THE PARALLEL GENERATION OF COMBINATIONS

by

Hassan Elhage

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada
November 15, 1995

ISBN 0-612-11553-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

In this thesis we consider the problem of the generation of combinations of m elements chosen from a set of n elements. We provide a survey of the sequential and parallel algorithms for the generation of the (m,n)-combinations.

The major achievement of this thesis is a parallel algorithm for generating all combinations *without repetitions* of m elements from an arbitrary set of n elements in lexicographic ascending order. The algorithm uses a linear systolic array of m processors, each having constant size memory (except processor m, which has $O(n)$ memory), and each being responsible for producing one element of a given combination. There is a constant delay between successive combinations, leading to an $O(C(m,n))$ time solutions, where $C(m,n)$ is the total number of (m,n)-combinations. The algorithm is cost-optimal, assuming the time to output the combination is counted, and does not deal with very large integers. This algorithm is an improvement over all previous works because it generates combinations of a set of arbitrary elements $\{p_1, p_2, \ldots, p_n\}$, on which an order relation $<$ is defined such that $p_1 < p_2 < \ldots < p_n$. This property is important since it allows us to distinguish between algorithms that generate combinations of any ordered set, and algorithms that can only generate combinations of the set $\{1, 2, \ldots, n\}$. Last, we modify this algorithm to generate combinations *with repetitions* in lexicographic order chosen from a set of arbitrary objects.

Acknowledgments

I wish to acknowledge several persons who directly or indirectly contributed to this thesis. First, I would like to thank my supervisor, Professor Ivan Stojmenovic. Professor Stojmenovic has provided me outstanding guidance and assistance throughout the course of my research. His course in parallel algorithms, which I took during my first semester in my Master program at Ottawa University, stimulated enormously my interest in the area of combinatorial algorithm in general and the area of parallel algorithms in particular. He proposed my thesis topic, was available to answer any question related the thesis, and provided me with all necessary feedback. Professor Stojmenovic had great patience with me and understood to the fact that I have a full time job while I am pursuing my graduate studies, which was not an easy task.

I would like to thank the Department of Computer Science at Ottawa University for giving me the opportunity to be enrolled in the graduate program and conduct my research. In particular, I would like to thank all faculty members and staff for their help and support.

Finally, I would like to thank my brothers, special friends, and colleagues for their encouragement to complete my Master program while I am occupying a full time job at the same time.

Contents

Tables

Table **Page**

Figures

Algorithms

Algorithm **Page**

Chapter I

Introduction

1.1 General

The field of combinatorial algorithms deals with the problem of how to carry out computations on discrete mathematical structures. It is a relatively new field, and only on the last thirty years has it started to evolve as a systematic body of knowledge about the design, implementation, and analysis of algorithms. The factors that contributed to its emergence as an important new discipline may include the following [12]:

- The increase in the practical importance of computations of combinatorial nature in the science and engineering field.
- The rapid progress in the design and analysis of algorithms, primarily of a mathematical nature.
- The shift in emphasis from the examination of properties shared by a class of combinatorial algorithms instead of the properties of particular algorithms.

The combinatorial problems are generally defined over a finite set of discrete elements, and involve counting, selecting, or ordering elements. The sets and sequences that are generated in these problems are called *combinatorial objects*. A number of combinatorial objects have been defined. The evolution of algorithm analysis has led to the development of a number of

sequential and parallel algorithms to generate these combinatorial objects. In this thesis we are interested in the problem of generating all combinations of m elements out of the set of n elements in parallel. A parallel computer is a computer with several processors. A parallel algorithm is a solution method for a given problem that will be performed on a parallel computer. Only during the last twenty years has parallelism become an attractive and viable approach to the achievement of very high computational speed [19].

1.2 Combinatorial objects

A combinatorial object is a set of sequences, arranged in a particular order, where each element of a sequence is selected from a set of n elements $S = (s_1, s_2, \ldots, s_n)$. There are usually many instances (by instance we mean one sequence) of a combinatorial object. The generation of a certain combinatorial object means producing as output all distinct instances of this object. There exist approximately a dozen combinatorial objects of general interest: combinations, permutations, derangements, integer partitions, compositions, subsets, equivalence relations, variations, t-ary trees, and well-formed parentheses. Even though the thesis focuses on the generation of combinations, a brief description for few other combinatorial objects will be also provided in the following.

1) ***Subsets.*** A subset of a set $S = \{s_1, s_2, \ldots, s_n\}$ is a set of elements, S', all of which belong to S, $S' \subseteq S$. Every subset $\{x_1, x_2, \ldots, x_r\}$ can be represented by a sequence $x_1\ x_2\ \ldots\ x_r$, $1 \le r \le n$, $s_1 \le x_1 < x_2 < \ldots < x_r \le s_n$. For example, $\{s_1, s_3, s_5\}$, or simply $s_1\ s_3\ s_5$, is a subset of the set $\{s_1, s_2, s_3, s_4, s_5, s_6\}$. The total number of subsets of an n-element set is 2^n [20].

2) ***Permutations.*** An n-permutation of $S = \{s_1, s_2, \ldots, s_n\}$ is an arrangement of n elements into n positions. For example, $s_1s_4s_3s_5s_2$ and $s_4s_1s_2s_5s_3$ are two different permutations for $n = 5$. The order of the objects is important. The number of choices for the first element is n, the number of choices for the second element is $(n-1)$, etc. In general, there are $n(n-1)\ldots1 = n!$ distinct permutations of n elements.

An (m,n)-permutation is an arrangement of m elements out the set of n elements S. For example, $s_2s_3s_5$ and $s_4s_1s_2$ are two $(3,5)$-permutations. The total number of (m,n)-permutations $P(m,n)$ is given as by:

$$P(m,n) = \frac{n!}{(n-m)!}$$

where $k! = 1x2x3x\ldots xk$. When $m = n$, we obtain the n-permutations.

3) ***Derangements.*** A derangement is an arrangement of n elements into n positions, such that element s_i does not appear in (its identity) position i, for all i, $1 \leq i \leq n$. For example, $s_2s_1s_4s_5s_3$ and $s_2s_1s_5s_3s_4$ are two derangements for $n = 5$. The number of derangements of n, D_n, is given for $n \geq 2$ by the following recurrence relation [20]:

$$D_n = (n-1)(D_{n-1} + D_{n-2})$$

where $D_0 = 1$ and $D_1 = 0$.

4) ***Compositions.*** A composition of an integer n is a sequence of integers, $c_1c_2\ldots c_k$, where $c_i > 0$, $1 \leq i \leq k$, for some k, $1 \leq k \leq n$, such that:

$$n = \sum_{i=1}^{k} c_i$$

For example, *2 2 1 3*, *2 3 1 2*, and *1 5 2* are *3* different compositions of *8*. The total number of compositions of n is 2^{n-1}. The elements of a composition can only be integers.

5) ***Integer partitions.*** A partition of an integer n is given by a sequence of integers, $p_1 p_2 \ldots p_k$, where $p_i \geq 0$, $1 \leq i \leq k$, and $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_k$ such that:

$$n = \sum_{i=1}^{k} p_i$$

For example, *4 3 3 1 1* is a partition of *12*. The elements of a partition can only be integers. A closed form formula for the total number of partitions of an integer does not exist. However, this number is the coefficient of the x^n term in the following polynomial [18, 20]:

$$p(x) = (1 - x^1)^{-1}(1 - x^2)^{-1} \ldots .$$

Also, the number of partitions $p(n)$ of n can be determined using the following recurrence relation [18]:

$$p(n,k) = p(n-k,k) + p(n,k-1)$$

where $n \geq k \geq 1$ and $p(n,k)$ is the number of partitions of n such that the largest part x_1 is no longer than k.

6) *Equivalence Relations (Set partitions).* An equivalence relation of the set $Z = \{1, 2, \ldots, n\}$ consists of classes $Z_1, Z_2, \ldots, Z_k$ such that the $Z_i \subset Z$, $1 \le i \le k$, $Z_i \cap Z_j = \emptyset$, if $i \ne j$ and

$$Z = \bigcup_{i=1}^{k} Z_i$$

For example, $\{\{1, 3\}, \{2\}, \{4, 5, 6\}\}$ is a partition of the set $\{1, 2, \ldots, 6\}$. The total number of partitions of an n element set is given by the following Bell number [20]:

$$B_n = \sum_{k=0}^{n-1} C(n-1,k) B_k$$

where $B_0 = 1$.

7) ***Variations.*** An m-variation of a set of n elements S is a string $v_1 v_2 \ldots v_m$ such that $v_i \in S$, for all i, $1 \le i \le m$. Note that repeated elements are allowed. For example, *aaaba* and *abcdd* are both 5-variations of the set $\{a, b, c, d\}$. Special instances of m-variations are binary and decimal counters, where $S = \{0, 1\}$ and $S = \{0, 1, \ldots, 9\}$, respectively. The total number of variations of m elements out of n is $V(m,n) = n^m$.

8) ***Parentheses sequences.*** A parentheses sequence is a *well-formed* sequence of n left and n right parentheses such that, when scanning from left to right, the number of right parentheses encountered never exceeds the number of left parentheses encountered. For example *(( )(( )))* and *( )(( )( ))* are two parentheses sequences with $n = 8$. These sequences are in one-to-one correspondence with binary trees on n nodes. The total number of parentheses sequences is given by [18, 20]:

$$\frac{(2n)!}{n!(n+1)!}$$

1.3 Combinations

A combination of m elements out of the set of n elements $S = \{s_1, s_2, \ldots, s_n\}$, called *(m,n)*-combinations, is obtained by selecting m distinct elements of S. In other words, each *(m,n)*-combination is a subset of S containing exactly m elements. When a linear order is specified on the elements of S, by convention, it is desirable to list the elements of the *(m,n)*-combination in increasing order. For example, when $m=4$ and $n=6$, all possible *(4,6)*-combinations of $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ are listed in lexicographic order in Table 1, which are referred as combinations *without repetitions*.

Table 1. Combinations without repetitions in lexicographic order, $n = 6$ and $m = 4$.

$s_1\ s_2\ s_3\ s_4$	$s_1\ s_3\ s_5\ s_6$
$s_1\ s_2\ s_3\ s_5$	$s_1\ s_4\ s_5\ s_6$
$s_1\ s_2\ s_3\ s_6$	$s_2\ s_3\ s_4\ s_5$
$s_1\ s_2\ s_4\ s_5$	$s_2\ s_3\ s_4\ s_6$
$s_1\ s_2\ s_4\ s_6$	$s_2\ s_3\ s_5\ s_6$
$s_1\ s_2\ s_5\ s_6$	$s_2\ s_4\ s_5\ s_6$
$s_1\ s_3\ s_4\ s_5$	$s_3\ s_4\ s_5\ s_6$
$s_1\ s_3\ s_4\ s_6$	

The total number of distinct *(m,n)*-combination of m elements, denoted by *C(m,n)*, is given by:

$$C(m,n) = \frac{n!}{m!(n-m)!}$$

where $k! = 1x2x3x...xk$.

This formula can be derived by using $P(m,n)$ as follows. All (m,n)-permutations can be arranged by first picking any (m,n)-combination and then arranging these m elements in any order. Thus $P(m,n) = C(m,n).P(m,m)$, and solving for $C(m,n)$ we have

$$C(m,n) = \frac{P(m,n)}{P(m,m)} = \frac{n!/(n-m)!}{m!} = \frac{n!}{m!(n-m)!}.$$

The number $C(m,n)$ is frequently called the ***binomial coefficient***.

1.3.1 Representation of combinations

An instance of a combinatorial object can be represented in several different ways. The most important representations that are applicable and widely used to generate ***combinations*** are given next [18, 20]:

1) ***Set representation.*** This notation is the most widely used. Each element of a given set S is used explicitly to represent an instance of the combinatorial object. For example, $a\ b\ c\ d$ is a $(4,6)$-combination of the set of n elements $S = \{a, b, c, d, e, f\}$.

2) ***Binary representation.*** This notation is not appropriate notation for combinatorial objects where element order is important. Each element of a given set is associated with a bit j in a string. The bit is set to 1 if the element is present in an instance of the combinatorial object, otherwise the bit is set to 0. For example, $0\ 1\ 1\ 1\ 0\ 1$ is a $(4,6)$-combination ($b\ c\ d\ f$ in the set representation) of the set $S = \{a, b, c, d, e, f\}$.

1.3.2 Estimates on the cardinalities

It is assumed that a computer memory is capable of storing an integer of size $O(n)$, where n is the size of a combinatorial object under consideration. The memory representation of integer n requires $O(\log n)$ bits which is bounded by the word size of a computer. Also, it is assumed that arithmetic operations on numbers of size $O(n)$ are performed in constant time. In the case of (m,n)-combinations, $C(m,n) = n(n-1)(n-2)\ldots(n-m+1)/m! < n^m$. However, for $m \leq n/2$ we have $C(m,n) = (n/m)(n-1/m-1)\ldots(n-m+1)/1 > 2 \times 2 \times 2 \ldots \times 2 = 2^m$ because $(n-i)/(m-i) > 2$ for $n > 2m$. Therefore $C(m,n) = O(n^m)$ which is polynomial for small values of m. For $m = O(1)$, constant number of memory locations will be sufficient to store $C(m,n)$. On the other hand, for $m = O(n)$, the number $C(m,n)$ is exponential in n and its storage may require $O(n)$ registers. A number that may need $O(n)$ memory for storage, such as $n!$ or 2^n, will be referred to as a ***large integer***.

