Taylor & Francis
Taylor & Francis Group

Listing combinatorial objects in parallel

IVAN STOJMENOVIC*

SITE, University of Ottawa, Ottawa, Ont., Canada K1N 6N5

This article surveys parallel generation algorithms for listing all combinatorial objects of certain type.
The algorithms are designed for a very simple model, linear array of processors. The methods are divided
into three groups: division of instances into groups, shared instances and combined methods.

Keywords: Combinatorial algorithms; Linear array of processors; Permutations; Combinations; Subsets

1. Introduction

The number of instances of a combinatorial object is typically exponential in the size of
instance; for example, there are $n!$ n-permutations (i.e. permutations of n elements).
Generating all instances is, therefore, a time consuming operation, and sequential algorithms
(these running on today's computers which have one processor, i.e. one CPU which performs
instructions in a sequence, one by one) may prove to be too slow in practice to be used for
testing all instances of given object.

A natural choice to overcome this difficulty is to use new technology of parallel computer
architectures, where several processors or CPUs may run simultaneously. Since the advent of
parallel computers, interest has naturally been paid to the development of parallel algorithms
for generating combinatorial objects. Our purpose in this article is to describe a collection of
such parallel algorithms. The algorithms presented satisfy a number of strict conditions, and
as a result are optimal in every known and reasonable sense.

The cost of a parallel algorithm is the product of the number of processors used and its
running time. A parallel algorithm is said to be cost-optimal if its cost matches a lower bound
on the number of operations required to solve the problem sequentially.

Various ways of using multiple processors appeared in literature but most of them can be
classified into the following three approaches, which will be described in subsequent
sections.

  Method 1. Division of instances into groups.
  Method 2. Shared instances.
  Method 3. Combination of methods 1 and 2.

---

*Corresponding author. Email: ivan@site.uottawa.ca

A parallel algorithm is said to be adaptive if it is capable of modifying its behavior according to the actual number of processing elements available on a particular machine being used to execute the algorithm. In other words, an algorithm is adaptive if it can run on a parallel model of computation with an arbitrary number k of processors.

2. Division of instances into groups

In the first approach used, there are arbitrary number of k processors available; each of them produces an interval of N/k objects, where N is total number of instances to be generated (for example, 1024 subsets of 10-elements set are produced by four processors generating subsets ranked $1-256$, $257-512$, $513-768$ and $769-1,024$, respectively).

A simple technique, which can also be used in the sequential listing, is to generate all combinatorial objects of given kind by unranking 1st, 2nd, 3rd,... object in that order. However, unranking functions are usually computationally expensive for efficient listing. The best-known technique to follow the approach of division of instances into groups is to apply a sequential algorithm on each interval (i.e. for each processor).

This method is used by Akl [1] and Kokosinski [22] to generate combinations and permutations. We now elaborate the approach in more detail. Suppose a sequential generation algorithm is available, with a procedure unrank (t, X) to discover the tth generated object X. The job is divided equally into k processors. The distribution is such that processor i generates objects from $(i - 1)\lceil N/k \rceil + 1$ to $i\lceil N/k \rceil$, except for the last processor which should finish enumeration at the object ranked N. It is assumed that the sequential algorithm is capable of generating $t + 1, t + 2,...$ objects for a given tth object. Most iterative algorithms are suitable for such division of all instances of given combinatorial object into k groups. On the other hand, for most recursive sequential algorithms, the latter property is difficult to achieve.

In order to start producing its group of instances of given combinatorial objects, each processor needs to be given the beginning and end of its interval of consecutive instances. It can be done by a master processor, which can supply necessary data and program for generating all instances. After the instances are divided and each processor is given the beginning and end of its interval of consecutive instances, there is no need for cooperation between processors, as each of them may produce its instances by running a sequential algorithm. Because communication among processors is not needed, k independent processors, the simplest possible model one can imagine, suffice for our purpose. It is also easy to verify that the algorithm is cost-optimal. The only undesirable property is that each processor needs the ability to store the whole instance it generates, i.e. it needs memory of size $\Omega(n)$, where n is the size of each instance.

The advantage of the method is its simplicity, adaptivity and cost-optimality (it can be checked directly that the product of the number of processors k and running time is asymptotically of the order of the time complexity of sequential algorithm). The cost-optimality is valid for both versions of producing combinatorial objects: generating and listing.

There are two problems with the method. One problem is that the division into groups may involve large integers or could be computationally expensive. The other problem is that the processors should be able to store the whole instance, i.e. they must have storage of size $O(m)$

where m is the size of instance. This means that processors must be powerful, close to the power of the processor on a sequential computer. In reality, interconnection networks are usually made of a large number of small processors. Therefore, the realistic assumption is that one processor has a memory of constant size, and because of that alternative approaches for generating combinatorial objects must be investigated (Method 2, for example).

We now address the problem of using large integers when dividing instances into (approximately) equal sets or groups. One can use a numbering system or unranking to find the initial and final instance in each group. However, most known numbering systems use large integers and are, for practical purposes, inefficient.

In [37], we suggested a new numbering system, for some kinds of combinatorial objects, which does not deal with large integers. The new system finds, for given g such that $1 \leq g \leq n$, the combinatorial object $x = x_1, \ldots, x_m$ such that the ratio of the number of objects preceding x and the total number of objects is $\leq g$. In other words, it finds the object with the ordinal number $\lfloor gN \rfloor + 1$.

The processor $j(1 \leq j \leq k)$ will produce instances numbered from $\lfloor (j - 1)N/k \rfloor + 1$ to $\lfloor jN/k \rfloor$. Thus, we apply the mentioned algorithm for $g = j/k$, $0 \leq j \leq k$, to find the beginning and the end of each group. Note that with such division and truncations during computation, the number of objects in each group to be generated is not strictly equal, but is balanced asymptotically.

As an example, a parallel subset generation algorithm can be designed as in [15]. It uses the set representation of subsets. Without loss of generality it is assumed that subsets are taken from the set $\{1,2,\ldots,n\}$.

```
read (n,k);
for i ← 1 to k do in parallel {
    g ← ⌈(2ⁿ − 1)/k⌉; t ← (i − 1)*g + 1;
    unranksets (t, n, x₁, x₂,..., xᵣ);
    u ← 0;
    repeat
        process x₁, x₂,..., xᵣ; u ← u + 1;
        if xᵣ < n then extend else reduce;
    until u = g or x₁ = n
extend ≡ {xᵣ₊₁ ← xᵣ + 1; r ← r + 1}
reduce ≡ {r ← r − 1; xᵣ ← xᵣ + 1}.
```

The method can similarly be applied to other combinatorial objects, combining the unranking function and sequential generating algorithm. It has been applied to generate permutations and combinations [1], set partitions and (m,n)-subsets out of the set $\{1,2,\ldots, n\}$ [15], topological orders [45] and (m,n)-permutations [17]. There is one additional detail to clarify that may depend on particular method. The unranking function gives the first instance in a group but there are usually some more variables in generation process, and their appropriate values should be initialized as if they were obtained in the sequential algorithm at the moment it is supposed to report the initial instance for given group. The above algorithm has no such variable.

3. Shared instances

In the second approach, m processors produce a m-element combinatorial object $a_1, a_2, \ldots, a_m$ such that processor i is responsible for producing element a_i. For example, subset {2,3,5} is produced in the following way: processor 1 produces 2; processor 2 produces 3; processor 3 produces 5.

