicographic order. Various sequential algorithms have been given for these problems ([15, 18, 19] for generating subsets; [8, 9, 15, 17] for generating equivalence

relations). All these techniques enumerate objects in lexicographic order, which is often emphasized for its speed, simplicity and possibility to cut or recover some objects easily. Note that, since each of the $S(m, n)$ subsets ($B(n)$ set partitions) requires $O(m)$ ($O(n)$) time to be produced as output, the best possible sequential algorithm runs in $O(mS(m, n))$ ($O(nB(n))$, respectively) time. If the time to produce the output is not counted, then there are faster algorithms, in $O(S(m, n))$ time for generating subsets, and in $O(B(n))$ time for generating set partitions. However, for set partitions the delay between producing two set partitions is $O(n)$, i.e. non-constant, in the worst case.

Recently, the fast generation of subsets and set partitions in parallel (using the first approach described above) has been studied in the literature. In [5] a parallel algorithm to generate the subsets of at most m out of n objects is presented. It runs on a linear array of k processors (including a selector), each producing an interval of consecutive subsets. The algorithm is cost-optimal but each processor requires memory of size $O(m)$. The algorithm [7] uses any number of independent processors and is also cost-optimal. Again, each processor requires memory of size $O(m)$, and has to deal with large integers. The same properties are also valid for the set partition algorithm from [7].

In this paper we solve the same problems using a different approach (described above as the second approach). We use a linear array of m processors, each having a constant size local memory (thus, the processors do not need as much memory space as in [5, 7]). It is well known that any algorithm for a linear array of processors can be simulated without penalty on other theoretical and practical models (SM EREW PRAM, hypercube, mesh-connected computer etc.); in other words, the model is practical and weaker than other known models. The algorithms require constant time per subset/equivalence relation, and therefore are cost-optimal. Thus we achieve cost-optimality, optimizing space (constant size memory per processor) at the same time, and avoid any calculation with very large integers. All properties 1–7 listed above are satisfied with our algorithms.

Our algorithms are cost-optimal under the measure 2b). This measure is considered since in applications we are aware of the fact that combinatorial objects generated need to be reported (each object with all its elements) for further processing. For example, the list of bases of the set of predicate functions (mapping n-valued input to binary output) can be constructed using the list of equivalence relations on the set of n elements [13]. The subset generation algorithm has a similar application in base enumeration [19] and in listing all clique Boolean functions [16]. In addition, our parallel algorithm for generating equivalence relations has the advantage of constant time delay between two relations, which is not the case for any previously known sequential or parallel algorithm.

We further elaborate the approach used in the paper. Consider a parallel algorithm to generate a class of combinatorial objects (subsets, equivalence relations, or some other class of objects) using n processors and n shared variables $D_1, \ldots, D_n$, where $D_1 D_2 \ldots D_n$ represents one of the objects in a given class. Each processor i is

responsible for maintaining D_i by, we hope, reading only data from processors $i-1$ and $i+1$. Each processor has a while loop that at each iteration prints D_i so that, since the processors act in lock-step fashion, each iteration will print an object D. We might naively expect that each processor i would maintain D_i. Thus, the body of the loop of each processor would be

Print D_i;
update D_i;

If objects to be generated are subsets, D_i can be updated using a known sequential algorithm. Suppose that $D_1, \ldots, D_r$, $r \leq n$, is the current subset. Then the next subset is sequentially obtained in a constant number of operations (see section 2), each performed on elements with indices $r-1, r$ or $r+1$; this has a straightforward parallelization, and the obtained algorithm is cost-optimal, plus other desirable properties listed above. The subset generating algorithm is described here to illustrate the model and the approach. However, if equivalence relations are to be generated in parallel, the computation of D_i by a known sequential generating procedure would require too much time ($O(n)$ in the worst case), or shared memory with too many (read/write) references (an $O(\log n)$ algorithm on an SM EREW PRAM model can be designed in a manner that is analogous to the combination and permutation procedures from [1]). We therefore look for a slight variation on this theme.