1.4 Combinations with repetitions

Consider m types of elements, with an unlimited supply of each element. A combination with repetitions allowed is then a selection of a number of elements, m_1, of type *1*, a number of elements, m_2, of type *2*, and in general, a number of elements, m_i, of type i, such that $m = m_1+m_2+\ldots+m_n$. It is clear in this case that there is no ordering of the elements in the selection. As in the case of combinations without repetitions, generating combinations of m elements with repetition allowed sequentially from a set of n ordered elements in lexicographic order involves the generation of each combination from its immediate predecessor. This process to generate the next combination without repetitions is

also applicable for generating combinations with repetitions. We call these combinations *(m,n)-r_*combinations. The number of *(m,n)-r_*combinations, out of n types, such that at least one item is selected out of each type is $C(m-1,n-1)$. The number of *(m,n)-r_*combinations, out of n types, is $C(m,n+m-1)$. For example, consider the set $\{1, 2, \ldots, n\}$, i.e. $n = 6$, and $m = 4$. In this case, there are *126 (4,6)-r_*combinations in lexicographic order with repetitions allowed. Some of these *(4,6)-r_*combinations are presented in Table 2.

Table 2. Combinations with repetitions, $n=6$ and $m=4$.

1 1 1 1	1 1 3 3	1 2 2 3	1 2 5 5
1 1 1 2	1 1 3 4	1 2 2 4	1 2 5 6
1 1 1 3	1 1 3 5	1 2 2 5	1 2 6 6
1 1 1 4	1 1 3 6	1 2 2 6	1 3 3 3
1 1 1 5	1 1 4 4	1 2 3 3	1 3 3 4
1 1 1 6	1 1 4 5	1 2 3 4	1 3 3 5
1 1 2 2	1 1 4 6	1 2 3 5	1 3 3 6
1 1 2 3	1 1 5 5	1 2 3 6	1 3 4 4
1 1 2 4	1 1 5 6	1 2 4 4	. . . .
1 1 2 5	1 1 6 6	1 2 4 5	. . . .
1 1 2 6	1 2 2 2	1 2 4 6	6 6 6 6

1.5 Orderings on combinatorial objects

There exist a number of different orders to list instances of a combinatorial object. The most important orders that are used to list ***combinations*** are presented below [18,20].

1) ***Lexicographic order.*** Let $x_1\, x_2 \ldots x_m$ and $y_1\, y_2 \ldots y_m$ be two *(m,n)*-combinations of the set of n elements S. We say that $x_1\, x_2 \ldots x_m$ precedes $y_1\, y_2 \ldots y_m$ in lexicographic order if there exists an i, $1 \le i \le m$, such that $x_j = y_j$ for all $j < i$ and $x_i < y_i$. The *(m,n)-*

combinations can be produced in either ***lexicographic ascending*** order or ***lexicographic descending*** order. For example, *a b c d, a b c e, a b c f, a b d e, ..., b c d f, c d e f* are *(4,6)*-combinations in lexicographic ascending order whereas *c d e f, b c d f, ..., a b d e, a b c f, a b c e, a b c d* are *(4,6)*-combinations in lexicographic descending order.

2) ***Reverse Lexicographic order.*** One combination $x_1\ x_2 \ldots\ x_m$ is said to precede another $y_1\ y_2 \ldots\ y_m$ in reverse lexicographic order if there exists an i, $1 \leq i \leq m$, such that $x_j = y_j$, for all $j > i$ and $x_i > y_i$. For example, *f c b a* precedes *d c b a* in reverse lexicographic order.

3) ***Adjacent exchange order.*** Combinations are listed in a way that successive combinations differ by the exchange of two adjacent elements. For example, *c f e a b d* and *c f a e b d* differ by one pairwise exchange.

4) ***Gray code order.*** All n-bit string contain m *1*-bits so that successive strings differ in exactly two bits one changes from zero to one another from one to zero. When new combinations are generated with the least possible changes (by a single insertion of an element, single deletion or single replacement of one element by another, interchange of two elements, updating two elements only etc.) corresponding sequences of all instances of a combination are referred to as ***Gray codes***. In addition, the same property must be preserved when going from the last to the first combination. In most cases, there is no difference between minimal change and Gray code orders. For example, *1 1 1 1 0 0* precedes *1 1 1 0 1 0* in minimal change order and Gray code order.

1.6 Models of computation

Computers operate by executing instructions on data. Algorithms are constructed by a stream of instructions that tells the computer what to do at each step. The input to the algorithms is made of a stream of data that is manipulated by the instructions. A model of computation is an abstract definition of a computer, specifying the number and type of processors, the size and type of memory, and the connections between processors and from processors to memory. The most popular computer models that will be considered in this thesis are the following [19]:

1) Sequential computers.

2) Parallel computers with Single Instruction stream, Multiple Data stream (SIMD).

1.6.1 Sequential computers

This computer model consists of a single processor that receives a single stream of instructions that operate on a single stream of data. At each step of the computation process, the processor receives an instruction from the control unit and executes it on a datum obtained from the memory. The processor is connected to a random access memory with constant time access to any memory cell. This computer model is referred as the Random Access Machine (RAM). This class of computers is said to follow the *Von Neuman* architecture and represents the sequential models of computation which use sequential algorithms. Most existing computers, like PC, Mac, etc., belong to this model.

1.6.2 Parallel computers

We are primarily concerned with the model where all processors execute the same instruction but have their own data, denoted as SIMD. This computer model consists of n identical processors that receive a single instruction stream from one central control unit. Each of the n processors possesses its own local memory where it can store both programs and data. There are n data streams, one per processor. In this model all the processors operate synchronously. This means, at each step, all processors execute the same instruction, each on a different datum. The instruction could be a simple one (such as adding two numbers) or a complex one (such as merging two lists of numbers). Similarly, the datum may be simple (one number) or complex (several numbers).

In most problems that are solved on an SIMD computer, during the computation it is desirable for the processors to be able to communicate between each other in order to exchange data or intermediate results. The communication among the processors may be achieved in two ways, giving rise to two sub-models: SIMD computers where communication is through a *shared memory* and those where it is done via an *interconnection network* [19].

1.6.2.1 Shared-Memory (SM) SIMD Model

This computation model is also known in the literature as the Parallel Random-Access Machine (PRAM) model [19]. In this case, the n processors share a common memory. When two processors wish to communicate, they do so through the shared memory. When

processor i wishes to pass a number to processor j, two steps will be performed. First, processor i writes the number in the shared memory at a given location known to processor j. Then processor j reads the number from that location. During computation, the n processors gain access to the shared memory for reading input data, for reading or writing intermediate results, and for writing final results. Based on whether two or more processors can gain access to the same memory location at the same time, the following three subclasses of the SM SIMD model may be considered:

1) **Exclusive-Read, Exclusive-Write (EREW) SM SIMD model.** No two processors can read from or write into the same memory location simultaneously.

2) **Concurrent-Read, Exclusive-Write (CREW) SM SIMD model.** Multiple processors are allowed to read from the same memory location but no two processors are allowed to write into the same location simultaneously.

3) **Concurrent-Read, Concurrent-Write (CRCW) SM SIMD model.** Both multiple-read and multiple-write operations are allowed.

No problems should be encountered during *multiple-read* accesses to the same address in memory since each of the several processors reading from that location makes a copy of the content of that memory location and stores it in its own local memory. However, difficulties arise during *multiple-write* accesses. When several processors attempt to write simultaneously to the same memory location, which of the processors should succeed? Several methods, which we will not describe in this thesis, are available to resolve this write conflict.

1.6.2.2 Interconnection-Network (IN) SIMD Model

In most applications, a small subset of all pairwise connections is usually sufficient to obtain a good performance. The most popular simple network for the SIMD model is the ***Linear Array*** model. In this case, processor p_i is linked to its two neighbors p_{i-1} and p_{i+1} through a two-way communication line. Each of the end processors, p_1 and p_n has only one neighbor. This model is shown for $n = 5$ in Figure 1.

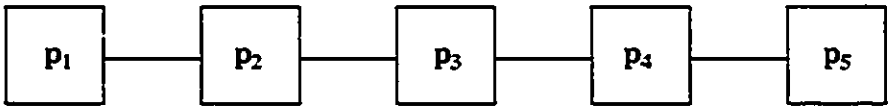

Figure 1. Linear array connection.

1.7 Research problem

Many parallel combination generation algorithms were developed in the past fifteen years. Each of these algorithms satisfies different properties out of the desirable properties presented in chapter III. The parallel combination algorithms presented in [27-30] satisfy the following properties: **1)** generate combinations of $\{1, 2, \ldots, n\}$ in lexicographic order, **2)** they are cost optimal, **3)** the time required between two consecutive combinations is constant, **4)** the model of computation is as simple as possible, and **5)** each processor needs as little memory as possible. The algorithms in [27-30] are the simplest ones; they run in

time $O(C(m,n))$ on an m processor array and are adaptive. However, none of the algorithms in [27-30] satisfies the property of generating combinations of a set of arbitrary elements $\{p_1, p_2, \ldots, p_n\}$, on which an order relation $<$ is defined such that $p_1 <_1 p_2 <_2 \ldots <_{n-1} p_n$. This property becomes important because it allows us to distinguish algorithms that can only generate combinations of the integer set $\{1, 2, \ldots, n\}$ and algorithms that generate combinations of any ordered set. Therefore, the purpose of this thesis is to develop two *cost optimal* parallel algorithms to generate from a set of arbitrary objects: 1) combinations *without repetitions* and 2) combinations *with repetitions* allowed. This problem is a trivial one in the case of sequential combination generation algorithms. However, is not trivial in the case of parallel combination algorithms, because the processes have constant memory and do not keep all the elements of the set $\{a_1, a_2, \ldots, a_n\}$ in their memory; they keep only a constant number of them.

1.8 Contributions

The important and original contributions that are incorporated in this thesis are presented as follows:

1. The most important sequential combination generation algorithms, which generate combinations from the set $\{1, 2, \ldots, n\}$, are described, implemented, measured and compared.

2. A comprehensive survey of all existing parallel combinations generation algorithms is presented. These algorithms are evaluated with respect to the desirable properties and compared in a tabular form.

3. We provide a new parallel combinations generation algorithm for generating all combinations ***without repetitions*** of m elements chosen from an arbitrary set of n elements. This is an improvement over all previous results, since all previous algorithms simply generate combinations from the set of n integers $\{1, 2, \ldots, n\}$. The algorithm satisfies all the desirable properties presented in Chapter III. Specifically, the algorithm is cost optimal, lexicographic, executes on a linear array of m processors, needs only constant size memory per processor, needs constant delay between successive combinations, and does not deal with very large integers.

4. We describe a modified version of the parallel algorithm mentioned in 3) to generate all combinations ***with repetitions*** of m elements chosen from an arbitrary set of n elements. The algorithm satisfies the same properties as in the case of the algorithm for combinations ***without repetitions***.

Contribution 3 was published in the *Parallel Processing Letters* journal [49]. This is a joint paper with my supervisor, *Dr. Ivan Stojmenovic*, presenting the parallel algorithm for generating all combinations ***without repetitions*** of m elements chosen from an arbitrary set of n elements in lexicographic order. Other contributions belong solely to the thesis author.

1.9 Organization

The remainder of this thesis is organized as follows. Chapter II provides a brief survey and comparison of the most important sequential combinations generation algorithms. Chapter III presents a comprehensive survey of existing parallel combinations generation algorithms. All the desirable properties are stated at the beginning of the chapter. A summary of the characteristics with respect to the properties they satisfy, is also provided. Chapter IV presents our new algorithm for the parallel generation of combinations. This algorithm satisfies all the important desirable properties presented in Chapter III. Chapter V presents a summary of the achieved results and a statement of the open problems. A comprehensive bibliography is provided at the end of the thesis.

Chapter II

Sequential Combinations Generation Algorithms

2.1 General

In this chapter, we will describe the common sequential combinations generation algorithms to provide some background for the consideration of parallel combinations generation algorithms. The sequential generation of combinations has been extensively studied since 1950s. Various sequential algorithms have been given for generating combinations using the binary representation [7, 8, 9, 11, 13, 16, 22] and set representation [5, 6, 15]. Payne and Ives [13] introduced one sequential combinations generation algorithm and compared it with several other algorithms [6, 7, 8, 9, 11] that existed before. In their comparison they claimed that their algorithm is the fastest. Akl [14] reviewed nine sequential combinations generation algorithms that had appeared to that point [5, 6, 7, 8, 9, 10, 11, 13], including the algorithms that were considered in [13]. Akl concluded that the algorithm of Mifsud [6] is the best in terms of speed, for all values of n and m tried; which contradicts the result presented in [13]. Few other algorithms have followed since then [11, 15, 16, 22]. In [11] the Gray codes were used to obtain a simple and efficient iterative algorithm. Semba [15] proposed a faster algorithm than the one proposed in [6]. Ruskey [16] proposed an adjacent

interchange algorithm using the Gray codes and in [22] the authors presented an algorithm to generate the new combination based on its lexicographical index without being dependent on the previous combination.