This approach is used by a number of authors, which gave solutions to various generating problems. The model of computation used also differed. Some authors use very powerful and theoretical models like CREW PRAM or EREW PRAM, which stand for concurrent (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 write into the same memory location simultaneously. In the CREW PRAM, several processors can read simultaneously from the same memory location while in the EREW PRAM this is not allowed. Using this model, algorithms are designed to generate combinations [2], permutations [2], etc. It should be noted that designing algorithms in the model requires significant experience and background in parallel computing.

The model of parallel computation that will be used here is very simple and such that one can follow the algorithms without having any background in parallel computing. In this sense, the article is self-contained. On the other hand, the solutions that are obtained are in all cases faster than those designed for PRAM, achieving the best possible constant delay in the worst case.

The model of computation chosen to solve the problem of generating combinatorial objects is the linear array of processors. In this model, n processors, indexed $1-n$, are connected in a linear array. As shown in figure 1, processor i, $2 \le i \le n - 1$, is connected by a bidirectional link to its two neighbors, processors $i - 1$ and $i + 1$; processors 1 and n have only one neighbor, to which they are connected.

In one step (requiring constant time), a processor can send a datum (of constant size) to a neighbor. If two processors that are not directly connected wish to communicate, they have to do so by sending data through the processors separating them. Thus a message from processor 1 takes $n - 1$ steps to reach processor n. Alternative models (counter model, associative model) that also allow $O(1)$ time parallel algorithms were studied by Kokosinski [22−24].

There exists a difference between parallel and distributed computing. In a distributed (or asynchronous) computing, there are several processors or computers connected in a network and solving jointly the same problem, but each of them is running its own program on its own data. On the other hand, in a parallel computation, all processors are running synchronously the same program, but on different data (SIMD—single instruction multiple data). Usually any parallel architecture has a master processor that supplies the other processor with the program, which is common for all processors. In a parallel algorithm, processors are operating in a lock step fashion: they all perform the same instructions at any given time and
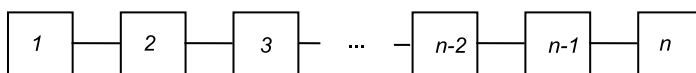

Figure 1.   A linear array of processors.

the next instruction starts when a signal to perform is given by a master processor or clock. It is also called synchronized computation.

Using this approach, algorithms are designed to generate various combinatorial objects. We now describe some optimality properties of parallel generation algorithms, explain the importance of these characteristics, and show that if an algorithm possesses these properties, then the algorithm is optimal in every known and reasonable sense.

A combinatorial object consists of n elements selected from a set of m elements $S = \{s_1, s_2, \ldots, s_m\}$, and arranged in a particular order. There are usually many instances of a combinatorial object. For example, two combinations of n out of m given elements are distinct if they differ in the elements they contain, while two permutations of m given elements are distinct if they differ in the order of their elements.

PROPERTY 1. The algorithm generates instances of an object from a set $S = \{s_1, s_2, \ldots, s_m\}$ of arbitrary elements, on which an order relation $<$ is defined such that $s_1 < s_2 < \ldots < s_m$.

This property is important as it allows us to distinguish between algorithms that generate instances of an object from any ordered set, and algorithms that work only when S is a particular set (e.g. when $S = \{1, 2, \ldots, m\}$). In a sequential algorithm, this property is trivially satisfied as the index i can easily be replaced by a "table" value a_i. However, in a parallel algorithm the corresponding value a_i may be stored in a local memory of a processor and the access to it from another processor may become computationally expensive.

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. Equivalently, the cost of the parallel algorithm should match the running time of an optimal sequential algorithm.

This property can be further specified according to the way in which the lower bound is defined. In our context, we identify two such definitions:

(a) The time required to "create" the instances of an object, without actually producing as output the n elements of each instance, is counted. Thus, for example, an optimal sequential algorithm in this sense, would generate all n-bit strings in $\theta(2^n)$ time, i.e. time linear in the number of instances.

(b) The time to actually "output" each instance in full is counted. Here, an optimal sequential algorithm generates all n-bit strings in $\theta(n2^n)$ time, since it takes $\theta(n)$ time to produce a string.

As mentioned at the beginning of this section, we interpret "generating" a combinatorial object as "producing" its instances as output. Therefore, we use definition (b) to determine whether the cost of a parallel algorithm is indeed optimal. Designing combinatorial object generation algorithms whose cost is optimal under measure (a) remains an open problem at the time of this writing.

PROPERTY 3. The time required by the algorithm between any two consecutive instances of an object is constant.

Constant time to produce each instance of an object is, of course, the best any algorithm can aim to achieve from the theoretical point of view. A constant time delay between outputs is also important in applications where the output of one computation serves as input to another; this is particularly true in applications using systolic arrays. As usual in sequential computation, we assume that a processor requires constant time to perform an elementary operation. Examples of such operations are the addition or comparison of two numbers, each encoded using $\theta(\log m)$ bits.

PROPERTY 4. The model of parallel computation should be as simple as possible.

From both the theoretical and practical points of view, an algorithmic result is more valuable if it is derived on a parallel model of computation that makes the fewest assumptions possible about processor connectivity. Indeed, the weaker the model, the stronger an optimality result, since a more powerful model can simulate the algorithm with no increase in running time. Arguably, the simplest such model is the linear array of processors. This model is weaker than the two-dimensional array or hypercube models, which are in turn weaker than the parallel random access machine [2]. In addition, the linear array is practical, and it is amenable to VLSI implementation.

PROPERTY 5. Each processor needs as little memory as possible, specifically a constant number of words each of $\theta(\log m)$ bits. Each word is thus capable of storing a binary encoding of one of the elements of $S = \{s_1, s_2, \ldots, s_m\}$, or of an integer no larger than m.

Theoretically, given a collection of algorithms for a certain problem, all with the same running time but different memory requirements, the one needing the least memory is usually the best. In practice, when the algorithm is implemented using a collection of tiny processors on a chip, small memory usage is preferable, if not necessary. The above property implies that no processor can by itself store an array of size m, or a large combinatorial number such as $m!$.

PROPERTY 6. The algorithm is adaptive, i.e. it is capable of adjusting itself automatically to the number of processors available.

The actual number of processors that can be used to run an algorithm is not always equal to the theoretical number for which the algorithm was designed. In these circumstances, an adaptive algorithm is desirable. Such an algorithm is designed to run on a linear array with an arbitrary number of processors; at run time it adjusts itself to the existing number of processors, and executes correctly. There are two cases to consider:

(a) $k < n$: here each processor will do the job of n/k processors in the original algorithm (with n/k rounded appropriately if not an integer, so that the last processor does slightly less work);

(b) $k \geq n$: here the processors are divided into $g = \lfloor k/n \rfloor$ groups of n processors each (with up to $n - 1$ leftover processors), such that each group produces an interval of $h = N/g$ consecutive instances, where N is the total number of instances of the given object. The first and the last instance in each group can be determined in a preprocessing step by

applying known unranking functions [37]. Thus, processors of the ith group, $1 \leq i \leq g$, begin with the instance corresponding to the integer $j = ((i - 1)h) + 1$.

The above procedure is sequential and is supposed to be done by one processor in each of the groups (in the second approach; in the first approach only one processor is responsible for each group). To avoid the need for more than constant space for this processor, one can decide that the processor, which normally should produce the last element in any object (for a given group) will find the initial object as described. As soon as the new element of the initial object is found, all currently known elements are shifted towards the first processor in a given group. In this way, there is no need to store the full object by the chosen processor. In case of permutations there is a list of the remaining elements. The list can be stored one element per processor, and the desired element can be extracted by a shift operation. This will require $O(n)$ time per element, or $O(n^2)$ time to find the initial permutation for a given group (assuming no large integers are used).