The paper is organized as follows. Section 2 presents a parallel algorithm for generating subsets. The algorithm is a straightforward parallelization of an existing serial algorithm and is given to illustrate the approach and the model used, and to complete the repertoire of combinatorial objects generation algorithms. The main contribution of the paper is a parallel technique for generating equivalence relations, described in sections 3, 4 and 5. The technique does not follow any existing serial algorithm; in fact, it is much simpler than any of them. In section 4 we show that the method used to enumerate equivalence relations can be applied for generating some other types of combinatorial objects, such as variations; we also describe the criteria for suitability of our approach. Section 5 describes how our algorithm can be made adaptive. A conclusion containing some open problems is also given.

2. Subsets.

In this section we consider the problem of generating all r-subsets (subsets containing r elements) of the set $\{1, \ldots, n\}$ for all r, $1 \leq r \leq m \leq n$ (i.e. all subsets with up to m elements) on a linear array of m processors. We assume that each subset will be represented as a sequence $a_1 \ldots a_r$, where $1 \leq a_1 < \ldots < a_r \leq n$.

Recall the definition of lexicographic order of sequences. For two sequences $a = (a_1, \ldots, a_p)$ and $b = (b_1, \ldots, b_q)$, $a < b$ is satisfied if and only if there exists an $i (1 \leq i \leq q)$ such that $a_j = b_j$ for $1 \leq j < i$ and either $a_i < b_i$ or $p = i - 1$. This order has an important property that enables simple calculation with sequences.

A backtracking algorithm for generating all subsets with up to m elements can be given as follows (cf. [19]):

```
read (n); r := 0; a_r := 0;
repeat
  if a_r < n and r < m
    then {a_{r+1} = a_r + 1; r := r + 1} (*extend*)
    else if a_r < n then a_r := a_r + 1 (*increase last*)
                    else {r := r - 1; a_r := a_r + 1}; (*reduce*)
  output a_1, ..., a_r
until a_1 = n
```

This algorithm has a straightforward implementation on a linear array of m processors. Each processor j ($1 \leq j \leq m$) is responsible for producing element a_j (if $j \leq r$, where r is the number of elements in current subset; otherwise it produces no element, and we may assume $a_j = 0$ for such processors). Given a subset $a_1 \ldots a_r$, the next subset (in lexicographic order) is determined in a constant number of operations, and each operation is done on either a_{r-1}, a_r, or a_{r+1}. Thus, all operations are done on processors $r - 1$, r and $r + 1$ which have direct communication (processor r is connected to both processor $r - 1$ and $r + 1$). In addition to the field a_j, each processor has a field m for message exchange ($m_r = 1$ and $m_j = 0$ for $j \neq r$), and fields to keep m and n. The parallel algorithm can be described as follows.

```
for each processor j (1 ≤ j ≤ m) do in parallel
  m_j := 0; a_j := 0;
  terminate_j := 0; (*termination flag*)
  if j = 1 then {m_j := 1; a_j := 1; output a_j};
                (*output first subset*)
  repeat
    if m_j = 1 (*j = r, index of last element*)
    then if a_j < n and j < m
      then m_j = 2 (*extend*)
        else if a_j < n then a_j = a_j + 1 (*increase last*)
                        else m_j := 3 (*reduce*);
    if j > 1 then {read m_{j-1};
                   if m_{j-1} = 2 then (*extend*)
                                  {read a_{j-1}; a_j := a_{j-1} + 1;
                                   m_j := 1 (*new last element')}};
    if j < n then {read m_{j+1};
                   if m_{j+1} = 3 then (*reduce*)
                                  {a_j := a_{j+1};
                                   m_j := 1 (*new last element*)}};
```

```
    if m_j > 1 then m_j: = 0; (*delete old message*)
    if a_j > 0 then output a_j;
  if j = 1 and a_j = n then {terminate_j := 1; a_j := 0};
  if j > 1 then {read terminate_{j-1};
                 if terminate_{j-1} > 0
                 then terminate_j := terminate_j + 1};
  until terminate_j := n - j + 1
```

In the algorithm all processors will terminate simultaneously. The termination is initiated by processor 1 when it recognizes $a_1 = n$. The termination message is communicated in n steps to processor n; during the communication there is no new output by any processor since $a_j = 0$ for all of them.