Combinations with repetitions and without repetitions and the relationship between the two will be introduced in the next sections. Three combinations generation algorithms [6,12, 15] will be described. Next, the experimental results for algorithms [6, 15] will be compared. In this thesis, we focus on the generation of the $C(m,n)$ combinations of m elements chosen from the set $A = \{a_1, a_2, \ldots, a_n\}$, in lexicographically ascending order, with the components of each combination in increasing order from left to right. We call these combinations *(m,n)*-combinations. Combinations generation using the minimum change order and at random will be introduced briefly. When instances of a combinatorial object are listed in a table, the order in which they were generated is top to bottom, left to right.

2.2 Important characteristics

When analyzing the efficiency of a combination generation algorithm, it is important to distinguish between the cost of ***generating*** and the cost of ***listing*** all instances of *(m,n)*-combinations. The word *generating* means producing all instances of *(m,n)*-combinations without actually outputting them. Some properties of *(m,n)*-combinations can be tested dynamically, without the need to check each element of the new instance. The word *listing* means the output of each object is required. The lower bound for producing all instances of *(m,n)*-combinations depends whether *generating* or *listing* is required [10, 12, 19].

In the case of generating, the time required to produce the *(m,n)*-combinations, without actually producing as output the elements of each combination, is counted. For example, an optimal sequential algorithm in this sense will generate all *(m,n)*-combinations in *O(C(m,n))* time, where *C(m,n)* is the number of *(m,n)*-combinations. Thus, the time required to generate all *(m,n)*-combinations is ***linear*** in the number of *(m,n)*-combinations. The ***delay*** when generating these combinations is the time needed to produce the next combination from current one. A generating algorithm has ***constant average delay*** if the time to generate all *(m,n)*-combinations is *O(C(m,n))*, i.e. the ratio *T(m,n)/C(m,n)* of the time *T(m,n)* needed to generate all *(m,n)*-combinations and the number of generated *(m,n)*-combinations, *C(m,n)*, is bounded by a constant. A generating algorithm has ***constant delay*** if the time to generate the next combination from the current one is bounded by a constant which does not depend on *m* and/or *n*. Constant delay algorithms are also called ***loopless*** algorithms, as the code for updating a given combination does not contain *repeat*, *while* and/or *for* loops.

In the case of listing, the time to actually output each of the *(m,n)*-combination is counted. In this case, an optimal sequential algorithm generates all *(m,n)*-combinations in *O(mC(m,n))* time, since it takes *O(m)* time to produce a combination as output.

The desirable properties of generating or listing all *(m,n)*-combinations can be presented briefly as follows:

Property 1. The algorithm lists all *(m,n)*-combinations in asymptotically optimal time. If *C(m,n)* is the total number of combinations and *m* is the average size of a combination, the time to list all *(m,n)*-combinations should be *O(mC(m,n))*.

Property 2. The algorithm generates all *(m,n)*-combinations with constant average delay. In other words, the algorithm takes $O(C(m,n))$ time to generate all *(m,n)*-combinations.

Property 3. The algorithm generates all *(m,n)*-combinations with constant delay.

Property 4. The algorithm does not use large integers in generating all *(m,n)*-combinations.

Property 5. The algorithm is the fastest known algorithm for generating all *(m,n)*-combinations.

An algorithm satisfying property 3 is sometimes called a ***loopless*** algorithm, since there is no loop in the procedure that finds the next combination from the current one. An algorithm satisfying property 3 also satisfies property 2. We consider property 3 independently of property 2 because of the following reason. In some cases an algorithm having constant delay property is considerably more sophisticated than the one satisfying the average delay property. In addition, an algorithm having constant delay property may need more time to generate all *(m,n)*-combinations than an algorithm having only constant average delay property.

The algorithms for generating all *(m,n)*-combinations can thus be classified into those following ***lexicographic order*** and those following ***minimum change*** order (i.e. using Gray code, adjacent exchange, or pairwise exchange).

2.3 Backtrack strategy

The backtrack strategy is a systematic search strategy that is used to enhance the efficiency of an exhaustive search, since the number of candidates for a solution is often exponential in input size [12]. Backtrack, in general, works on partial solutions to a problem. The solution is extended to bigger partial solution if there is a possibility of reaching a complete solution. We call this the *extend* phase. If an extension of the current solution is not possible, or a complete solution is reached and another one is sought, it backtracks to a shorter partial solution and tries again. This is called the *reduced* phase. In this thesis, we are concerned with the application of the backtrack strategy to the generation of the *(m,n)*-combinations. The backtrack strategy is normally related to the lexicographic order of the *(m,n)*-combinations. A very general form of the backtrack strategy to generate the *(m,n)*-combinations can be presented as follows [12]:

```
Initialize;
Repeat
    If current partial solution is extendable Then extend Else reduce;
    If current solution is acceptable Then report it;
Until search is over
```

This form does not show all the details involved in using the strategy and some modification may be applied. The crucial part of the method is to find an efficient test whether the current solution is extendable. The backtrack method is applied in the following to generate the *(m,n)*-combinations in lexicographic order.

2.4 Combinations without repetitions in lexicographic order

Generating combinations without repetitions of m elements chosen from the set of n elements $\{a_1, a_2, \ldots, a_n\}$ in lexicographic ascending order, using the *backtrack strategy*, involves the generation of each combination from its immediate predecessor. For simplicity, we assume that elements are integers $\{1, 2, \ldots, n\}$. Starting with the combination $\{1, 2, \ldots, m\}$, the next combination is found by scanning the current combination from right to left so as to locate the rightmost element that has not yet attained its maximum value. This element is incremented by one, and all positions of the elements to its right are reset to the lowest values possible. This means a combination $c_1c_2\ldots c_m$ is ***updatable*** if for some j, $1 \le j \le m$, $c_i = n\text{-}m+i$ for all $i > j$ and $c_j < n\text{-}m+j$ (index j is called the turning point). If $c_m < n$, then the next combination is obtained by simply incrementing c_m by one. If $c_j < n\text{-}m+j$, $2 \le j \le m$, to produce the next combination, c_j is incremented by one and then set $c_j \leftarrow c_{j-1}+1$, $c_{j+1} \leftarrow c_j+1$, ..., and $c_m \leftarrow c_{m-1}+1$. For example, when $n=9$, $m=6$ and the current combination is *1 3 4 5 6 8* the next combination becomes *1 3 4 5 6 9* because the element at the mth position has not attained its maximum value. If the current combination is *1 3 6 7 8 9* the next combination is obtained by backtracking to the 2nd element, which has not yet attained its maximum value, and then incrementing the element in this position by *1* and the elements to its right are reset to *5*, *6*, *7*, and *8*; then the combination becomes *1 4 5 6 7 8*. Last, when the current combination is *4 5 6 7 8 9*, the backtrack algorithm will terminate since this combination is no longer updatable. There exist different implementations of the backtrack algorithm [6, 12, 15]. The one given in [12] is shown below as ALGORITHM 1, whereas those given in [6, 15] will be presented in the next sections.

ALGORITHM 1. SCLRND(n, m) [12].

```
BEGIN
   Initialize c0, c1, c2, .... cm to -1, 1, 2, .... m:
   j←1:
   While j ≠ 0 Do
      output(c1, c2, .... cm):
      j ← m:
      While cj = n-m+j Do j ← j-1:
      cj ← cj+1:
      For i = j+1 To m Do ci ← ci-1+1:
   End While
END.
```

The algorithm starts with the combination $\{1, 2, \ldots, m\}$ and then the next combinations are generated until j becomes 0. The algorithm has *constant average delay* since the time to generate all (m,n)-combinations is $O(C(m,n))$. This can be easily obtained once we know the number of times the comparison $c_j = n-m+j$ is made [12]. This comparison is made once for each combination (i.e. $C(m,n)$ times), once more for each combination with $c_m = n$ (i.e. $C(m-1,n-1)$ times), once more for each combination with $c_{m-1} = n-1$ and $c_m = n$ (i.e. $C(m-2,n-2)$ times), ..., and once more still for each combination with $c_1 = 1$, $c_2 = 2$, ..., $c_n = n$ (i.e. $C(m-m,n-m) = 1$ time). So it is made $\sum_{j=0}^{m} C(m-j,n-j) = C(m,n+1)$ times. Knowing this, it can be seen that the statement $c_i \leftarrow c_{i-1}+1$ is executed $C(m,n+1) - C(m,n) = C(m-1,n)$ times. The comparison $j \neq 0$ is made $C(m,n)+1$ times, since $C(m,n+1) = ((n+1)/(n-m+1))C(m,n)$ and $C(m-1,n) = (m/(n-m+1))C(m,n)$. The algorithm is linear except *when* $m = n-o(n)$. In this case the algorithm can be applied to generate the $(n-m,n)$-combinations. Then, these combinations can be complemented to obtain the (m,n)-combinations [12]. For

the set $\{1, 2, 3, 4, 5, 6\}$ and $m = 4$, the (m,n)-combinations in lexicographic order without repetitions are listed in Table 3.

Table 3. Combinations without repetitions, $n=6$ and $m=4$.

1 2 3 4	1 3 5 6
1 2 3 5	1 4 5 6
1 2 3 6	2 3 4 5
1 2 4 5	2 3 4 6
1 2 4 6	2 3 5 6
1 2 5 6	2 4 5 6
1 3 4 5	3 4 5 6
1 3 4 6	

2.5 Correspondence between combinations without repetitions and with repetitions

A (m,n)-combination chosen from the set of n elements $A = \{a_1, a_2, \ldots, a_n\}$ can be represented as a sequence $c_1c_2\ldots c_m$ where $a_1 \leq c_1 < c_2 < \ldots < c_m \leq a_n$. A 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) can be established in the following way. Assume A is a set of n ordered integers $\{1, 2, \ldots n\}$. Let $x_i = c_i - i + 1$ where $1 \leq i \leq m$. Then $1 \leq x_1 \leq x_2 \leq \ldots \leq x_m \leq n-m+1$, and $x_1x_2\ldots x_m$ is a combination with repetitions of m out of $n-m+1$ elements; we call it an $(m,n-m+1)$-r_combination.

Because of the simple relation between x_i and c_i, 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 statement of the algorithms that generate these two types of combinations. 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 the same. For example, the combinations without repetitions and combinations with repetitions of $m = 4$ elements chosen from the set of $n = 6$ elements $A = \{1, 2, 3, 4, 5, 6\}$ are presented in Table 4. It can be seen that the combinations with repetitions are obtained from the combinations without repetitions by applying the relationship presented earlier. As an example, let $c_1c_2c_3c_4$ be a combination without repetition and equal to *1246*. By applying the relationship $x_i = c_i - i + 1$, the combination with repetitions can be found as follows. For $c_1 = 1$, $i = 1$ and hence $x_1 = 1$. For $c_2 = 2$, $i = 2$ and hence $x_2 = 1$. For $c_3 = 4$, $i = 3$ and hence $x_3 = 2$. For $c_4 = 6$, $i = 4$ and hence $x_4 = 3$. The combination with repetitions $x_1x_2x_3x_4$ becomes equal to *1123*. Also in Table 4, if combinations without repetition are *1 2 3 4*, *1 3 4 6*, *2 3 4 6* and *3 4 5 6* then the corresponding combinations with repetitions are *1 1 1 1*, *1 2 2 3*, *2 2 2 3*, and *3 3 3 3* respectively.

Table 4. Combinations without and with repetitions, $n=6$ and $m=4$.

Without repetitions	With repetitions
1 2 3 4	1 1 1 1
1 2 3 5	1 1 1 2
1 2 3 6	1 1 1 3
1 2 4 5	1 1 2 2
1 2 4 6	1 1 2 3
1 2 5 6	1 1 3 3
1 3 4 5	1 2 2 2
1 3 4 6	1 2 2 3
1 3 5 6	1 2 3 3
1 4 5 6	1 3 3 3
2 3 4 5	2 2 2 2
2 3 4 6	2 2 2 3
2 3 5 6	2 2 3 3
2 4 5 6	2 3 3 3
3 4 5 6	3 3 3 3

2.6 Mifsud [6]

In [6], an algorithm to generate combinations in lexicographic order was proposed. The combinations of the first n integers taken m at a time, $c_1c_2...c_m$, are generated in the array X in lexicographic order starting with an initial combination of the first m integers $\{1, 2, ..., m\}$. The algorithm is presented in the form of a procedure that takes n, m and X as arguments. Each of the $C(m,n)-1$ subsequent procedure calls, after the first, must have in X the previous generated combination in order to produce the next combination. The algorithm uses a global Boolean variable, *first*, which must be *true* before the first call. Then, *first* remains *false* until all combinations have been generated. When the procedure is called with X containing $n-m+1, n-m+2, ..., n$, X is left unchanged and the process stops. This algorithm is shown in ALGORITHM 2.1 and ALGORITHM 2.2. It can be observed that the algorithm requires $O(m)$ steps to produce the initial combination when *first* is set to *true*. Then for each subsequent procedure call, where *first* is set to *false*, the algorithm determines the next combination using the backtrack search by scanning a given combination once from right to left and then, *from an updatable position*, left to right. The average delay of the algorithm was found in [15] as follows. For a fixed h, the number of (m,n)-combinations which have h as the turning point is $C(h,n-m+h-1)$. This follows because $c_i = n-m+i$ when $i > h$ for each of these combinations while $c_1c_2...c_h$ can be any $(h,n-m+h-1)$-combination. The backtrack search examines and updates $m-h+1$ elements. Hence, the total number of examined elements to generate all combinations can be derived as follows:

$$\sum_{h=1}^{m}(m-h+1)C(h,n-m+h-1)+m$$

$$= C(m,n+1)-1$$

$$= \frac{n+1}{n-m+1}C(m,n)-1 .$$

Thus the algorithm examines, on average, less than $\frac{n+1}{n-m+1}$ elements and average delay is $O(\frac{n}{n-m})$. The delay is constant whenever $m = O(n)$. On the other hand, the average delay may be non-constant in some cases, for example when $n\text{-}m = O(\sqrt{n})$ [6].