Since all algorithms presented here can be made adaptive in the way described above, we do not emphasize in the sequel the transformation required in each case to satisfy Property 6. Note that, using described method and method of unranking without large integers [37], the algorithms for generating combinations, permutations and t-ary trees can be made adaptive without using calculations with large integers.

The recent survey paper [10] lists lexicographic order as other desirable characteristics. Lexicographic ordering is the most commonly used ordering primarily because it has a simple characterization, but also because it agrees with numerical or dictionary ordering and is intuitive. Further, with this ordering it is easy to verify that all instances have been generated. Here, we accept minimal change order as an acceptable alternative, especially for applications where the order of instances is not important. In some cases, most notably permutations, it has led to significantly simpler generating algorithm.

Although not formally listed, we also believe that the simplicity of algorithms is also a desirable property. Readers and users should be able to understand algorithm and to implement it with reasonable programming efforts.

In the following sections, we describe parallel algorithms for generating subsets, combinations and permutations. Each algorithm satisfies most, if not all, of the above properties. We survey other known parallel algorithms for generating combinatorial objects and list some open problems.

4. Generating subsets

In [13], a parallel algorithm to generate the (m,n)-subsets is presented. With respect to our criteria, it does not satisfy Properties 3 and 5, and also does not follow lexicographic order of subsets.

We consider generating (m,n)-subsets out of set $\{1,2,\ldots,n\}$ using Method 2 (shared instances). The sequential algorithm in Section 2 finds the next subset in a constant number of operations, each performed on elements with indices $r - 1$, r or $r + 1$. This characteristic of an algorithm enables its straightforward parallelization, and the obtained algorithm satisfies Properties 2–5. The 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 the current subset; otherwise it produces no

element, and we assume $a_j = 0$ for such processors). Since the next subset of $x_1 \ldots ,x_r$ is determined by updating at most elements x_{r-1}, x_r and/or x_{r+1}, only processors $r - 1$, r and $r + 1$ are involved in the update; these processors have direct communication (processor r is connected to both processor $r - 1$ and $r + 1$), allowing constant time exchange of information between them. In addition to the field x_j, we add to each processor a field u for message exchange recognition of "active" processors $r - 1$, r and $r + 1$ ($u_r = 1$ and $u_j = 0$ for $j \neq r$), and fields to keep m and n.

It is reasonable to assume that processor j is responsible for any operation on element x_j (in our case j assumes values $r - 1$, r and $r + 1$). It corresponds to the variant of a linear array of processors in which each processor may perform (in constant time) any operation on its local variables or read a datum from one of its neighbors as the only allowed actions. With such assumptions, the instruction $x_{r+1} \leftarrow x_r + 1$ is performed in the following way:

- processor r calculates $x_r + 1$;
- processor r stores calculated value in a variable ch_r which serves for communication exchange;
- processor $r + 1$ reads the value ch_r from processor r and
- processor $r + 1$ stores the received value in its local memory x_{r+1}.

Although the steps are clear, their correct implementation into a program requires special attention; for example, in a statement like **if** $u_r = 1$ **then** $x_{r+1} \leftarrow x_r + 1$ the condition is satisfied only by processor r and processor $r + 1$ does not receive a chance to help processor r in performing the statement. A parallel algorithm using the above assumptions for the case of (m,n)-subsets is given in [35]. The algorithm was rediscovered in [32].

Here, we follow a different approach, which allows simpler algorithms and leaves implementation details to specific (parallel) programming language. We assume a variant of linear array of processors in which each processor may perform (in constant time) any operation on its local variables or read a datum from one of its neighbors, and, in addition, to store a value in any local variable of one of its neighboring processors. With such assumptions, the instruction $x_{r+1} \leftarrow x_r + 1$ is performed in the following way:

- processor r calculates $x_r + 1$;
- processor r stores calculated value in local memory x_{r+1} of processor $r + 1$.

In this variant, for example, the statement **if** $u_r = 1$ **then** $x_{r+1} \leftarrow x_r + 1$ is executed completely by processor r and processor $r + 1$ is treated here as inactive (although it is activated in any implementation of the statement on a real machine). The parallel algorithm can be described as follows. Processor r performs all the work inside the repeat loop, updating also the message field u indicating which processor will be active in the next iteration of the loop.

```
for each processor r (1 ≤ r ≤ m) do in parallel
        read (n); u_r ← 0; x_r ← 0; ter_r ← 0; if r = 1 then {u_r ← 1; x_r ← 1; output x_r};
        repeat
        if u_r = 1 then {
                if x_r < n and r < m then {x_{r+1} ← x_r + 1; u_{r+1} ← 1; u_r ← 0}
                        else if x_r < n then x_r ← x_r + 1
```

$$\textbf{else } \{x_{r-1} \leftarrow x_{r-1} + 1; x_r \leftarrow 0; u_{r-1} \leftarrow 1; u_r \leftarrow 0\};$$

if $x_r > 0$ **then** output x_r;

if $r = 1$ and $a_r = n$ **then** $\{ter_r \leftarrow 1; x_r \leftarrow 0\}$;

if $r > 1$ and $ter_{r-1} > 0$ **then** $ter_r \leftarrow ter_r + 1\}$;

until $ter_r = n - r + 1$.

In the algorithm all processors will terminate simultaneously. The termination is initiated by processor 1 when it recognizes $x_1 = n$. The termination message is communicated in n steps to processor n; during the communication there is no new output by any processor since $x_j = 0$ for all of them.

The subset generating algorithm does not satisfy Property 1 because the update of elements is performed using arithmetic operations on its element values. If subsets are taken from $S = \{1,2,\ldots,n\}$ then the operation $x_r \leftarrow x_r + 1$ clearly gives the next integer from S and thus a "legal" element of a subset. In contrast, if subsets are to be made out of a set $\{a_1, a_2,\ldots, a_n\}$ of arbitrary elements, finding the next possible value for rth element cannot be done by doing arithmetic on it—it should be made available in other way. In the next section, we study a case (permutations) where the update is made without any arithmetic on elements, thus preserving Property 1.

It is an open problem to generate subsets in the set representation and in lexicographic order such that all Properties 1–6 are satisfied.

5. Generating permutations

We begin by reviewing some existing parallel algorithms for generating permutations, and provide a brief assessment of each in light of the mentioned properties. The algorithm in [1] uses Method 1 (division of permutations into groups). In [13], the permutations of at most m out of n elements are generated on a linear array of processors, also equipped with a so-called selector. The algorithm is cost-optimal. However, the permutations are not produced in lexicographic order, each processor needs memory of size $O(m)$—again to produce a full permutation—and the delay between permutations is non-constant (owing to the fact that non-permutations are produced during the generation process). The algorithm of [33] runs on a vector computer. It is not cost optimal and deals with large integers. Furthermore, the delay between generating two permutations is non-constant, each processor requires memory of size $O(n)$, and the order of generation is not lexicographic. Algorithms described in [14,22] satisfy Properties 1 and 4 only. The first permutation generation algorithm in which a given permutation is produced by n processors rather than one only is given in [2]. The algorithm runs on an EREW PRAM. It uses n processors, and a shared memory of size $O(n)$. Each processor produces at every step one element of the current permutation. The algorithm requires $O(\log n)$ time per permutation and, therefore, is not cost-optimal.