3. Equivalence relations.

Let $Z = \{1, \ldots, n\}$. An equivalence relation (or partition) of the set Z consists of classes $\pi_1, \ldots, \pi_k$ such that the intersection of every two classes is empty and their union is equal to Z. The algorithms [8, 9, 15, 17] generate equivalence relations in lexicographic order of codewords, where a codeword $x_1 \ldots x_n$ represents an equivalence relation of the set Z if and only if $x_1 = 1$ and $1 \leq x_r \leq g_{r-1} + 1$ for $2 \leq r \leq n$, where $x_i = j$ if i is in π_j, and $g_r = \max(x_1, \ldots, x_r)$ for $1 \leq r \leq n$. A list of codewords and corresponding equivalence relations for $n = 4$ is given in Fig. 1.

The next equivalence relation is found from the current one by a backtracking or recursive procedure in all known sequential generating techniques that deal with lexicographic order of elements; in both cases an increasible element (one for which $x_j \leq g_{j-1}$ is satisfied) with the largest possible index t is found ($t \leq n - 2$); we call this element the *turning point*. For example, the turning point of equivalence relation 1123 is the second element ($t = 2$). A straightforward implementation of this leads to an $O(n)$ worst case delay between two equivalence relations (which is not desirable), even in case a linear array of processors is to be used. Also, for processors which do not take part in a backtracking step, deciding on exactly when to produce the next element, may require either large counters or a sophisticated procedure.

A standard parallel algorithm would run in $O(\log n)$ time per equivalence relation, and would also require a more powerfull (and theoretical) SM EREW PRAM model (shared memory exclusive read exclusive write parallel random access machine). We present a generating procedure in which each processor can produce the corresponding element in the next equivalence relation in constant time, and on a weaker model: linear array of processors.

By a *run of a processor* j we mean the output of the processor when $x_1 \ldots x_{j-1}$ are fixed ($1 \leq j \leq n$). One run of processor j ($2 \leq j \leq n$) consists of the elements $1, 2, \ldots, g_{j-1} + 1$, such that each is repeated an appropriate number of times in succession. Processor 1 produces the element 1 all the time while processor n always changes its

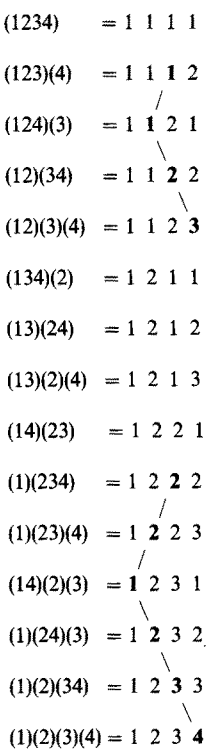

Fig. 1. List of codewords and corresponding equivalence relations for $n = 4$.

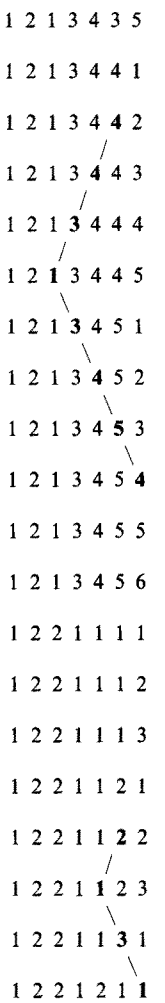

Fig. 2. Fragment of equivalence relations of the 7-element set.

element x_n. Other processors will produce several runs. The total number of runs of processor j is B_{j-1}, where B_n is the number of equivalence relations of a set of n elements (Stirling number of the second kind). It is obvious that processor j can produce a run independently on the data in other processors provided it is given g_{j-1} for the run. However, this approach would lead to dealing with large integers (the total number of equivalence relations is exponential in n). We therefore choose another approach. It is easy to show that processors $j + 1, j + 2, \ldots, n$ will always finish their current runs whenever processor j does (for example, processors 3 and 4 have finished their current runs in equivalence relation 1123 simultaneously; the next relation is 1211). Here we describe a very simple technique to achieve constant

time delay in generating any equivalence relation, thus leading to an optimal $O(B_n)$ time generating algorithm.

In order to make simultaneous changes, the processors j for $t < j \leq n$ should receive appropriate message in good time. The message is initiated from processor n, routed to the processor t (turning point) and sent back toward processor n. There are, obviously, at least $2(n - t)$ steps necessary to broadcast all the information needed for simultaneous change (the indices t and g_t). Here one step corresponds to the output of one set partition and routing the message from a processor to a neighboring one. If t is less than $n/2$ (the limit is not strict), one run of processor n is too short to communicate the message to processor t and back to n; there will be a delay. Two runs of processor n are, on the other hand, sufficient for this.