ALGORITHM 2.1 SCLM1 [6].

```
BEGIN
   For h = 1 To C(m,n) Do
      SCLM2 (X, n, m);
      Produce x1x2...xm as output;
   End For
END.
```

ALGORITHM 2.2 SCLM2(X, n, m) [6].

```
BEGIN
   If first Then
      Initialize X to 1, 2, ..., m;
      first = false;
   Else
      If X(m) < n Then
         Increment X(m) by 1;
      Else
         For j = m To 2 Do
            If X(j-1) < n-m+j-1 Then
               Increment X(j-1) by 1;
               For k = j To m Do X(k) = X(j-1)+k-j+1;
            Else
               First = true;
            End If
         End For
      End If
   End If
END.
```

2.7 Semba [15]

In [15], an improved version of the algorithm [6] is presented. The kth combination is denoted by $X^{(k)} = \{ x_1^{(k)}, x_2^{(k)}, \ldots, x_m^{(k)} \}$ where $1 \le x_1^{(k)} < x_2^{(k)} \ldots < x_m^{(k)} \le n$. Let the index $h^{(k)}$ be the largest i such that $x_i^{(k)} < n\text{-}m+i$. If $x_i^{(k)} = n\text{-}m+i$ for all i $(1 \le i \le m)$, then the index $h^{(k)}$ is defined to be 0. The number of different elements between $X^{(k)}$ and $X^{(k+1)}$ is $d^{(k)}$ for $1 \le k \le C(m,n)$. In [6], the combinations are divided into these three classes:

$$C_1 = \{X^{(k)} \mid x_{h(k)}^{(k)} = n\text{-}m+h^{(k)}\text{-}1\}$$
$$C_2 = \{X^{(k)} \mid x_{h(k)}^{(k)} < n\text{-}m+h^{(k)}\text{-}1\}$$
$$C_3 = \{X^{(k)} \mid h^{(k)} = 0\}$$

The original algorithm generates $X^{(k+1)}$ from $X^{(k)}$ as follows: **(1)** Examine elements of $X^{(k)}$ from right to left to determine the index $h^{(k)}$, **(2)** Change values of $x_i^{(k+1)}$ for $i = h^{(k)}, \ldots, m$. The number of elements examined to determine the index $h^{(k)}$ in the first step is equal to the number of elements whose value is changed in the second step, in order to generate all the (m,n)-combinations in lexicographic order. This number was derived earlier and is equal to $\frac{n+1}{n-m+1}C(m,n)-1$. The improved algorithm is based on the following property. For $1 \le m \le n$:

(1) If $X^{(k)} \in C_1$, then
$$\begin{aligned} x_{h(k)}^{(k+1)} &= x_{h(k)}^{(k)}+1, \\ x_i^{(k+1)} &= x_i^{(k)} \quad (i \ne h^{(k)}), \\ h^{(k+1)} &= h^{(k)}\text{-}1 \end{aligned}$$

(2) If $X^{(k)} \in C_2$, then
$$\begin{aligned} x_i^{(k+1)} &= x_{h(k)}^{(k)}+i-h^{(k)}+1 \quad (h^{(k)} \le i \le m), \\ x_i^{(k+1)} &= x_i^{(k)} \quad (i < h^{(k)}), \\ h^{(k+1)} &= m. \end{aligned}$$

This property suggests that we are able to determine $h^{(k+1)}$ from $h^{(k)}$ after the element $x_{h(k)}^{(k)}$ is compared with $n-m+h^{(k)}-1$. Therefore, it is no longer needed to examine elements of $X^{(k+1)}$ from right to left. In particular, when $X^{(k)} \in C_1$, only the element $x^{(k+1)}$ will have to be changed. Hence, it is no longer required to change $m-h^{(k)}+1$ elements. In the original algorithm, to generate all combinations, the number of elements examined to determine the index $h^{(k)}$ is equal to the number of elements whose value is changed; this number is given above. In the case of the improved algorithm, shown in ALGORITHM 3.1 and ALGORITHM 3.2., the number of elements examined is equal to $C(m,n)$ and the number of elements whose value is changed is determined as follows. If $X^{(k)} \in C_1$ then $d^{(k)} = 1$. For a fixed h, $x_1^{(k)}, \ldots, x_{h-1}^{(k)}$ can be any $(h-1)$-subset of $\{1, \ldots, n-m+h-2\}$. If $X^{(k)} \in C_2$, then $d^{(k)} = m-h^{(k)}+1$. For a fixed h, $x_1^{(k)}, \ldots, x_h^{(k)}$ can be any h-subset of $\{1, \ldots, n-m+h-2\}$. We need to change m times to initialize $X^{(1)}$. Therefore, the total number of updated elements to generate all (m,n)-combinations is

$$\sum_{h=1}^{m} C(h-1, n-m+h-2) + \sum_{h=1}^{m} (m-h+1) C(h, n-m+h-2) + m$$

$$= C(m-1, n-1) + C(m, n) - 1$$

$$= (1+\frac{m}{n}) C(m, n) - 1.$$

Thus the improved algorithm does, on average, one examination and fewer than $(1+m/n) < 2$ updates per (m,n)-combinations. Therefore, the average delay is constant for any m and n, when $m \leq n$. This algorithm generates the (m,n)-combinations, $c_1 c_2 \ldots c_m$, of m objects from the set $\{1, 2, \ldots, n\}$ in the array X in lexicographic order. The integer h is non local to the

algorithm and must be greater than n before the first call. During the generation of the (m,n)-combinations, the integer variable h remains $1 \leq h \leq m$. When the last (m,n)-combination is generated, the algorithm sets h to zero.

ALGORITHM 3.1 SCLS1 [15].

```
BEGIN
   While (h ≠ 0) Do
      SCLS2(X, n, m);
      Produce x1x2...xm as output;
   End For
END.
```

ALGORITHM 3.2 SCLS2(X, n, m) [15].

```
BEGIN
   If m-h < 0 Then
      Initialize X to 1,2, ..., m;
      p = n-m; h = m;
      If n = m Then
         h = 0;
      Else
         If m-h = 0 Then
            Increment X(h) by 1;
            IF X(h) = n Then Decrement h by 1;
         Else
            Increment X(h) by 1;
            IF X(m) < p+h Then
               Increment h by 1;
               X(h) = X(h-1)+1;
               While h < m Do
                  Increment h by 1;
                  X(h) = X(h-1)+1;
               End While
            Else
               Decrement h by 1;
            End If
         End If
      End If
   End If
END.
```

The output of the algorithm for $n = 6$ and $m = 4$ is shown in Table 5. In this table, the index h, the number of different elements between two consecutive combinations d, and the combinations without repetitions are presented.

Table 5. Output of algorithm SCLS1 [15], for $n=6$ and $m=4$.

h	*d*	*Combination without repetitions*
4	1	1 2 3 4
4	1	1 2 3 5
3	2	1 2 3 6
4	1	1 2 4 5
3	1	1 2 4 6
2	3	1 2 5 6
4	1	1 3 4 5
3	1	1 3 4 6
2	1	1 3 5 6
1	4	1 4 5 6
4	1	2 3 4 5
3	1	2 3 4 6
2	1	2 3 5 6
1	1	2 4 5 6
0		3 4 5 6

2.8 Experimental comparison between Mifsud and Semba

The two algorithms [6, 15] were coded in *C* and run on a ***486* Patriot PC - *33* Mhz.** Table 6 contains the execution times of the programs for various values of m and n. The table entry is in the form *hours : minutes : seconds : ticks*. Table 7 shows the execution time to generate all the *(m,n)*-combinations of $\{1, 2, \ldots, 25\}$ for $4 \leq m \leq 23$. Both algorithms were

coded to generate the combinations and not to list them. Therefore, the time shown in these tables represents the time to create the combinations *without* actually producing as output the elements of each combination. It can be noticed in both tables that Semba's algorithm is faster than Mifsud's algorithm for large m.

Table 6. Running times for various values of n and m on a *486* Patriot *33* MHz.

	Execution time when m, n =				
Algorithm	**10, 20**	**8, 25**	**12, 25**	**20,40**	**25,50**
Mifsud	00:00:00:77	00:00:03:62	00:00:21:04	00:01:42:16	00:10:45:16
Semba	00:00:00:61	00:00:03:24	00:00:16:21	00:01:29:53	00:08:06:81

Table 7. Running time for *(m,n)*-combinations of *{1, 2, ..., 25}*, $4 \leq m \leq 23$ on a *486* Patriot *33* MHz.

m	Mifsud	Semba
4	00:00:00:00	00:00:00:00
5	00:00:00:22	00:00:00:16
6	00:00:00:55	00:00:00:50
7	00:00:01:53	00:00:01:37
8	00:00:03:63	00:00:03:13
9	00:00:07:19	00:00:05:93
10	00:00:12:03	00:00:09:61
11	00:00:17:25	00:00:13:30
12	00:00:21:20	00:00:15:65
13	***00:00:22:41***	***00:00:15:87*** ←
14	00:00:20:49	00:00:13:79
15	00:00:16:09	00:00:10:22
16	00:00:10:88	00:00:06:48
17	00:00:06:26	00:00:03:46
18	00:00:03:07	00:00:01:90
19	00:00:01:27	00:00:00:50
20	00:00:00:44	00:00:00:22
21	00:00:00:11	00:00:00:05
22	00:00:00:05	00:00:00:00
23	00:00:00:00	00:00:00:00

2.9 Generation of combinations at random

The original method was given by Moses and Oakland [23] to generate random permutations. Balbine [24] and Pike [25] observed that the method proposed by [24] can be applied to generate random combinations. This application is described in [12] as follows.

Consider the set of n element $A = \{a_1, a_2, \ldots, a_n\}$ where each element a_i has a probability $1/n$ of being selected. Generating combinations of m elements from the set A at random involves the selection of one of the n elements at random and then the selection of a random $(m-1)$ combination of the remaining $n-1$ elements. If each of the $C(m,n)$ combinations is equally probable, then the probability of a combination $\{a(i_1), a(i_2), \ldots, a(i_m)\}$ occurring is $1/C(m,n)$. The algorithm can be implemented by using an auxiliary array to save the indices of those elements not yet chosen. After the first $j - 1 \leq k$ elements have been chosen, p_j, $p_{j+1}, \ldots, p_n$ are the subscripts of the $n - j + 1$ elements that have not yet been chosen. If it is unnecessary to preserve the order in the set, the array p can be eliminated and that leaves the random combination in the first k positions of the set. This algorithm is shown in ALGORITHM 4.

ALGORITHM 4. SCRRND (n, m) [12].

```
BEGIN
   For j = 1 To n Do p_j ← j;
   X ← ∅;
   For j = 1 To m Do
      r ← rand(j, n);
      X ← X ∪ {a_pr};
      p_r ← p_j;
   End For
END.
```

2.10 Generation of combinations in a minimal change order

Each of the combinations of m elements out of the set of n elements $A = \{a_1, a_2, \ldots, a_n\}$ using the binary representation, *(m,n)*-combination, corresponds to an *n-bit* vector in which the *i*th bit is *1* if and only if a_i is in the combination. Thus the problem of generating all possible *(m,n)*-combinations is reduced to the problem of generating all possible *n*-bit vectors of weight *m*, i.e. containing exactly *m 1-bits*. The most straightforward way of generating all *n-bit* vectors of weight *m* is to count in binary, but in this case many elements may change from one subset to the next, which for some applications is not a desirable property. It is desirable to have the change from one combination to the next as small as possible. This can be achieved by using the *Gray codes* [12], defined in chapter I. Several iterative and recursive algorithms, using the binary representation, for computing such sequences are discussed in [11]. The authors in [11] concluded that these algorithms are considerably complicated and relatively inefficient. In the case of generating the *(m,n)*-combinations, the smallest possible change between successive combinations is the changing of one element for another. In terms of bit vectors, this means that successive vectors must differ in exactly two bits, a *0* becomes a *1* and *1* becomes a *0*. Also in [11], the authors presented a simple efficient iterative algorithm for the generation of all *n-bit* vectors and all *n-bit* vectors of weight *m*. In both cases, successive bit vectors differ as little as possible and the amount of work needed for the transformation from one vector to the next is a small constant independent of *n* and *m*. An adjacent interchange, for the case of combinations in the binary representation, is the replacement of *01* by a *10*, or *10* by a *01*, in a bit vector. Ruskey [16] presented a constant average time adjacent interchange algorithm for generating

all n-bit vectors of weight m for the case where n is even and m is odd. In this algorithm, succeeding bit vectors differ by a single adjacent interchange.