An algorithm for the linear array, satisfying Properties 2–6, is given in [30] (the algorithm requires that the elements be decimal numbers, i.e. drawn from the set $\{1,2,\ldots,n\}$, since arithmetic is performed on the actual elements). Also, the algorithm executes in skew form (which means that processor i prints out its element of a given permutation $n - i$ steps after processor n prints its element), and requires a start-up delay. Finally, the order of permutations generated by the algorithm [30] does not possess any well-known order, such

as minimal change order or lexicographic order. The algorithm [29] uses a mesh with $O(n^2)$ processors and, therefore, does not satisfy Properties 2 and 4.

As observed in [1], the other known parallel permutation generation methods ([20] and others cited in [1,2]) are restricted in at least one of the following two ways: they are capable of generating only a subset of all possible permutations; and/or they are not cost-optimal in any sense.

Algorithms for generating permutations satisfying Properties 1–6 are given in [6] (lexicographic order) and [8] (minimal change order, by adjacent interchange). The algorithm [6] uses some sophisticated techniques to achieve a constant worst-case delay between any two permutations and lexicographic order. We present a simple algorithm of [8], based on idea of adjacent transpositions, with the corresponding sequential algorithm being modified (by author of this article) from the one proposed by G. Ehrlich and described in [16]. It is based on the idea of generating the permutations of $\{p_1, p_2, \ldots, p_n\}$ from the permutations of $\{p_1, p_2, \ldots, p_{n-1}\}$ by taking each such permutation and inserting p_n in all n possible positions of it. The nth element sweeps from one end of the $(n-1)$-permutation to the other by a sequence of adjacent swaps, producing a new n-permutation each time. Each time the nth element arrives at one end, a new $(n-1)$-permutation is needed. The $(n-1)$-permutations are produced by placing the $(n-1)$th element at each possible position within an $(n-2)$-permutation. That is, by applying the algorithm recursively to the $n-1$ elements. Details of sequential implementation are given in [38]. Here, we discuss only parallel implementation.

An element is said to be mobile if its direction points to a smaller adjacent neighbor. The first permutation of the set $\{p_1, p_2, \ldots, p_n\}$ is $p_1, p_2, \ldots, p_n$. Assign a direction to every element, denoted by an arrow above the element. Initially all arrows point to the left. Let processor i produces element d_i of given permutation, $1 \leq i \leq n$. How do all processors in steps 2 and 4 (see algorithm below) know the largest mobile element? This is done by propagating this information during the $n-1$ "pulses" that precede each of these steps. One can simply imagine a variable which travels along with p_n and holds at any given time the largest mobile element in $\{p_1, p_2, \ldots, p_{n-1}\}$ encountered so far.

When p_n reaches its destination at the end of step 1 (or step 3), the largest mobile element is known. Unfortunately, it is known only to processor 1 (respectively, processor n). An additional $n-1$ "pulses" are needed to make this information known to the other processors. In order to avoid this, we can imagine a second variable traveling in the direction opposite to p_n and also holding at any given time the largest mobile integer encountered so far. To implement this idea, we let each processor i store two variables:

> $leftmax_i =$ the index of the largest mobile element from the set $\{p_1, p_2, \ldots, p_{n-1}\}$ in processors $1, 2, \ldots, i$.
> $rightmax_i =$ the index of the largest mobile element from the set $\{p_1, p_2, \ldots, p_{n-1}\}$ in processors $i, i+1, \ldots, n$.

These two variables are initialized at the beginning of steps 1 and 3 as follows. Let $d_i = p_k$; then $leftmax_i, rightmax_i = k$, if $d_i < p_n$ and mobile, and $= 0$, otherwise.

They are then updated during the $n-1$ "pulses" of steps 1 and 3, as shown in the following example.

.

leftmax$_i$	0	2	3	4	0	
	←	←	←	←		
d$_i$	p_1	p_2	p_3	p_4	p_5	
rightmax$_i$	0	2	3	4	0	
	Initially					
	↓	↓				
leftmax$_i$	0	2	3	4	0	compare 0 and 2
	←	←	←		←	
d$_i$	p_1	p_2	p_3	p_5	p_4	
rightmax$_i$	0	2	3	4	0	compare 4 and 0
				↑	↑	
	After one pulse					
		↓	↓			
leftmax$_i$	0	2	3	4	0	compare 2 and 3
	←	←		←	←	
d$_i$	p_1	p_2	p_5	p_3	p_4	
rightmax$_i$	0	2	4	4	0	compare 3 and 4 and replace 3 by 4
			↑	↑		
	After 2 pulses					
			↓	↓		
leftmax$_i$	0	2	3	4	0	compare 3 and 4
	←		←	←	←	
d$_i$	p_1	p_5	p_2	p_3	p_4	
rightmax$_i$	0	4	4	4	0	compare 2 and 4 and replace 2 by 4
		↑	↑			
	After 3 pulses					
				↓	↓	
leftmax$_i$	0	2	3	4	4	compare 0 and 4 and replace 0 by 4
		←	←	←	←	
d$_i$	p_5	p_1	p_2	p_3	p_4	
rightmax$_i$	4	4	4	4	0	compare 0 and 4 and replace 0 by 4
	↑	↑				
	After 4 pulses					

Example. Let $n = 5$.

A detailed implementation of the algorithm follows. Note that:

(1) The initial permutation $\{p_1, p_2,\ldots, p_n\}$ is assumed to have been produced before the algorithm starts.
(2) The direction is stored in a variable *arrow*; initially *arrow$_i$* = *left* for $i = 1, 2,\ldots, n$; when two elements are interchanged, we assume that their arrows are also interchanged implicitly.
(3) The algorithm terminates when no mobile element is found; this condition is detected simultaneously by all processors since in this case *leftmax$_i$* = *rightmax$_i$* = 0, for all $1 \le i \le n$.
(4) Two dummy variables $d_0 = d_{n+1} = \infty$ are used.

ALGORITHM.

   Repeat steps 1–4 below **until** termination
Step 1

(1.1) **for** $i = 1$ to n **do** in parallel

    if ($arrow_i = left$ and $d_{i-1} < d_i < p_n$) or

        ($arrow_i = right$ and $p_n > d_i > d_{i+1}$)

    then $leftmax_i \leftarrow k$ {where $d_i = p_k$}

      $rightmax_i \leftarrow k$

    else $leftmax_i \leftarrow 0$

      $rightmax_i \leftarrow 0$

    endif

  endfor

(1.2) **for** $i = 1$ to $n - 1$ **do**

      $d_{n-i} \leftrightarrow d_{n-i+1}$ {new permutation is output}

      $leftmax_{i+1} \leftarrow \max \{leftmax_i, leftmax_{i+1}\}$

      $rightmax_{n-i} \leftarrow \max \{rightmax_{n-i}, rightmax_{n-i+1}\}$

    endfor

Step 2

  for $i = 2$ to n **do** in parallel

    if $max\{leftmax_i, rightmax_i\} < k$ {where $d_i = p_k$}

      then reverse the direction of $arrow_i$

      else if $max \{leftmax_i, rightmax_i\} = k$ {where $d_i = p_k$}

          then if $arrow_i = left$

             then $d_{i-1} \leftrightarrow d_i$

             else $d_i \leftrightarrow d_{i+1}$

             endif

          endif

      endif

  endfor

    {new permutation is output}

Step 3

  (3.1) same as step (1.1)

  (3.2) **for** $i = 1$ to $n - 1$ **do**

      $d_i \leftrightarrow d_{i+1}$ {new permutation is output}

      $leftmax_{i+1} \leftarrow max \{leftmax_i, leftmax_{i+1}\}$

      $rightmax_{n-i} \leftarrow max \{rightmax_{n-i}, rightmax_{n-i+1}\}$

    endfor

Step 4

    for $i = 1$ to $n - 1$ **do** in parallel

      same as body of the loop in step 2

    endfor

    {new permutation is output}.