The method is illustrated in Figures 1 and 2 and described below in more detail. Fig. 2 shows a fragment of equivalence relations of the 7-element set. Messages looking for the turning point are depicted as "/" while messages with desired information are marked by "\".

4. Algorithm for generating equivalence relations.

Initially $x_j = 1$ and $g_j = 1$ for each processor j ($1 \leq j \leq n$). At a given step, each processor j communicates its immediate neighbors $j - 1$ and $j + 1$ (if they exist) to check whether any change in the "system" has appeared. In general, messages will be rarely sent, and processors will not change their status unless appropriate messages are received. Messages are communicated through the field m. If $m_j = 0$, then processor j sends a "no change" message to its neighbors.

There are two kinds of "active" messages ($m_j = 1$: "looking for the turning point"; $m_j = 2$: "the turning point has been found"). When a message of the first kind from $j + 1$ ($m_{j+1} = 1$) is received, processor j will immediately check whether $x_j \leq g_{j-1}$ is satisfied, in which case it is the desired turning point. If j is not a turning point, it will pass (in the next step) the same message to its lower numbered neighbor $j - 1$ (by assigning $m_j := 1$).

On the other hand, if j is indeed the turning point, then it will send an appropriate message back to processor $j + 1$ by assigning $m_j := 2$, $t_j := j$ (to mark the turning point), and $g_j := \max(g_j, x_j + 1)$. Every processor j for $j > k$ receives the mentioned message "package" from processor $j - 1$, at the time when $m_{j-1} = 2$. The index t of the turning point and new value of g_j ($g_j = g_{j-1}$) are passed to processor j together with the message (the new value is not in effect during the waiting period). Processor j will still produce the same element during the waiting time which is equal to $w_j = 2t - 3 + n - j$ steps (so that the change is done synchronously). Note that a processor may keep the status of carrying active message ($m_j = 1$ or $m_j = 2$) for one step only; in the next step m_j is set back to 0.

Processors n and $n - 1$ play somewhat different roles. In addition to the described general job, processor $n - 1$ also increases its element x_{n-1} during the process of

collecting information; it happens when $x_n = g_{n-1} + 1$. Processor $n-1$ also initiates the search for turning point. A message will be activated ($m_{n-1} := 1$) if $x_n = 2$ and $x_{n-1} = g_{n-1}$. The processor n will increase its field x_n by 1 if it did not reach $g_{n-1} + 1$ yet; otherwise it starts a new run from $x_n := 1$.

Summarizing, we obtain the following theorem.

THEOREM. *The algorithm described above generates all equivalence relations of n objects with constant delay per equivalence relation (thus, in optimal $O(B_n)$ time) on a linear array of n processors, where each processor*
- *has memory of constant size, and*
- *can generate elements without need to deal with large integers, e.g. B_i $(1 \leq i \leq n)$.*

PROOF. Using mathematical induction we prove that at step k the output of the algorithm is exactly the kth equivalence relation. This is obviously satisfied for the first equivalence relation by the initial conditions. Assuming that this is true for the first $k-1$ relations, we prove that the kth output is precisely the kth equivalence relation, in lexicographic order. We consider the output of each processor j for $1 \leq j \leq n$. For $j = n$ the element x_n is increased by 1 unless it has the maximal possible value $g_{n-1} + 1$, in which case the next value is 1. This property of lexicographic order enumeration of relations is exactly followed in the described algorithm.

Assume now $j < n$. In lexicographic order, the element x_j will not change for a number of relations, in both the lexicographic order and in the described algorithm. It will change its value only when $x_t = g_{t-1} + 1$ for each $t > j$ (i.e. if all processors $j+1, j+2, \ldots, n$ produce the maximal possible values). It is easy to show that $x_t = g_j + t - j = g_{n-1} + t - n + 1$ is then satisfied for $t > j$ (since $x_t = g_t$ when x_t has maximal value);; from $x_t > 0$ it follows that $n - t < g_{n-1} + 1$ and thus $2(n-t) < 2g_{n-1} + 1$. When this happens, x_j increases by 1 unless it has already reached its maximal possible value $g_{j-1} + 1$, in which case it changes its value to 1. We check whether this property of lexicographic order is properly followed by the algorithm. At the moment of change of x_j, by induction, x_{n-1} has correct value g_{n-1}, which is its maximal possible value, i.e. the end of its current run of $g_{n-1} + 1$ relations. The former run contained g_{n-1} relations. Thus the two last runs contained $2g_{n-1} + 1$ relations. The message was initiated by processor $n-1$ at the beginning of the next to the last run. Since $2(n-t) < 2g_{n-1} + 1$ there was sufficient time for the initiated message to reach processor t ($n - t$ steps) and to return back (another $n - t$ steps). The waiting time can also be checked similarly. Therefore the information about change is available to the processor j together with the exact moment for change. This means that both lexicographic order and algorithm will do the same change, which completes the proof of correctness of the algorithm. The other two statements are obvious from he description. ■