2.11 Combinations generation in lexicographic order from arbitrary elements

In this section, we briefly discuss the correspondence between combinations from the set of n integers $S = \{1, 2, \ldots, n\}$ and the combinations from the set of n arbitrary elements $A = \{a_1, a_2, \ldots, a_n\}$, using sequential combination generation algorithms. The correspondence is trivial and can be stated as follows. A (m,n)-combination chosen from the set S can be represented as a sequence $c_1c_2\ldots c_m$ where $1 \le c_1 \le c_2 \le \ldots \le c_m \le n$. Let $x_i = a(c_i)$ where $1 \le i \le m$. Then $a_1 \le x_1 \le x_2 \le \ldots \le x_m \le a_n$, and $x_1 x_2 \ldots x_m$ is an (m,n)-combination chosen from the set A. Because of the simple relation between x_i and $a(c_i)$ any algorithm that generates the (m,n)-combinations from the set S may be used to generate the (m,n)-combinations from the set A and vice versa. The only difference is in the output statement of the algorithms that generate these combinations. For example, let *2 3 5 6* be a *(4,6)*-combination from the set *{1, 2, 3, 4, 5, 6}*. By applying the relationship $x_i = a(c_i)$, the *(4,6)*-combination *b c e f* from the set *{a, b, c, d, e, f}* will be obtained. Table 8 lists all the (m,n)-combinations for $n = 6$ and $m = 4$ from both sets S and A.

Table 8. Combinations from $S=\{1, 2, 3, 4, 5, 6\}$ and $A=\{a, b, c, d, e, f\}$ in lexicographic order, $n=6$ and $m=4$.

1 2 3 4	a b c d	1 3 5 6	a c e f
1 2 3 5	a b c e	1 4 5 6	a d e f
1 2 3 6	a b c f	2 3 4 5	b c d e
1 2 4 5	a b d e	2 3 4 6	b c d f
1 2 4 6	a b d f	2 3 5 6	b c e f
1 2 5 6	a b e f	2 4 5 6	b d e f
1 3 4 5	a c d e	3 4 5 6	c d e f
1 3 4 6	a c d f		

2.12 Numbering combinations

There exits a one-to-one correspondence between the integers *1, 2, ..., C(m,n)* and the set of *(m,n)*-combinations of *{1, 2, ..., n}* listed in lexicographic order. The mapping of all *(m,n)*-combinations into integers is called ***ranking***. For example, let *f(X)* be a ranking procedure for *(3,4)*-combinations of the set *{1, 2, 3, 4}*. Then, in lexicographic order, $f(123) = 1$, $f(124) = 2$, $f(134) = 3$, and $f(234) = 4$. The inverse of ranking, called ***unranking***, is mapping integers *1, 2, 3, ..., C(m,n)* to corresponding combinations. For instance, $f^{-1}(3) = 134$ in the last example. In this section we will focus only on the lexicographic order, although ranking and unranking functions are also used for combinations listed in minimal change order.

Let $x_1x_2...x_m$ represents a *(m,n)*-combination out of *{1, 2, ..., n}*. Then $1 \le x_1 \le n-m+1$, $x_1 \le x_2 < n-m+2$, ... $x_{m-1} < x_m \le n$. Therefore, element x_i has $n-m+1-x_{i-1}$ different choices. Let y be such value for which $x_1x_2...x_{i-1}y$ can be completed to represent a *(m,n)*-combination. In the case of ranking combinations out of *{1, 2, ..., n}*, y is ranged between $x_{i-1}+1$ and $x_i - 1$. Any *(m,n)*-combination that starts with $x_1x_2...x_{i-1}y$ is in fact a *(m-i,n-y)*-combination. The number of such combinations is *C(m-i,n-y)*. Thus, the ranking algorithm for combinations out of *{1, 2, ..., n}* is shown in ALGORITHM 5.1 [19].

ALGORITHM 5.1. RANKC($x_1x_2...x_m$)
```
BEGIN
    RANKC ← 1;
    x0 ← 0;
    For i ← 1 To m Do
       For y ← xi-1+1 To xi-1 Do
          RANKC ← RANKC + C(m-i,n-y);
END.
```

In lexicographic order, the $C(4,6) = 15$ $(4,6)$-combinations are listed in Table 3. The rank of *2346* is determined as $1+C(4-1,6-1)+C(4-4,6-5) = 1+10+1 = 12$ where the last two summands correspond to combinations that start with *1* and *2345*, respectively. The rank of *3578* in $(4,9)$-combinations *is* $1+C(4-1,9-1)+C(4-4,9-2)+C(4-2,9-4)+C(4-3,9-6) = 104$ where the last four summands correspond to combinations starting with *1, 2, 34,* and *356,* respectively. A simpler formula is given in [48] where the rank of combination $x_1x_2...x_m$ is $C(m,n) - \sum_{j=1}^{m} C(j, n-1-x_{m-j}+1)$. It comes from the count of the number of combinations that follow $x_1x_2...x_m$ in lexicographic order. These are all combinations of j out of elements $\{x_{m-j+1}+1, x_{m-j+1}+2, ..., x_n\}$, for all j, $1 \le j \le m$. In the last example, combinations that follow *3578* are all combinations of *4* out of *{4, 5, 6, 7, 8, 9}*, combinations with first element *3* and three others taken from *{6, 7, 8, 9}*, combinations which start with *35* and having two more elements out of set *{8, 9}* and combination *3579*.

Now, we consider an unranking procedure for combinations. It is the inverse of ranking function and given in ALGORITHM 5.2. [19].

ALGORITHM 5.2. UNRANKC(rank,n, $x_1x_2...x_m$)

```
BEGIN
    i ← 0;
    x0 ← 0;
    Repeat
        i ← i+1;
        y ← xi-1+1;
        While C(m-i,n-y) ≤ rank Do
            rank ← rank - C(m-i,n-y);
            y ← y+1;
        End While
        xi ← y;
    Until rank = 0;
    For j = i+1 To m Do xj ← n-m+j;
END.
```

One may ask the question, what is the *104*-th *(4,9)*-combination? There are *C(3,8)=56* *(4,9)*-combinations starting with a *1* followed by *C(3,7)=35* starting with *2* and *C(3,6)=20* starting with *3*. Since $56+35 \leq 104$ but $56+35+20 > 104$, the requested combination begins with a *3*, and the problem is reduced to finding the *104-56-35=13*-th *(3,6)*-combination. There are *C(2,5)=10* combinations starting with *34* and *C(2,4)=6* starting with a *5*. Since $13>10$ but $13<10+6$, the second element in combination is *5*, and we need to find the *13-10=3*-rd *(2,4)*-combination out of *{6,7,8,9}* which is *78*, resulting in combination *3578* as the *104*-th *(4,9)*-combination.

The number of *(m,n)*-combinations is *C(m,n)*, which may become a very large integer. The ranks, being large integers, may need *O(n)* number of memory location to be stored and also *O(n)* time for their manipulation. It should be mentioned that dealing with large integers is not a desirable property. Both algorithms have runtime *O(mn)*.

2.13 Conclusion

There are several different techniques for the sequential generation of combinations of m elements from the set of n elements $S = \{s_1, s_2, \ldots, s_n\}$. We can generate these *(m,n)*-combinations in lexicographic order, in minimal change order, and at random. The backtracking method is widely used to generate the *(m,n)*-combinations in lexicographic order using the set representation. The Gray code order was mainly used to generate the *(m,n)*-combinations in the minimal change order using the binary representation. The algorithms that generate the *(m,n)*-combinations without repetitions may be used to generate

the *(m,n)*-combinations with repetitions allowed. The sequential algorithms that generate the *(m,n)*-combinations from the set of n integers $S = \{1, 2, \ldots, n\}$ may be used to generate the *(m,n)*-combinations from a set of n arbitrary objects $A = \{a_1, a_2, \ldots, a_n\}$. The best sequential combination generation algorithm: **1)** lists the *(m,n)*-combinations in lexicographic order, **2)** is cost-optimal, **3)** uses constant time between any two consecutive combinations, **4)** does not use large integers, and **5)** is the fastest known algorithm to generate all *(m,n)*-combinations. The unranking function can be used to map integers *1, 2, ..., C(m,n)* into *(m,n)*-combinations. However, this function deals with large integers, such as *C(m,n)*, which is not a desirable property and must be avoided whenever possible.

Chapter III

Parallel Combinations Generation Algorithms

3.1 General

Parallel processing becomes more and more feasible in practice due to the drastically lowered hardware cost and the advancement of hardware technology. Parallel computers are computers which have several processors. In this approach, the problem to be solved is broken down into subparts which are solved simultaneously, each on a different processor. Since the cost of electronic components is decreasing rapidly, it is feasible to build computers economically with thousand of processors. Using a parallel computer is a way to achieve higher computing speeds. This appealing approach has promptly increased interest in parallel computation and hence in the area of design and analysis of parallel algorithms. The growing importance of parallel computers and algorithms is highlighted in [19,42,43].

Many parallel algorithms for enumeration of combinatorial objects have been developed in the last fifteen years, in particular parallel algorithms for generating combinations. Optimal or near optimal algorithms for generating combinations on a number of parallel computers were introduced. The first to be investigated were algorithms designed for more powerful

models, this means the various variants of the Parallel Random Access Machine (PRAM), then later for weaker models, namely the linear array of processors. At first, the number of processors required were on the order $O(C(m,n))$ then later only on the order $O(n)$. In this chapter, all known parallel algorithms for generating combinations that exist in the literature are first discussed and then their characteristics are compared.

Various ways of using multiple processors, when generating combinations, appeared in the literature. However, most of them can be classified into the following methods:

- Division of instances into groups.
- Shared instances.

These methods will be briefly described in the subsequent sections.

3.2 Division of instances into groups

In this method, there are an arbitrary number of k processors available; each of them produces an interval of N/k combinations, where N is the total number of (m,n)-combinations to be generated (i.e. $N = C(m,n)$). For example, when $S = \{1, 2, 3, 4, 5, 6\}$, $m = 4$, and $k = 4$ the *15* $(4,6)$-combinations are produced by *4* processors generating combinations *1 2 3 4*, *1 2 3 5*, *1 2 3 6* and *1 2 4 5*; *1 2 4 6*, *1 2 5 6*, *1 3 4 5* and *1 3 4 6*; *1 3 5 6*, *1 4 5 6*, *2 3 4 5* and *2 3 4 6*; *2 3 5 6*, *2 4 5 6* and *3 4 5 6* respectively. The best known technique for the division of instances into groups is to apply a sequential algorithm on each processor. This method is used by Akl [28] to generate combinations. This technique will be described in more detail in the following.

Suppose a sequential combinations generation algorithm is available, with a procedure unrank(h,X) to generate the h-th (m,n)-combination in X. The job is divided equally into k processors. The distribution is such that processor i generates (m,n)-combinations 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 (m,n)-combination ranked N. Each processor needs to receive the beginning and end of its interval of consecutive instances in order to produce its group of instances of the (m,n)-combination. A master processor can be used to supply each other processor the necessary data and program for generating all (m,n)-combinations. After the combinations are divided and each processor is given the beginning and end of its interval of consecutive combinations, there is no need for cooperation between processors, since each processor may generate its combinations by running a sequential algorithm. This is the simplest possible model since communication among processors is not needed.

There are two problems with the method. The first problem is that the division into groups may involve large integers or could be computationally expensive. The second problem is that the processors should be able to store the whole (m,n)-combination, i.e. they must have storage of size $O(m)$. This is an undesirable property since processors must be powerful, close to the power of processor on a sequential computer. In the case of the interconnection networks, each computational model is made of a large number of small processors. It is desirable that each processor has a memory of constant size. Therefore, an alternative (shared instances) approach for generating the (m,n)-combinations is investigated in the following.

3.3 Shared instances

In this approach, m processors produce the (m,n)-combination $c_1c_2...c_m$ such that processor i is responsible for producing element c_i. For example, the $(4,6)$-combination *2 3 5 6* is produced in the following way: processor *1* produces *2*; processor *2* produces *3*; processor *3* produces *5*; processor *4* produces *6*. This method is widely used for various generating problems. Also different models of computation are used, including the simple model, linear array of processors, and the very powerful models, CREW PRAM and EREW PRAM, explained in chapter I.

In a parallel computation, all processors are running the same program synchronously, but on different data (SIMD model explained in chapter I). Usually any parallel architecture has a master processor that supplies other processors with the program which is common for all processors. In a parallel algorithm, processors are operating in a ***lock step*** fashion. This means all processors perform the same instructions at any given time and the next instruction starts when a signal to perform it is given by a master processor or clock. This is also called synchronized computation.