The algorithm has all the required Properties 1–6 and is very simple. It is an open problem to design parallel algorithms for listing the following cases of the restricted or generalized permutations: permutations of *m* out of *n* elements in lexicographic order, permutations with repetitions, cyclic permutations, alternate permutations, rosary permutations, reflection-free permutations, queens on a chessboard, linear extensions.

6. Generating combinations

In [2,12], an $O(C(m,n)\log m)$ time algorithm for generating m-combinations on an EREW PRAM using Method 2 is described. The algorithm requires $O(\log m)$ time per combination, and, therefore, is not cost-optimal. Ref. [5] improved the algorithm by presenting a cost-optimal algorithm using a weaker model of computation, linear array of processors. The algorithm is concise and is based on sophisticated mathematical arguments.

In this section, we describe a simple algorithm [36] to generate (m,n)-combinations out of the set $\{1,2,\ldots,n\}$ in parallel using Method 2. It follows lexicographic order of combinations and does not satisfy Property 1 (same is valid for algorithms ([ACG, LT]). An algorithm that will satisfy all listed properties will be given in the next section.

Recall the correspondence between (m, n)-combinations and combinations with repetitions of m out of $n - m + 1$ elements (where multiple choice of the same element is possible). Let $x_i = c_i - i + 1$. Then $1 \leq x_1 \leq x_2 \leq \ldots \leq x_m \leq n - m + 1$, and $x_1, x_2, \ldots, x_m$ is a combination with repetitions of m out of $n - m + 1$ elements; we call it $(m, n - m + 1)$-$r\_combination$.

Because of the simple relation, any algorithm that generates $(m, n + m - 1)$-combinations may be used to generate (m, n)-$r\_combinations$, and vice versa. The only difference is in the output. In particular, the output x_i of a (m, n)-$r\_combination$ can be replaced by $x_i + i - 1$ to yield a $(m, n + m - 1)$-combination, while the generating algorithm remains same. This will be exploited in our algorithm, to make the facts used in generating even more apparent.

The well-known sequential algorithm [16] for generating (m, n)-$r\_combinations$ determines the next $r\_combination$ by a backtrack step that finds an element x_t with the greatest possible index t such that $x_t < n$, therefore, increasable. The element x_t is increased by one, and all following elements x_i for $i > t$ become equal to x_t.

The algorithm to generate the (m, n)-$r\_combinations$ uses a linear array of m processors, indexed 1 to m, and m variables $x_1, \ldots, x_m$, where $1 \leq x_i \leq x_2 \leq \ldots \leq x_m \leq n$. Each processor i is responsible for maintaining x_i by reading only data from processors $i - 1$ and $i + 1$. The processors act in lock-step fashion (processors execute always the same instructions using a joint program), and each step produces a new (m, n)-$r\_combination$.

Processors 1 to $m - 1$ will always produce the same element unless they are advised to change their output. Processor m produces elements between x_{m-1} and n. This is called a run of processor m. For convenience, let $x_0 = n$ and $x_{m+1} = n$. Whenever $x_t = n - 1$ and $x_{t-1} < n - 1$ processor t (such an element is called the turning point) initiates a message that informs processors indexed $t + 1, \ldots, m$ that a change in their value is about to happen. Processor i designates registers w_i and s_i for counting waiting time and recording the message, respectively. Each step in the message path corresponds to producing a combination. The message s_i, in fact, contains the value of $x_{t-1} + 1$ since it will become the new value in processors i for $t \leq i \leq m$. The new value of x_i becomes effective $m - i + 2$ steps following the receipt of the message (when w_i reaches 0). In the above table the message path is marked in bold. Whenever $x_{i+1} = n$ but $x_i < n$, x_i increases by one in the next step. Such elements are underlined in the above table. There is enough time for message passing because the message path is $m - t + 1$, which is one less than the length of the current run of processor m; the current run of processor m consists of $m - t + 2$ combinations ending with $x_t \ldots x_m = n - 1, n - 1, \ldots, n - 1, n, \ldots, n$. When $x_1 = n - 1$, the message is also initiated, for the last time, since at the time of update the new combination is the last combination $nn, \ldots, n$, and all processors terminate simultaneously.

The above algorithm can be coded as follows. Each iteration consists of five if statements. If given criteria are satisfied, these statements correspond to decreasing waiting time, forwarding the message, increasing the element preceding n, updating all elements following the turning point, and initiating the message from the turning point, respectively.

For $i \leftarrow 1$ **to** m **do** in parallel {
    $x_i \leftarrow 1$; $x_0 \leftarrow n$; $x_{m+1} \leftarrow n$; $w_i \leftarrow 0$; $s_i \leftarrow 0$;
    Repeat
        output $x_i + i - 1$;
        if $w_i \geq 1$ **then** $w_i \leftarrow w_i - 1$;
        if $s_i = 0$ and $s_{i-1} > 0$ **then** $\{s_i \leftarrow s_{i-1}; w_i \leftarrow m - i + 2\}$;
        if $x_{i+1} = n$ and $x_i < n$ **then** $x_i \leftarrow x_i + 1$;
        if $w_i = 0$ and $s_i > 0$ **then** $\{x_i \leftarrow s_i; s_i \leftarrow 0\}$;
        if $x_i = n - 1$ and $x_{i-1} \neq n - 1$ and $s_i = 0$ **then** $\{s_i \leftarrow x_{i-1} + 1$;
          $w_i \leftarrow m - i + 2\}$
    until $x_i = n + 1$}

The algorithm given above generates all $(m, m + n - 1)$-combinations. It will generate (m, n)-r_combinations if the current output is simply x_i instead of $x_i + i - 1$. The dynamic of variables s_i and w_i are illustrated in table 1 (last four columns).

7. Generating combinations from arbitrary elements

In this section, we describe a combination generation technique [18] that satisfies all desirable criteria (1)–(6), except a "minor" modification to the architecture: processor m is allowed to have memory of size $O(n)$, to keep the data from the set $\{p_1, p_2, \ldots, p_n\}$. Its role in the algorithm will be to supply data from the set $\{p_1, p_2, \ldots, p_n\}$ to other processors. We note that in practice the network models of parallel computation usually have a master processor that distributes the job and/or data to other ones, and has a clock to synchronize the execution of all processors. From that point of view our model is not a restriction with respect to

Table 1. (4,6)-combinations (the first column) and the corresponding (4,3)-r_combinations (the second column).

Combinations	r_combinations	$s_1 w_1$	$s_2 w_2$	$s_3 w_3$	$s_4 w_4$
1 2 3 4	1 1 1 1	00	00	00	00
1 2 3 5	1 1 1 2	00	00	00	22
1 2 3 6	1 1 1 3	00	00	00	21
1 2 4 5	1 1 2 2	00	00	23	00
1 2 4 6	1 1 2 3	00	00	22	22
1 2 5 6	1 1 3 3	00	00	21	21
1 3 4 5	1 2 2 2	00	24	00	00
1 3 4 6	1 2 2 3	00	23	23	00
1 3 5 6	1 2 3 3	00	22	22	22
1 4 5 6	1 3 3 3	00	21	21	21
2 3 4 5	2 2 2 2	45	00	00	00
2 3 4 6	2 2 2 3	44	44	00	00
2 3 5 6	2 2 3 3	43	43	43	00
2 4 5 6	2 3 3 3	42	42	42	42
3 4 5 6	3 3 3 3	41	41	41	41
		00	00	00	00

realistic linear array of processor models. Also, $O(n)$ memory is necessary to store the set $\{p_1, p_2, \ldots, p_n\}$, meaning that the total space used remains optimal.