5. Formal description of the algorithm.

A formal statement of the algorithm described in the previous section is given below.

```
For each processor j (1 ≤ j ≤ n) do in parallel
  x_j := 1; g_j := 1; m_j := 0; w_j := 0; t_j := 0; new_g_j := 0;
  repeat
    print x_j; read m_{j+1}; read m_{j-1}; read g_{j-1}; m_j := 0;
    if j = n then {if x_j = n then stop;
                   if x_j ≤ g_{j-1} then x_j := x_j + 1 else x_j := 1;}
    if j = n - 1 then {read x_{j+1};
                       if x_{j+1} = g_j + 1 and x_j ≤ g_{j-1}
                         then {x_j := x_j + 1; g_j := max(g_j, x_j)}
                       if x_{j+1} = 2 and x_j = g_{j-1} then
                         (*activate search*) m_j := 1}
    if j < n then
            {if m_{j+1} = 1 then if x_j ≤ g_{j-1} and j ≤ n - 2 then
               {(*search successful*) m_j := 2;
               {(*set up turning point*) t_j := j;
               (*set up waiting time*) w_j := j - 3 + n;
            (*new value of g*) new_g_j := max(g_j, x_j + 1);} else m_j := 1;
              if m_{j-1} = 2 then {new_g_j := g_{j-1}; m_j := 2; read t_{j-1};
            (*read the index of turning point*) t_j := t_{j-1};
            (*set up waiting time*) w_j := 2t_j - 3 + n - j;}
            if t_j > 0 then if w_j > 0 then w_j := w_{j-1}
                                       else {g_j := new_g_j;
                                       if t_j := j then x_j := x_j + 1 else x_j := 1;
                                       t_j := 0}}
  until t_j = 1 and w_j = 0
```

6. Variations.

In this section we show that our method can be used to enumerate some other types of combinatorial objects, in lexicographic order. As an illustration we choose variations of the set $\{1,\ldots,n\}$. A variation of m out of n elements is any sequence $x_1,\ldots,x_m$ such that $1 \leq x_j \leq n$ for $1 \leq j \leq m$. Obviously, there are n^m such variations. The turning point of a variation is the element with greatest index for which $x_j < n$ is satisfied. A search for turning point can again be initiated (in advance) by the processor $m-1$ when this processor started to produce its run for the value $n-1$. There are n values of x_m when $x_{m-1} = n-1$ and n values of x_m when $x_{m-1} = n$. These $2n$ values are sufficient for the message to search for the turning

point and return with its value. The other modifications of the set partition algorithm are also obvious.

In general, any combinatorial enumeration problem (for generating m out of n objects) for which there are between $\Omega(m)$ and $O(n^c)$ objects (c any positive integer) when the values of $x_1 \ldots, x_{m-1}$ are fixed (or, at least, when element x_{m-1} has the maximal or next to the maximal possible value) can be solved using our technique. The listed property seems to be the key for applying the described method. $\Omega(m)$ is necessary since searching for the turning point may require that much time. The other limit $O(n^c)$ on the number of objects is needed to avoid counters that are very large numbers. For instance, permutations of n elements do not meet this requirement since the number of mentioned objects is $O(1)$.

7. Adaptive algorithms.

The described algorithms are designed to be executed on a linear array of exactly m (in case of subsets) and n (in case of set partitions) processors. In order to obtain adaptive algorithms (an algorithm is adaptive if it can run on any number of processors), one can divide the number k of available processors into k/m groups for subsets and k/n groups for set partitions, with m and n processors, respectively, in each group. Each group will run the corresponding algorithm described in this paper, starting with an object and finishing with another one. Dividing the job into a given number of equal parts, i.e. determining the first and the last object of a given group of processors, has been described in [7] (a numbering system, i.e. unranking procedure from [15] can also be used). This calculation involves very large numbers; however this is done once as a preprocessing step (and can even be done in parallel by the whole linear array of processors) and therefore does not significantly increase the time complexity of our algorithms.