3.4 Optimality properties of parallel generation algorithms

There exist a number of optimality properties for parallel algorithms that generate combinatorial objects. If the algorithm satisfies all these properties, then it can be stated that the algorithm is optimal in every known and reasonable sense. These optimality properties are stated and explained as follows [20]:

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

This property allows the distinction 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 $S = \{1, 2, \ldots, n\}$). In a sequential algorithm this property is trivially satisfied as the index i can 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 [20].

Property 2. The instances of an object are listed in lexicographic order.

For example *1 2 3 4* precedes *1 2 3 5* in lexicographic order. This ordering is intuitive and one may verify easily that all instances have been generated. It is the most commonly used ordering primarily because it has a simple characterization and it agrees with dictionary and numerical ordering.

Property 3. 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 is to match the running time of an optimal sequential algorithm.

This property can be further specified as follows:

a) The time required to *create* the instances of a combination is counted without actually producing as output the m elements. For example, an optimal sequential algorithm in this sense, generates all (m,n)-combinations in $O(C(m,n))$, i.e. time linear in the number of instances.

b) The time to actually *output* each instance in full is counted. An optimal sequential algorithm generates all (m,n)-combination in $O(mC(m,n))$ time, since it takes $O(m)$ time to produce a combination as output.

In the context of this thesis, generating combinations in parallel is considered as producing its instances as output. Therefore, we use definition (b) to determine whether the cost of a parallel algorithm is indeed optimal. Designing parallel combination generation algorithm whose cost is optimal under definition (a) remains an open problem.

Property 4. The time required by the algorithm between any consecutive instances of an object is ***constant***.

From the theoretical point of view, the best algorithm aims to produce each instance of an object in constant time. In the case where the output of one computation serves as input to another, constant time delay between outputs becomes important. This is particularly true in applications using systolic arrays.

Property 5. The model of parallel computation should be as *simple* as possible.

The result of an algorithm is more valuable if it was generated on a parallel model that is based on the fewest assumptions possible about processors and connectivity, from both theoretical and practical points of view. One may indicate that the weaker the model the stronger an optimality result, since a more powerful model can simulate the algorithm with no increase in running time. The simplest such model is the linear array of processors, which is weaker than the two-dimensional array and hypercube models which are in turn weaker than the PRAM machine [20].

Property 6. Each processor needs as little memory as possible, specifically a ***constant number of words*** each of $O(\log n)$ bits. Each word is thus capable of storing a binary encoding of one of the elements of $S = \{s_1, s_2, \ldots, s_n\}$, or of an integer no larger than n.

This property indicates that no processor can by itself store an array of size n, or a large combinatorial number such as $n!$. For a collection of algorithms for a specific problem all having the same running time but different memory considerations, the algorithm that needs the least memory requirements is usually considered the best. In practice, when an algorithm is implemented on a collection of tiny processors on a chip, small memory usage is usually desirable.

Property 7. The algorithm is *adaptive*, i.e. it is capable of adjusting itself automatically to the number of processors available.

An adaptive algorithm is desirable when the actual number of processors that is needed to run the algorithm is not equal to the theoretical number for which the algorithm was designed. A linear array with an arbitrary number of processors is used for such an algorithm. The algorithm adjusts itself to the available number of processors at run time in order to be able to execute correctly.

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

3.5 Available algorithms

In this chapter, the principal parallel combinations generation algorithms are briefly described. All known algorithms are summarized in a table highlighting the desirable properties they do or do not satisfy.

3.5.1 Gupta and Battacharjee [21]

In [21], a parallel algorithm to generate the *(m,n)*-combinations in lexicographic order is presented. The algorithm uses the Concurrent-Read, Exclusive-Write Shared Memory SIMD model, i.e. CREW SM SIMD. Before presenting the main parallel algorithm for generating the *(m,n)*-combinations, PCLGB, two parallel algorithms that are used by PCLGB are

presented first. These algorithms are: the Parallel Cumulative Sum, PCSGB, and the Parallel Arithmetic Square, PASGB.

Algorithm PCSGB deals with the problem of computing the partial sums $\sum_{i=1}^{h} x_i$, $1 \le h \le n$, of a set of n numbers $\{x_1, x_2, \ldots, x_n\}$. The algorithm replaces the (i,j)th element $M(i,j)$ of a tm matrix M by the partial row sum $\sum_{k=h}^{j} M(i,k)$ where $1 \le i \le t$, $h \le j \le m$, and $1 \le h \le m$. The algorithm computes $t(m-h+1)$ partial row sums in all. The algorithm is shown in ALGORITHM 6.1.

ALGORITHM 6.1. PCSGB(h, m, t, M) [21].

```
BEGIN
  For z = 1 To h Do in parallel p = ⌈(m - h + 1)/2⌉;
  For k = 1 To ⌈log(m - h + 1)⌉ Do
     For z = 1 To h*p Do in parallel
          i = ⌈z/p⌉;
          j = z - (i - 1)*p;
          v = ⌈j/2^(k-1)⌉;
          s = j - (v - 1)*2^(k-1);
          w = (2v - 1)*2^(k-1) + a - 1;
          M(i,w+s) = M(i,w+s) + M(i,w);
     Endfor
  Endfor
END.
```

In a CREW SM SIMD computer having $t\lceil (m-h+1)/2 \rceil$ processors, the algorithm computes all the desired sums in $O(\log(m-h+1))$ steps (proof is given in [21]).

The algorithm PASGB computes for given m and n, a mn matrix N such that the (i,j)th element of N, $N(i,j)$, is equal to $C(i-1, n-j)$ for $m-i+1 \le j \le n$ and $1 \le i \le m$; otherwise it is equal to 0. The algorithm is presented in ALGORITHM 6.2.

ALGORITHM 6.2. PASGB(n, m, N) [21].

```
BEGIN
    For j = 1 To n Do
        For h = 1 To m Do in parallel N(h, j) = 0;
        For h = 1 To n Do in parallel N(1, h) = 1;
        For k = 1 To n-1 Do
            For h = 2 To m Do in parallel
                If (h ≤ m - n + k) Then N(h, n-k) = N(h-1, n-k+1) + N(h, n-k+1);
            Endfor
        Endfor
    Endfor
END.
```

In a CREW SM SIMD computer having m processors, the algorithm computes all desired elements of the matrix N in $O(n)$ steps (proof is given in [21]). We now present the main algorithm for generating the combinations of m elements of the set $\{1, 2, \ldots, n\}$ in lexicographic order in parallel, PCLGB. The algorithm uses $C(m,n)$ processors. The output of the algorithm is the array $X(C(m,n),m)$. The jth element of the ith combination is $X(i,j)$, where $1 \leq j \leq m$ and $1 \leq i \leq C(m,n)$. The algorithm is shown in ALGORITHM 6.3.

ALGORITHM 6.3. PCLGB(m, n, X) [21].

```
BEGIN
    Call PASGB(n, m, N);
    For k = 1 To m Do
        For z = 1 To n*(n-m+1) Do in parallel
            i = ⌈z/n⌉; j = z - (i - 1)*n;
            If (k+i-1) ≤ (n-m+k) Then
                M(i,j) = N(m-k+1,j);
            Else
                M(i,j) = 0;
            Endif
        Endfor
        Call PCSGB(k, n-m+k, n-m+1, M);
        For z = 1 To C(n, m) Do in parallel
            If (k = 1) Then
              p = 0; i = 1; M(i, 0) = 0; u = 0;
            Else
              p = p+ M(i, u); i = u-k+2;
            Endif
            f = u + 1; u = n - m + k;
            While (f ≠ u) Do
                d = ⌊(f+u)/2⌋;
                If (z ≤ p+M(i,d) Then
                    u = d;
                Else
                    f = d + 1;
                Endif
            Endwhile
            N(z, k) = u;
        Endfor
    Endfor
END.
```

It was proved in [21] that, using a CREW SM SIMD computer, algorithm PCLBG generates all the *(m,n)*-combinations in lexicographic order in time of $O(\min(m\log(n-m+1),(n-m)\log(m+1)))$ using $C(m,n)$ processors. Since the elements in each combination are in lexicographic order, at the kth iteration each of the $C(m,n)$ processors finds the element among $k, k+1, \ldots, n-m+k$ in which it belongs by using the binary search technique. Only one element is generated at each iteration by each of the $C(m,n)$ processors. The algorithm is optimal with respect to time when $C(m,n)$ processors are available. The algorithm is not adaptive.

3.5.2 Tang, Du and Lee [22]

In [22], the authors indicated that the algorithms that generate the *(m,n)*-combination based upon the previously generated one are not suitable to generate all combinations in parallel, because each new combination must depend on the previous one. Hence, they presented an algorithm to generate the new combination based on its lexicographical index without being dependent on the previous combination as follows. The total number of *(m,n)*-combinations is divided into $n-m+1$ groups. Each combination is decided by m levels of decisions, using the tree hierarchical structure. In general the group $G_{i(1),i(2),\ldots,i(j),i(j+1),\ldots,i(m)}$ for *some* $1 \le i(1) \le i(2) \le \ldots \le i(j) \le n-(m-j) = n-m+j$ and $i(j+1) = i(j+2) = \ldots = i(m) = *$, contains $C(m-j,n-i(j))$ combinations and can be divided into $(n-(m-(j+1)))-i(j)$ groups $G_{i(1),i(2),\ldots,i(j),i(j+1),i(j+2)\ldots i(m)}$ for $i(j) < i(j+1) \le n-m+j+1$ and $i(j+2) = i(j+3) \ldots = i(m) = *$, by the $(j+1)$th level decision. For example, when $n = 6$ and $m = 3$, $C(m,n) = 20$. Each combination is decided by three levels of decisions. The first level decision divides the

entire set $G_{*,*,*}$ into $n-m+1 = 4$ groups: $G_{1,*,*}$ $G_{2,*,*}$ $G_{3,*,*}$ and $G_{4,*,*}$, as shown in Figure 2. Each group $G_{i,*,*}$ contains $C(m-1, n-i)$ combinations which are headed by element i, $1 \leq i \leq n-m+1 = 4$. For instance, $G_{1,*,*}$ is a group that contains $C(2,5) = 10$ combinations where the first element is *1* and next two elements are selected out of the five elements *2, 3, 4, 5, 6*. Furthermore, the group $G_{1,*,*}$ can be divided into *4* groups $G_{1,2,*}$ $G_{1,3,*}$ $G_{1,4,*}$ and $G_{1,5,*}$ by the second level decision. There are $C(1,n-3)=3$ combinations in $G_{1,3,*}$ where the first element is *1*, the second element is *3* and the third element is out of the items *4, 5, 6*. The group $G_{i,j,*}$ for some $1 \leq i \leq j \leq n = 6$, can be divided into $G_{i,j,j+1}$ $G_{i,j,j+2}$ ... $G_{i,j,n} = G_{i,j,6}$. In each group $G_{i,j,k}$ there is only one combination *i j k*. For instance, the group $G_{4,5,*}$ contains only one combination, *4 5 6*. Thus, to determine the *i*th combination, we need to determine which group it belongs to. Assume we want to determine the *6*th *(m,n)*-combination. Since $G_{1,*,*}$ contains $C(2,5) = 10$ combinations and $6 < 10$, the *6*th combination belongs to $G_{1,*,*}$. Hence the first element must be *1*. Since $G_{1,2,*}$ contains $C(1,4) = 4$ combinations and $6 > 4$, the *6*th combination does not belong to $G_{1,2,*}$. Hence, we now try to decide whether it belongs to $G_{1,3,*}$. Since $G_{1,3,*}$ contains $C(1,3) = 3$ combinations and $6-4 = 2$, the *6*th combination belongs to $G_{1,3,*}$. Using the same reasoning, we conclude that the *6*th combination belongs to $G_{1,3,5}$ and hence it is *1 3 5*.