An (m, n)-combination can be represented as a sequence $c_1, c_2, \ldots, c_m$ where $p_1 \leq c_1 < c_2 < \ldots < c_m \leq p_n$. Let $z_1, z_2, \ldots, z_m$ be the corresponding array of indices, i.e. $c_i = p(z_i)$, $1 \leq i \leq m$, where the notation $y(r) = y_r$ is used to avoid double indices.

Our algorithm to generate the (m, n)-combinations uses a linear array of m processors, indexed 1 to m. Each processor i is responsible for maintaining c_i and z_i by reading only data from processors $i - 1$ and $i + 1$. The processors act in lock-step fashion, and each step produces a new (m, n)-combination.

The well known sequential algorithm for generating (m, n)-combinations determines the next combination by a backtrack search that finds an element c_t with the greatest possible index t such that $z_t < n - m + t$, therefore, increasable (processor t is called the turning point). Note that always $c_i \leq p_{n-m+i}$ for each processor i. The new value of z_i for $i \geq t$ becomes equal to $z_t + i - t + 1$. A straightforward implementation of the backtracking step would result in an occasional $O(m)$ delay on a linear array.

To avoid non-constant delays, we consider two cases of the backtrack search.

Case 1. $z_t = n - m + t - 1$. This is the next to the maximal possible value of the index z_t. Since t is the turning point, $z_{t+1} = n - m + t + 1$, i.e. processor $t + 1$ keeps its maximal value at the moment. The only change in the system is that z_t will increase by one while other elements will not change. This is called a minor change in the system. Minor changes are trivial to implement; in order to keep the constant delay property in this case it is sufficient that each processor i keeps two maximal values $p_{n-m+i-1}$ and p_{n-m+i}. This is sufficient for processor i to recognize itself as a turning point in a minor change and to complete the minor change with constant delay.

Case 2. Otherwise the system is supposed to perform a major change. To achieve a constant delay in this case, we decide to start the backtrack search in advance, such that the new values of all indices z_i and elements c_i are known at the time when "major" changes in the system are due. Note that between two major changes the system undergoes a series of minor ones.

Consider the following table which shows the indices $z_t, z_{t+1}, \ldots, z_m$ of combinations between two major changes. Indices that actually perform minor changes are marked in bold. The turning points for the first and last combinations in the table are processors $t + 1$ and t, respectively (corresponding indices of elements are underlined).

t	$t+1$	$t+2$	$t+3$	...	$m-2$	$m-1$	m	processors
z_t	$\underline{n-m+t-1}$	$n-m+t+2$	$n-m+t+3$	...	$n-2$	$n-1$	n	major change
z_t	$n-m+t$	$n-m+t+1$	$n-m+t+2$	...	$n-3$	$n-2$	$\mathbf{n-1}$	minor change
z_t	$n-m+t$	$n-m+t+1$	$n-m+t+2$	...	$n-3$	$\mathbf{n-2}$	n	minor change
z_t	$n-m+t$	$n-m+t+1$	$n-m+t+2$	...	$\mathbf{n-3}$	$n-1$	n	minor change
...								
...								
z_t	$n-m+t$	$n-m+t+1$	$\mathbf{n-m+t+2}$	...	$n-2$	$n-1$	n	minor change
z_t	$n-m+t$	$\mathbf{n-m+t+1}$	$n-m+t+3$	...	$n-2$	$n-1$	n	minor change
z_t	$\mathbf{n-m+t}$	$n-m+t+2$	$n-m+t+3$	...	$n-2$	$n-1$	n	minor change
z_t	$n-m+t+1$	$n-m+t+2$	$n-m+t+3$	...	$n-2$	$n-1$	n	major change

For the special case $t = m - 1$ the table reduces to the following three rows:.

z_{m-1}	$n - 2$
z_{m-1}	$\boldsymbol{n - 1}$
z_{m-1}	n

There are at least $m - t$ minor changes before a major change with turning point in processor t. In order to perform the major change in the last combination of the above table, we decide to prepare the data starting the process with the second one; more precisely, whenever $z_m = n - 1$ processor m activates the search for the turning point. First, a message is sent towards the turning point to find it; this will take $m - t$ steps. Next, the message is returned back towards processor m, informing all processors between t and m what the turning point processor t and index z_t are. This step will take another $m - t$ steps. Finally, processor m sends the new data for all processors between t and m in a pipelined fashion. The next combination (following the major change in the last combination of the above table) has the following indices:

t	$t + 1$	$t + 2$	$t + 3$	...	$m - 2$	$m - 1$	m	processors
$z_t + 1$	$z_t + 2$	$z_t + 3$	$z_t + 4$	...	$z_t + m - t - 1$	$z_t + m - t$	$z_t + m - t + 1$	indices

The data $p(z_t + 1), \ldots, p(z_t + m - t + 1)$ are broadcast from processor m in a pipelined fashion, using the links of the linear array of processors. The data path is illustrated in the following table.

t	$t + 1$	$t + 2$	$t + 3$	...	$m - 2$	$m - 1$	m processors
							$p(z_t + 1)$
						$p(z_t + 1)$	$p(z_t + 2)$
					$p(z_t + 1)$	$p(z_t + 2)$	$p(z_t + 3)$
			...	...	...	...	...
			$p(z_t + 1)$	...	$p(z_t + m - t - 4)$	$p(z_t + m - t - 3)$	$p(z_t + m - t - 2)$
		$p(z_t + 1)$	$p(z_t + 2)$	...	$p(z_t + m - t - 3)$	$p(z_t + m - t - 2)$	$p(z_t + m - t - 1)$
	$p(z_t + 1)$	$p(z_t + 2)$	$p(z_t + 3)$	...	$p(z_t + m - t - 2)$	$p(z_t + m - t - 1)$	$p(z_t + m - t)$
$p(z_t + 1)$	$p(z_t + 2)$	$p(z_t + 3)$	$p(z_t + 4)$	...	$p(z_t + m - t - 1)$	$p(z_t + m - t)$	$p(z_t + m - t + 1)$

Such a broadcast clearly takes another $m - t$ steps. Therefore, the total number of steps necessary to prepare data for a major change in the system is $3(m - t)$. These are to be done during $m - t$ minor changes. This would be clearly possible if we decide to set the communication speed to (at least) three messages between any two minor changes. Thus, for example, the search message is communicated from processor m to processors $m - 1, m - 2$ and $m - 3$ before the first minor change (in the above table it is the second combination) is done. To avoid precise calculation of the number of message steps vs the number of minor changes, we use a simpler criterion to determine when the new values for z_i and c_i become effective, i.e. when a major change is due: each processor $i (t \leq i \leq m)$ repeats $i - t$ times its maximal index value $n - m + i$. Termination criteria can also be specified in terms of repetitions of maximal index value, combined with an indication from processor 1 to other processors that no turning point is found for a "major" change in the last combination. This analysis proves that Case 2 can be implemented with constant delay between any two

combinations. The algorithm can be coded in the following way.