8. Conclusion.

We succeeded in deriving an optimal algorithm for generating subsets and equivalence relations of n objects on a linear array of processors. The algorithm uses n processors and produces relations in lexicographic order and in constant time per relation. Algorithms with the same characteristics were designed for some other types of combinatorial objects: combinations [3, 12] and permutations [4].

One of the remaining open problems is generating all derangements of m out of n objects, by an algorithm satisfying properties 1–7. A recent paper [11] does not meet properties 2 (in both sences), 3, 4, and 5. Another similar problem is generating all permutations of m out of n elements, and all cyclic permutations of n elements satisfying properties 1–7. Partitions and compositions of integers are further problems to consider. Finally, many of the mentioned problems are not solved optimally if different kinds of cost measures (discussed in the introduction) are applied.

Acknowledgement.

The author appreciates the comments made by a referee, especially on the measure of optimality.

REFERENCES

1. S. G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
2. S. G. Akl, *Adaptive and optimal parallel algorithms for enumerating permutations and combinations*, The Computer Journal, 30, 5, 433–436, 1987.
3. S. G. Akl, D. Gries and I. Stojmenović, *An optimal parallel algorithm for generating combinations*, Information Processing Letters, 33 (1989/90) 135–139.
4. S. G. Akl, H. Meijer and I. Stojmenović, *Optimal parallel algorithms for generating permutations*, Technical report No. 90-270, Dept. Comp. and Inf. Sci., Queen's Univ., Kingston, January 1990.
5. G. H. Chen and M.-S. Chern, *Parallel generation of permutations and combinations*, BIT, Vol. 26, 1986, 277-283.
6. B. Chan and S. G. Akl, *Generating combinations in parallel*, BIT, 26, 1, 2–6, 1986.
7. B. Djokić, M. Miyakawa, S. Sekiguchi, I. Semba and I. Stojmenović, *Parallel algorithms for generating subsets and set partitions*, Proc. SIGAL Int. Symp. on Algorithms, Tokyo, Japan, August 1990.
8. B. Djokić, M. Miyakawa, S. Sekiguchi, I. Semba and I. Stojmenović, *A fast iterative algorithm for generating set partitions*, The Computer Journal, Vol. 32, No. 3, 1989, 281–282.
9. M. C. Er, *Fast algorithm for generating set partitions*, The Computer Journal, 31, 3, 283–284, 1988.
10. P. Gupta and G. P. Bhattacharjee, *Parallel generation of permutations*, The Computer Journal Vol. 26, No. 2, 1983, 97–105.
11. P. Gupta and G. P. Bhattacharjee, *A parallel derangement generation algorithm*, BIT, Vol. 29, 1989, 14–22.
12. C. J. Lin and J. C. Tsay, *A systolic generation of combinations*, BIT, Vol. 29, 1989, 23–36.
13. M. Miyakawa and I. Stojmenović, *Classification of* P_{k2}, Discrete Applied Mathematics, 23, 1989, 179–192.
14. M. Mor and A. S. Fraenkel, *Permutation generation on vector processors*, The Computer Journal, Vol. 25, No. 4, 1982, 423–428.
15. A. Nijenhius and H. S. Wilf, *Combinatorial Algorithms*, Academic Press, N.Y., 1978.
16. G. Pogosyan, M. Miyakawa and A. Nozaki, *On the number of clique Boolean functions*, Discrete Applied Mathematics, to appear.
17. I. Semba, *An efficient algorithm for generating all partitions of the set* $\{1, \ldots, n\}$, Journal of Information Processing, 7, 41–42, 1984.
18. I. Semba, *An efficient algorithm for generating all k-subsets* $(1 \leq k \leq m \leq n)$ *of the set* $\{1, 2, \ldots, n\}$ *in lexicographic order*, Journal of Algorithms, 5, 281–283, 1984.
19. I. Stojmenović and M. Miyakawa, *Applications of subset generating algorithm to base enumeration, knapsack and minimal covering problems*, The Computer Journal, 31, 1, 65–70, 1988.