Because of the capability of generating the combination for each index i, $1 \leq i \leq C(m,n)$, $C(m,n)$ processors, each labeled with an i, may be used. The *i*th processor will generate the *i*th combination using the SIMD model. The *i*th processor generates the *i*th combination in $O(n)$ time. Hence, all combinations can be produced in parallel in $O(n)$ time if $C(m,n)$

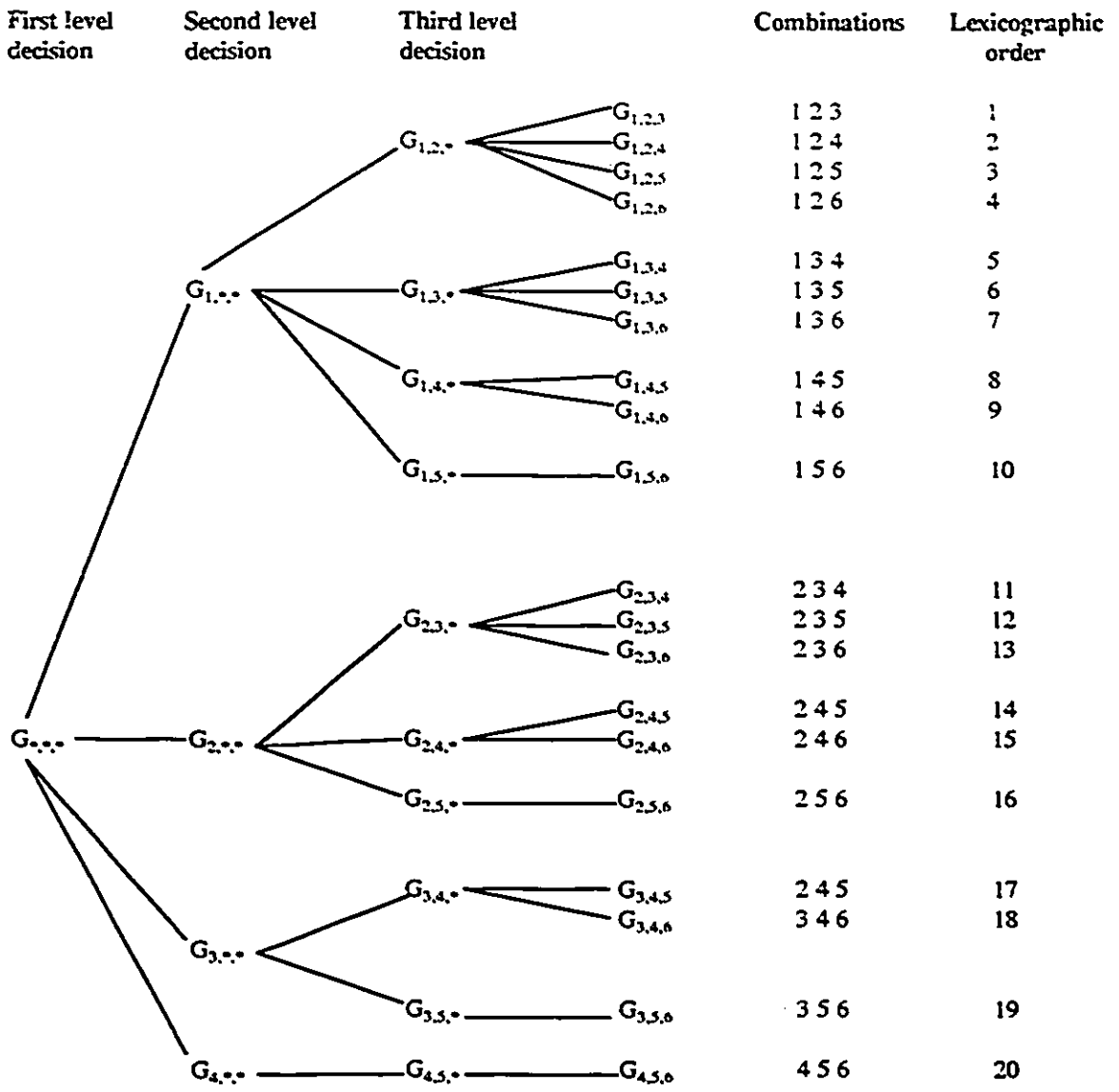

Figure 2. Combination hierarchical structure [22].

processors are used. If k processors are used and $k < C(m,n)$, a processor can be used more than once by assigning different indexes, and for each index the corresponding combination is generated. The ith processor is assigned index i at the first time, index $k+i$ at the second time, and index $(h-1)k+i$ at the hth time, for $1 \le h \le \lceil C(m,n)/k \rceil$. In total, the algorithm requires $O(n\lceil C(m,n)/k \rceil)$ units of time to generate all $C(m,n)$ combinations, if k processors are available. The algorithm is given by ALGORITHMS 7.1, 7.2 and 7.3.

ALGORITHM 7.1 PCLTDL(n, m) [22].

```
BEGIN
    For i = 1 To C(m,n) Do in parallel:
        call SCLTDL(X,n,m,i);
        produce x1x2...xm;
    Endfor
END.
```

ALGORITHM 7.2 SCLTDL(X,n, m, i) [22].

```
BEGIN
    v = COMB(n,m)
    If i <= v Then
      rp = m;
      item = 1;
      v = v*m;
      v = v div n;
      n = n-1;
      m = m-1;
      For q = 1 To rp Do
          While (i > v) Do
              i = i-v;
              item = item+1;
              v = v*(n-m);
              v = v div n;
              n = n-1;
          Endwhile
          X(q) = item;
          item = item+1;
          v = v*m;
          v = v div n;
          n = n-1; m = m-1;
      Endfor
    Endif
END.
```

ALGORITHM 7.3 COMB(n, m) [22].

```
BEGIN
    t1 = 1; t2 = 1;
    For j = To m Do
      t1 = t1*n;
      t2 = t2*t3;
      n = n-1;
    Endfor
    COMB = t1 div t2;
END.
```

3.5.3 Chan and Akl [26]

In [26], a parallel algorithm for generating the *(m,n)*-combinations in lexicographic order is presented. The CREW SM SIMD model of computation is used. The processors are numbered from *1* to *m* and an individual processor is referred as p_i, $1 \leq i \leq m$. At any given time, all the processors operate synchronously. Some of them are active and execute the same instruction on a different set of data, while the other processors are inactive.

Assuming that the *(m,n)*-combinations are of the set *{1, 2, ..., n}*, the algorithm is based on the following two properties:

1. For $1 < m < n$, the first combination is $1\ 2\ \ldots\ m$ and the last combination is $(n\text{-}m+1)(n\text{-}m+2)\ \ldots\ n$.

2. Denote the last combination by $L = x_1\ x_2\ \ldots\ x_m$. If $E_q = y_1\ y_2\ \ldots\ y_m$ is one of the other possible combinations, then:

   (i) $y_1 < y_2 < \ldots < y_m$ and $y_i \leq x_i$ for $1 \leq i \leq m$.

   (ii) If there is a subscript i, $2 \leq i \leq m$, such that all y's from y_i to y_m equal x_i to x_m respectively, then the next successive combination is given by:

   $E_{q+1} = z_1\ z_2\ \ldots\ z_m$, where $z_j = y_j$ for $1 \leq j \leq (i\text{-}2)$, and
   $z_j = y_{i-1}+j-i+2$ for $(i\text{-}1) \leq j \leq m$.
   Otherwise, $E_{q+1} = y_1\ y_2\ \ldots\ y_{m-1}\ (y_m+1)$.

The validity of property *2* is established by induction on i [26]. The parallel algorithm is based on the following argument. The initial combination is $1\ 2\ \ldots\ m$. If E_q is the current

combination, then the next successive combination is given by *2(ii)*. In shared memory, the algorithm uses four arrays *A*, *B*, *C* and *D*, each of length *m*. The first three arrays are used to store intermediate results, whereas the fourth array serves as an output buffer to place the new combination generated. The *i*th position of each of these arrays is denoted by *A(i)*, *B(i)*, *C(i)*, and *D(i)*. The algorithm is shown in ALGORITHM 8.

ALGORITHM 8. PCLCA(n, m) [26].

```
BEGIN
(1) For i = 1 to m do P(i) in parallel:
        A(i) = n - m + i;
        B(i) = i;
        C(i) = B(i) - A(i);
        D(i) = i;
    Endfor
    While not C(1) do
(2)     k = 0;
(3)     For i = 2 to m Do P(i) Do in parallel:
          If not C(i - 1) and C(i) Then
(3.1)         B(i-1) = B(i-1) + 1;
              k = i;
          Endif
        Endfor
(4)     If (k = 0) Then Do P(m);
(4.1)     B(m) = B(m) + 1;
        Else
(4.2)     For i = k To m Do P(i) in parallel B(i) = B(k-1)+(i-k+1);
        Endif
(5)     For i = 1 To m Do P(i) in parallel
          D(i) = B(i);
          C(i) = B(i) - A(i);
        Endfor
    Endwhile
END.
```

The overall complexity is governed by the *while* statement, which is executed *C(m,n)-1* times. Because *C(n,n) = 1*, the complexity of the algorithm is *O(C(m,n))* using *m* processors. The *cost* of the algorithm, which is the parallel running time multiplied by the number of processors used, matches the lower bound of *Ω(mC(m,n))* operations required by any sequential combinations generation algorithm. The algorithm is cost optimal but not adaptive. Table 9 presents an example for $n = 5$ and $m = 3$. Actual assignments are shown

in this table and, in column C, f stands for *false* and t for *true*. Throughout the execution, the content of A is *3 4 5*.

Table 9. Operation of algorithm PCLCA for $n = 5$ and $m = 3$. [26]

After step	*B*			*C*			*D*			*k*
1	1	2	3	f	f	f	1	2	3	
2										0
3										
4.1	1	2	4							
5				f	f	f	1	2	4	
2										0
3										
4.1	1	2	5							
5				f	f	t	1	2	5	
2										0
3.1	1	3	5							3
4.2	1	3	4							
5				f	f	f	1	3	4	
2										0
3										
4.1	1	3	5							
5				f	f	t	1	3	5	
2										0
3.1	1	4	5							3
4.2	1	4	5							
5				f	t	t	1	4	5	
2										0
3.1	2	4	5							2
4.2	2	3	4							
5				f	f	f	2	3	4	
2										0
3										
4.1	2	3	5							
5				f	f	t	2	3	5	
2										0
3.1	2	4	5							3
4.2	2	4	5							
5				f	t	t	2	4	5	
2										0
3.1	3	4	5							2
4.2	3	4	5							
5				t	t	t	3	4	5	

Algorithm stops because C(1) = true.

3.5.4 Akl [28]

In [28], a parallel algorithm for generating all *(m,n)*-combinations is presented. The algorithm runs on a SIMD machine with k processors where $1 \leq k \leq C(m,n)$. In this algorithm, simultaneous reads are not needed and the shared memory is never used. In fact the k processors, once started independently, execute the same algorithm and never need to communicate among themselves. This is the model used by Gupta in [21]. Each processor operates on a different data set stored in its local memory. The model precludes any communication among processors. There is no shared memory or interconnection network available. It can be regarded as a restricted version of the SIMD model of computation. The algorithm works on the set of n integers $S = \{1, 2, \ldots, n\}$ and is based on the following assumptions:

1. An algorithm SEQUENTIAL COMBINATIONS is available for the sequential generation of all *C(m,n)* *(m,n)*-combinations in lexicographic order. The running time of such an algorithm is *O(mC(m,n))*.

2. A system for numbering all *C(m,n)*-combinations is available. Let $X = x_1x_2\ldots x_m$ be a combination of m integers out of S. RANKC is a function which associates with each such combination X a unique integer RANKC*(X)*. This function, explained in detail in chapter 2, has the following properties: 1) it preserves lexicographic ordering; 2) its range is the set $\{1, 2, \ldots, C(m,n)\}$; and 3) it is invertible such that if d = RANKC*(X)* then X can be obtained from UNRANKC*(d)*. Both RANKC*(X)* and UNRANKC*(d)* have running time *O(mn)*, as described earlier.

The algorithm makes use of algorithm SEQUENTIAL COMBINATIONS and of function UNRANKC and is shown in ALGORITHM 9.

ALGORITHM 9. PCLA(n,m) [28].
BEGIN
  For i = 1 **To** k **Do in parallel**
    1. Let $j = (i - 1)\lceil C(m,n)/k \rceil + 1$
    2. **If** $j \leq C(m,n)$ **Then**
      1. Obtain the jth combination from UNRANKC(j), call this combination Q_i.
      2. Starting with Q_i, use algorithm Sequential combinations until a maximum of $\lceil C(m,n)/k \rceil - 1$ of the next combinations have been generated.
    Endif
  Endfor
END.

Step *1* requires *O(m)* operations. In step 2.1 *O(mn)* operations are needed to produce Q_i. It takes on the order of $m(\lceil C(m,n)/k \rceil - 1)$ operations to generate the $\lceil C(m,n)/k \rceil - 1$ subsequent combinations. The overall running time of the algorithm is therefore dominated by $max\{O(mn), O(m\lceil C(m,n)/k \rceil)\}$ [28]. Hence, the algorithm has an optimal cost of $O(mC(m,n))$ when $1 \leq k \leq C(m,n)/n$. The algorithm is capable of modifying its behavior according to the number of processors actually available; therefore, it is adaptive.

3.5.5 Lin [29]

In [29], a parallel algorithm for generating all *(m,n)*-combinations in lexicographic order is described. This algorithm is ***very complex*** and is briefly introduced in this section. The computational model used by the algorithm is a linear systolic array consisting of *m* identical processing elements (PEs) that operate in synchronous. The PEs do not need to know their indexed positions during execution. Only adjacent PEs can communicate their data via communication links. There exists a communication link from PE1 to PE2 for

sending data from PE1 to PE2. This link is called an input link of PE2 and an output link of PE1. Each PE performs the following three tasks: 1) Receiving data from its *four* input of communication links, 2) Executing the functions that are described by an existing algorithm and 3) Sending data to its *four* output of communication links. The needed time units to do these three tasks is specified as *time-step*. The algorithm works on the set of n integer $S = \{1, 2, \ldots, n\}$. The time complexity of the algorithm is $O(C(m,n))$ to generate all the $C(m,n)$ combinations. The algorithm is optimal but not adaptive. Since the processing elements are identical and executes the same procedure, the algorithm is suitable for VLSI implementations [29].