For $i \leftarrow 1$ **to** m **do** in parallel {
read p_i, p_{n-m+i} and $p_{n-m+i-1}$; $z_i \leftarrow i$; $c_i \leftarrow p_i$; $t_i \leftarrow 0$; $x_i \leftarrow 0$; $v_i \leftarrow 0$; $cn_i \leftarrow 0$;
Repeat
  output c_i;
  if $z_i = n - m + i$ **then** $cn_i \leftarrow cn_i + 1$; (*cn counts repetitions of maximal value in i*)
  if $i = m$ and $z_i = n - 1$ **then** $t_i \leftarrow -1$; (* initiate search for turning point t*)
  for $i \leftarrow 1$ **to** 3 **do** {
      if $i < m$ and $t_i = 0$ and $t_{i+1} = -1$
         then $\{t_i \leftarrow -1$; $t_{i+1} \leftarrow 0$; (*continue search for t*)
           if $z_i < n - m + i - 1$ **then** $\{x_i \leftarrow z_i$; $t_i \leftarrow i\}$ (*turning point t found*)
                   else if $i = 1$ **then** $t_i \leftarrow 2n\}$;
                       (*initiate termination message*)
      if $i > 1$ and $t_i = 0$ and $t_{i-1} > 0$ **then** $\{t_i \leftarrow t_{i-1}$; $x_i \leftarrow x_{i-1}\}$
(*distribute t and z_t to processors t, $t + 1, \ldots, m$; they are saved as t_i and x_i, respectively*)
      if $i = m$ and $t_i > 0$ and $v_i = v_{i-1}$ **then** $\{x_i \leftarrow x_i + 1$; $v_i \leftarrow x_i$; $r_i \leftarrow p(v_i)\}$;
(*generating new values, for next major change, for processors t, $t + 1, \ldots, n$*)
      if $i < m$ and $t_i > 0$ and $v_{i+1} > 0$ and $v_i \neq x_i + i - t_i + 1$ **then** $\{r_i \leftarrow r_{i+1}$;
                                             $v_i \leftarrow v_{i+1}\}\}$;
(*pipeline to the left new values v_i and r_i for z_i and c_i, respectively*)
  if $i < m$ and $z_i = n - m + i - 1$ and $z_{i+1} = n - m + i + 1$ **then** $\{z_i \leftarrow n - m + i$;
                                    $c_i \leftarrow p_{n-m+i}\}$; (*minor change*)
  if $i = m$ and $z_i < n$ **then** $\{z_i \leftarrow z_i + 1$; $c_i \leftarrow p(z_i)\}$ (*minor change in proc. n *)
  if $(cn_i = i - t_i$ and $i > t_i > 0)$ or $(i = t_i$ and $cn_{i+1} = i + 1 - t_{i+1})$
      then $\{z_i \leftarrow v_i$; $c_i \leftarrow r_i$; $t_i \leftarrow 0$; $x_i \leftarrow 0$; $v_i \leftarrow 0$; $cn_i \leftarrow 0\}$ (*major change*)
  until $t_i = 2n$ and $cn_i = i\}$
         }

Using the correspondence between (m, n)-combinations and combinations with repetitions of m out of $n - m + 1$ elements (where multiple choice of the same element is possible), one can design an algorithm for generating combinations with repetitions drawn from an arbitrary set by modifying the algorithm for generating combinations. It is also straightforward to describe an algorithm for generating subsets of a given set $\{p_1, p_2, \ldots, p_n\}$, based on the combination generation technique given in this section. The algorithm in this section assumes that n elements are all stored in one processor. It is an open problem to design an algorithm for generating combinations assuming that processors share data equally, i.e. each of them has memory of size $O(n/m)$. When $n = O(m)$ it is still a constant space and, in this case, an exception made on the size of one processor can be lifted.

8. Conclusions

The survey paper [10] presents a collection of other parallel algorithms for generating and listing combinatorial objects such as combinations, permutations, derangements, integer partitions, compositions, combinations with repetitions, subsets, equivalence relations,

binary trees, well-formed parentheses and variations, using the shared instances method. The algorithms satisfy a number of strict conditions, listed at the beginning of this chapter, and as a result are optimal in every known and reasonable sense. The most salient property of these algorithms, and the unifying theme of this survey, is that they are all designed to run on a very simple model of parallel computation, namely a linear array of processors. In addition, all algorithms from the survey generate combinatorial objects in lexicographic order.

Parallel algorithms with shared instances method, satisfying Properties 2–6, are designed for the following combinatorial objects: binary counters (i.e. integers in base 2, bitstrings, or subsets in the binary representation) [4,10], integer compositions into any number of parts [9], variations of $\{0, 1,\ldots, m-1\}$, i.e. counters or integers in base m [4], variations, satisfying Property 1 [10], combinations (directly, without using their relation to combinations with repetitions) [5], integer composition of n into exactly m parts [9], integer compositions [26], permutations, satisfying Property 1 [6], derangements, satisfying Property 1 [3], equivalence relations [35], binary tree sequences using the parent array representation [7], t-ary tree sequences using the parent array representation [7], binary tree sequences using the children array representation [10], t-ary tree sequences using the children array representation, and well-formed parentheses sequences [10,44].

A parallel algorithm for generating variations out of the set $\{0,1,\ldots,n-1\}$ in a Gray code order with constant delay on a linear array with reconfigurable bus system is designed in [41]. A parallel algorithm for generating variations out of the set $\{0,1,\ldots,n-1\}$ in a Gray code order with constant delay on a linear array of processors is designed in [39]. It can be generalized such that Properties 1–6 are satisfied.

Recently, some new sequential $O(1)$ time algorithms were developed for generating derangements [27], Schroeder trees [28], combinations and well-formed parenthesis strings [46] were described. They can obviously also be treated as a kind of parallel algorithms.

There are number of open problems left for further study. In [9], two parallel algorithms that generate partitions on a linear array of $n(n^{1/2})$ processors in the standard (multiplicity, respectively) representations are given, and they run with constant delay. The costs of the algorithms (product of the number of processors used and time complexity) are $O(nP(n))$ and $O(n^{1/2}P(n))$ (respectively), which may not be cost-optimal. How to design a cost-optimal parallel algorithm for generating integer partitions, using shared instances method? How to design an algorithm for generating subsets in the set representation that will satisfy Properties 1–6 and list subsets in lexicographic order? How to design an algorithm for generating set partitions that will satisfy Properties 1–6 (in lexicographic or other order)? The derangement generation algorithm [3] is made adaptive by dividing corresponding permutations (not derangements) into k groups and showing that each group does at most three times more work than necessary. On the other hand, some groups may do very little useful work (for example, the first group produces no derangement at all when $k > n$). How to design a more balanced adaptive algorithm for generating derangements, without using large integers for subdivision (like unranking algorithm [19])? How to design a parallel algorithm for generating integer partitions which will satisfy the listed Properties 2–6 (includes cost-optimality and constant delay)? Parallel algorithms for generating AVL trees, 2–3 trees, B-trees, non-regular trees, unordered trees, free trees, trees with n nodes and m leaves, trees with nodes of any degree, and trees with bounded height may be also designed. Finally, a constant delay algorithm for generating variations in a Gray code order using a linear array of processors needs to be designed.

References

[1] Akl, S.G., 1987, Adaptive and optimal parallel algorithms for enumerating permutations and combinations, *The Computer Journal*, **30**(5), 433–436.

[2] Akl, S.G., 1989, *The Design and Analysis of Parallel Algorithms* (Englewood Cliffs, NJ: Prentice-Hall).

[3] Akl, S.G., Calvert, J.M. and Stojmenović, I., 1992, Systolic generation of derangements. In: P. Quinton and Y. Robert (Eds.) *Algorithms and Parallel VLSI Architectures II* (Amsterdam: Elsevier Science Publication), pp. 59–70.

[4] Akl, S.G., Duboux, T. and Stojmenović, I., 1991, Constant delay parallel counters, *Parallel Processing Letters*, **1**(2), 143–148.

[5] Akl, S.G., Gries, D. and Stojmenović, I., 1989, An optimal parallel algorithm for generating combinations, *Information Processing Letters*, **33**, 135–139.

[6] Akl, S.G., Meijer, H. and Stojmenović, I., 1994, An optimal systolic algorithm for generating permutations in lexicographic order, *Journal of Parallel and Distributed Computing*, **20**(1), 84–91.

[7] Akl, S.G. and Stojmenović, I., 1996, Generating *t*-ary trees in parallel, *Nordic Journal of Computing*, **3**, 63–71.

[8] Akl, S.G. and Stojmenović, I., 1992, A simple optimal systolic algorithm for generating permutations, *Parallel Processing Letters*, **2**(2/3), 231–239.

[9] Akl, S.G. and Stojmenović, I., 1993, Parallel algorithms for generating integer partitions and compositions, *The Journal of Combinatorial Mathematics and Combinatorial Computing*, **13**, 107–120.

[10] Akl, S.G. and Stojmenović, I., 1996, Generating combinatorial objects on a linear array of processors. In: A.Y. Zomaya (Ed.) *Parallel Computing: Paradigms and Applications* (London: International Thomson Computer Press), pp. 639–670.

[11] Baril, J.L. and Vajnovszki, V., 2004, Gray code for derangements, *Discrete Applied Mathematics*, **140**(1–3), 207–221.

[12] Chan, B. and Akl, S.G., 1986, Generating combinations in parallel, *BIT*, **26**(1), 2–6.

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

[14] Cosnard, M. and Ferreira, A.G., 1989, Generating permutations on a VLSI suitable linear network, *The Computer Journal*, **32**(6), 571–573.

[15] Djokić, B., Miyakawa, M., Sekiguchi, S., Semba, I. and Stojmenović, I., 1990, Parallel algorithms for generating subsets and set partitions. In: T. Asano, T. Ibaraki, H. Imai and T. Nishizeki (Eds.) *Proceedings of SIGAL International Symposium on Algorithms*, Tokyo, Japan, Lecture Notes in Computer Science, Vol. 450, pp. 76–85.

[16] Even, S., 1973, *Algorithmic Combinatorics* (New York: Macmillan).

[17] Er, M.C., 1988, A parallel algorithm for cost-optimal generation of permutations of *r* out of *n* items, *Journal of Information & Optimization Sciences*, **9**, 53–56.

[18] Elhage, H. and Stojmenović, I., 1992, Systolic generation of combinations from arbitrary elements, *Parallel Processing Letters*, **2**(2/3), 241–248.

[19] Gupta, P. and Bhattacharjee, G.P., 1989, A parallel derangement generation algorithm, *BIT*, **29**, 14–22.

[20] Gupta, P. and Bhattacharjee, G.P., 1983, Parallel generation of permutations, *The Computer Journal*, **26**(2), 97–105.

[21] Kapralski, A., 1993, New methods for the generation of permutations, combinations, and other combinatorial objects in parallel, *Journal of Parallel and Distributed Computing*, **17**, 315–326.

[22] Kokosinski, Z., 1990, On generation of permutations through decomposition of symmetric groups into cosets, *BIT*, **30**(4), 583–591.

[23] Kokosinski, Z., 1999, On parallel generation of set partitions in associative processor architectures. *Proceedings of Fifth International Conference "Parallel and Distributed Processing Techniques and Applications" PDPTA'99*, Las Vegas, USA, CSREA, Vol. III, pp. 1257–1262.

[24] Kokosinski, Z., 2001, On parallel generation of *t*-ary trees in an associative model. *Parallel Processing and Applied Mathematics: Fourth International Conference, PPAM 2001 Na: czów*, Poland, September 9–12, Vol. 2328/2002, p. 228.

[25] Kokosinski, Z., 2003, A parallel dynamic programming algorithm for unranking *t*-ary trees, LNCS 3019/2004. *Parallel Processing and Applied Mathematics: Fifth International Conference, PPAM 2003*, Czestochowa, Poland, September 7–10 , pp. 255–260.

[26] Kokosinski, Z., 1996, Generation of integer compositions on a linear array of processors. *Proceedings of Second International Conference "Parallel and Distributed Processing Techniques and Applications" PDPTA'96*, pp. 56–64.

[27] Korsh, J.F. and LaFolette, P., 2004, Constant time generation derangements, *Information Processing Letters*, **90**(4), 181–186.

[28] Korsh, J.F. and LaFolette, P., 2003, Loopless generation of Schroeder trees, *The Computer Journal*, **46**(1), 106–113.

[29] Lee, W.P. and Tsay, J.C., 1994, A systolic design for generating permutations in lexicographic order, *Parallel Computing*, **20**, 775–785.

[30] Lin, C.J., 1990, Parallel algorithm for generating permutations on linear array, *Information Processing Letters*, **33**, 167–170 Parallel Computing, 15, 1, 1990, 267–276; *International Journal of Computer Mathematics*, **38**, 1991, 113–121.

[31] Lin, C.J. and Tsay, J.C., 1989, A systolic generation of combinations, *BIT*, **29**, 23–36; Computers and Mathematics with Applications, 17, 12, 1989, 1523–1533; *Parallel Computing*, **13**, 1990, 119–125.

[32] Lee, W.P., Tsay, J.C., Chen, H.S. and Tseng, T.J., 1997, An optimal systolic algorithm for set partitioning problem, *Parallel Algorithms and Applications*, **10**, 301–313.

[33] Mor, M. and Fraenkel, A.S., 1982, Permutation generation on vector processors, *The Computer Journal*, **25**(4), 423–428.

[34] Shen, H. and Evans, J., 1996, A new method for generating integer compositions in parallel, *Parallel Algorithms and Applications*, **9**, 101–109.

[35] Stojmenovic, I., 1990, An optimal algorithm for generating equivalence relations on a linear array of processors, *BIT*, **30**(3), 424–436.

[36] Stojmenović, I., 1992, A simple systolic algorithm for generating combinations in lexicographic order, *Computers & Mathematics with Applications*, **24**(4), 61–64.

[37] Stojmenović, I., 1992, On random and adaptive parallel generation of combinatorial objects, *International Journal of Computer Mathematics*, **42**, 125–135.

[38] Stojmenović, I., Listing combinatorial objects, manuscript.

[39] Stojmenovic, I., 1996, Generating n-ary reflected Gray codes on a linear array of processors, *Parallel Processing Letters*, **6**(1), 27–34.

[40] Tsay, J.C. and Lee, W.P., 1994, An optimal parallel algorithm for generating permutations in minimal change order, *Parallel Computing*, **20**, 353–361.

[41] Thangavel, P. and Muthuswamy, V.P., 1993, A parallel algorithm to generate N-ary reflected Gray codes in a linear array with reconfigurable bus system, *Parallel Processing Letters*, **3**(2), 157–164.

[42] Vajnovszki, V. and Phillips, C., 1996, Two optimal parallel algorithms for generating P-sequences. *Proceedings of ISCA International Conference on Parallel and Distributed Computing Systems*, Dijon, France, September, pp. 819–822.

[43] Vajnovszki, V. and Phillips, C., 1999, Systolic generation of k-ary trees, *Parallel Processing Letters*, **9**(1), 93–102.

[44] Vajnovszki, V., 1998, Parallel algorithms for listing well-formed parentheses strings, *Parallel Processing Letters*, **8**(1), 19–28.

[45] Wu, B.Y. and Tang, C.Y., 1991, Ranking, unranking and parallel enumerating of topological orders, *International Conference on Parallel Processing*, III, 284–285.

[46] Xiang, L. and Ushijima, K., 2001, On O(1) time algorithm for combinatorial generation, *The Computer Journal*, **44**(4), 292–302.