3.5.6 Akl, Gries and Stojmenovic [31]

In [31], a parallel algorithm for generating in lexicographic ascending order all the (m,n)-combinations of the set $S = \{1, \ldots, n\}$. The algorithm runs on a linear array of m processors, where each processor communicates only with its left and right neighbors. Each processor has constant size memory and is responsible for producing one element of each combination. The kth combination is denoted by $X^{(k)} = \{x_1^{(k)}, x_2^{(k)}, \ldots, x_m^{(k)}\}$ where $1 \leq k \leq C(m,n)$ and $1 \leq x_1^{(k)} < x_2^{(k)} \ldots < x_m^{(k)} \leq n$. The algorithm uses m shared variables $D_1, \ldots, D_m$. Each processor i is responsible for maintaining $D_i = x_i^{(k)}$ and reads only D_{i-1} and D_{i+1}. In addition, each processor i has a ***while*** loop that at each iteration prints $x_i^{(k)}$, so that each iteration will print $X^{(k)}$. In the calculation, processor i is one step ahead of processor $i+1$, as shown in Figure 3 for $n = 5$ and $m = 3$. In this figure, a sequence of lines connecting values indicates the values that are in the variables D_i at each step.

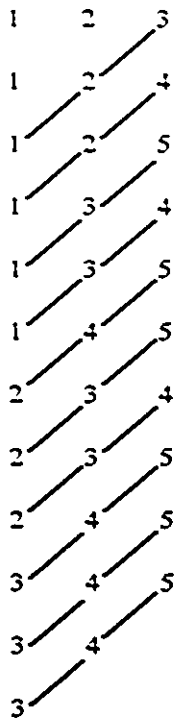

Figure 3. Combinations in lexicographic order for $n = 5$ and $m = 3$.

Initially, we have $D_1 = x_1^{(3)} = 1$, $D_2 = x_2^{(2)} = 2$, and $D_3 = x_3^{(1)} = 3$. This means when D_m is $x_m^{(C(m,n))}$, processors $1...m-1$ will have calculated further values. These further values as well as $x_0^{(k)}$ and $x_{m+1}^{(k)}$ are defined as follows: $x_0^{(k)} = n-m$ for $1 \leq k$; $x_{m+1}^{(k)} = n$ for $1 \leq k$; and $x_i^{(k)} = x_i^{(C(m,n))} = n-m+i$ for $k > C(m,n)$ and $1 \leq i \leq m$. Introducing D_0 and D_{m+1}, the first invariant of the loops of processors $1...m$, is introduced as follows [31]:

P0: $1 \leq k \leq C(m,n)+1$ **and**
the (m,n)-combinations $X^{(1)}, \ldots, X^{(k-1)}$ **are printed and**
$D_i = x_i^{(k+m-i)}$ for $0 \leq i \leq m+1$;

The key to the algorithm is the efficient calculation of each D_i at each step, when k is increased by *1*. The calculation is accomplished by Lemma 1.

Lemma 1. The following formula describes $x_i^{(k+1)}$ for $1 \le i \le m$:

$$x_i^{(k+1)} = \begin{cases} x_i^{(k)} = n-m+i \rightarrow x_{i-1}^{(k+1)} + 1 \\ x_i^{(k)} < n-m+i \wedge y_i^{(k)} \rightarrow x_i^{(k)} + 1 \\ x_i^{(k)} < n-m+i \wedge \neg y_i^{(k)} \rightarrow x_i^{(k)} \end{cases}$$

where $y_i^{(k)} = x_{i+1}^{(k-h)} = n\text{-}m+i$, for all h, $1 \le h \le m\text{-}i$.

The proof of Lemma *1* is given in [31]. P0 defines the shared variables $D_{0\ldots m+1}$, with each processor i, $1 \le i \le m$, maintaining D_i, D_0 and D_{m+1} remaining constant. Other values maintained by processor i must be determined as follows. *First* at iteration k, processor i prints $x_i^{(k)}$. However, D_i has the value $x_i^{(k+m-i)}$. In order to print $x_i^{(k)}$, processor i will maintain the whole range of values $x_i^{(k)} \ldots x_i^{(k+m-i)}$. *Second*, one should investigate what processor $i+1$ requires from processor i in order to calculate its next value D_{i+1} at each iteration. By P0, since k is to be increased by *1*, this next value is $x_{i+1}^{(k+1+m-(i+1))}$, i.e. $x_{i+1}^{(k+m-i)}$. By Lemma *1*, this calculation may require reference to $x_i^{(k+m-i)}$, so processor i will maintain it. *Third*, one should investigate what processor $i\text{-}1$ may require from processor i in order to calculate its next value D_{i-1}. By P0, this next value is $x_{i-1}^{(k+1+m-(i-1))}$. Evaluation of this may require evaluation of $y_{i-1}^{(k+m-(i-1))}$ of Lemma *1*, which references the value $x_i^{(k+1)} \ldots x_i^{(k+m-i)}$. Therefore, processor i will maintain it. Combining the three results,

one can conclude that processor i must maintain $x_i^{(k)} \ldots x_i^{(k+m-i)}$. As shown in the second invariant of the loop for processor i, P1, processor i uses five local variables D_i, r_i, $d1_i$, $r1_i$, and $d2_i$ to maintain $x_i^{(k)} \ldots x_i^{(k+m-i)}$:

P1: $D_i = d1_i = d2_i$ and
$x_i^{(k)} \ldots x_i^{(k+m-i)}$ is a suffix of the sequence $d2_i$.
($d1_i$ repeated $r1_i$ times). (D_i repeated r_i times).

The termination condition occurs when $k = C(m,n)+1$. By P1, this condition is written as $D_i = n-m+i$ and $r_i > m$. The algorithm is shown in ALGORITHM 10.

```
ALGORITHM 10. PCLAGS(n, m) [31].
BEGIN
  D0 = n-m; Dm+1, rm+1 = n, 0;
  Forparallel i = 1 To m Do
     Di, ri = i, m-i+1; j = 1;
     Do ¬(Di = n-m+i and ri > m)
        If ri > m-i Then
          print Di;
        Else If ri + r1i > m-i Then
          print d1i;
        Else print d2i;
        x = Di+1; y = Di+1; ry = ri+1;
        If Di = n-m+i Then
          If Di = x+1 Then
             ri = ri +1;
          Else
             d2i = d1i;
             d1i, r1i = Di, ri;
             Di, ri = x+1, 1;
          EndIf
        Else If y = n-m+i ∧ ry = m-i Then
             d2i = d1i;
             d1i, r1i = Di, ri;
             Di, ri = Di+1, 1;
          Else ri = ri+1;
        j = j+1;
     Enddo
  Endfor
END.
```

The time complexity of the algorithm is $O(mC(m,n))$. The algorithm is optimal and produces the combinations in constant time per combination and requires no integer larger than n during computation, assuming the time to output the combination is counted. The algorithm can be made adaptive if the array is divided into N/m groups of m processors each

(assuming the linear array is made of N arbitrary number of processors) such that each group produces an interval of consecutive *(m,n)*-combinations.

3.5.7 Stojmenovic [33]

In [33], a systolic algorithm is described for generating, in lexicographically ascending order, all combinations of m elements from the set of n elements $S = \{1, \ldots, n\}$. The algorithm is designed to be executed on a linear array of m processors, each having constant size memory, and each being responsible for producing one element of a given combination. As shown in chapter I, a correspondence between *(m,n)*-combinations and combinations with repetitions of m out of $n-m+1$, *(m,n-m+1)-r_*combination, can be established. To generate the *(m,n)-r_*combinations, the algorithm uses a linear array of m processors, indexed from 1 to m, and m variables $x_1, \ldots, x_m$, where $1 \leq x_1 \leq x_2 \leq \ldots \leq x_m \leq n$. Each processor i is responsible for maintaining x_i by reading only data from processor $i-1$ and $i+1$. The processors act in ***lock-step*** fashion, and each step produces a new *(m,n)-r_*combination.

Unless they are advised to change their output, processors 1 to $m-1$ will always produce the same element. Processor m produces elements between x_{m-1} and n (this is called a current run of processor m). Let $x_0 = n$ and $x_{m+1} = n$. Processor t (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 whenever $x_t = n-1$ and $x_{t-1} < n-1$. 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. In fact, the message s_i contains the value of x_t.

$_t+1$ 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 Table 10, the message path is marked in bold for $n = 6$ and $m = 4$.

Table 10. Combinations with and without repetitions for $n = 6$ and $m = 4$.

combinations	r_combinations	s1w1	s2w2	s3w3	s4w4
1 2 3 4	1 1 1 1	0 0	0 0	0 0	0 0
1 2 3 5	1 1 1 2	0 0	0 0	0 0	2 2
1 2 3 6	1 1 1 3	0 0	0 0	0 0	2 1
1 2 4 5	1 1 2 2	0 0	0 0	2 3	0 0
1 2 4 6	1 1 2 3	0 0	0 0	2 2	2 2
1 2 5 6	1 1 3 3	0 0	0 0	2 1	2 1
1 3 4 5	1 2 2 2	0 0	2 4	0 0	0 0
1 3 4 6	1 2 2 3	0 0	2 3	2 3	0 0
1 3 5 6	1 2 3 3	0 0	2 2	2 2	2 2
1 4 5 6	1 3 3 3	0 0	2 1	2 1	2 1
2 3 4 5	2 2 2 2	4 5	0 0	0 0	0 0
2 3 4 5	2 2 2 3	4 4	4 4	0 0	0 0
2 3 5 6	2 2 3 3	4 3	4 3	4 3	0 0
2 4 5 6	2 3 3 3	4 2	4 2	4 2	4 2
3 4 5 6	3 3 3 3	4 1	4 1	4 1	4 1
		0 0	0 0	0 0	0 0

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. The current run of processor m consists of m-t+2 combinations ending with $x_t \ldots x_m = n\text{-}1, n\text{-}1, \ldots, n\text{-}1, n, \ldots, n$. 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. When $x_1 = n\text{-}1$ 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. Table 10 also presents the dynamic of variables s_i and w_i. The algorithm is shown in ALGORITHM 11.

```
ALGORITHM 11. PCLS(n,m) [33]
BEGIN
  For i ← 1 to m do in parallel
    x_i ← 1; x_0 ← n; x_{m+1}  n; w_i ← 0; s_i ← 0;
    Repeat
      output x_i+i-1;
      If w_i ≥ 1 Then
        w_i ← w_i-1;
      If s_i ← 0 and s_i-1 ≠ 0 Then
        s_i ← s_i-1;
        w_i ← m-i+2;
      If x_i+1 ← n and x_i < n Then
        x_i ← x_i+1;
      If w_i ← 0 and s_i ≠ 0 Then
        x_i ← s_i;
        s_i ← 0;
      If x_i ← n-1 and x_i-1 ≠ n-1 and s_i ← 0 Then
        s_i ← x_i-1+1;
        w_i ← m-i+2;
    Until x_i ← n+1
  EndFor
END.
```

The algorithm generates all *(m,m+n-1)*-combinations in lexicographic order. If the current output is simply x_i instead of x_i+i-1, it will generate the *(m,n)-r_*combinations. There is a constant delay per combination, leading to an *O(C(m,n))* time solution. The algorithm is cost optimal (assuming the time to output the combinations is counted), and does not deal with very large integers, such as *C(m,n)*. Each processor has a constant size memory. The algorithm is much simpler than the previously known solutions that enjoy the same properties. In addition, it can be made adaptive and, at the same time, keeps all desirable properties.

3.5.8 New numbering system [17]

In [17], a function mapping the interval [0..1) into the set of *(m,n)*-combinations is described. This mapping can be used for generating the *(m,n)*-combinations at random, with equal probability of each combination to be chosen. Such method is also used for generating combinations in [35]; however, the method is not practical because it requires very large integers. This is not a desirable property. The technique presented in [17] avoids the use of very large integers. Once a random number g in [0..1) is taken, it is mapped into the set of *(m,n)*-combinations by a function $f(g)$. The method finds the combination X such that the ratio of the number of *(m,n)*-combinations that precede X and the total number of *(m,n)*-combinations is $\leq g$. In other words, it finds the *(m,n)*-combination $f(g)$ with the ordinal number $\lfloor gC(m,n) \rfloor + 1$. This technique will be useful in the design of an ***adaptive parallel*** algorithm for generating all *(m,n)*-combinations.

Let $x_1x_2...x_m$ be a *(m,n)*-combination of the set $\{1, 2, ..., n\}$ such that $1 \leq x_1 < x_2 ... < x_m \leq n$. The number of such combinations is $C(m,n)$. Suppose the first $k-1$ elements in given *(m,n)*-combination are fixed, i.e. $x_i = a_i$, $1 \leq i < k$. These are called *(k-1)*-fixed *(m,n)*-combinations. Possible values for x_k are $b_1 = a_{k-1}+1$, $b_2 = a_{k-1}+2$, ..., $b_h = n$ (thus $h = n-a_{k-1}$).

Lemma 1. The ratio of the number of *(k-1)*-fixed *(m,n)*-combinations for which $x_k = j$ and the number of *(k-1)*-fixed combinations for which $x_k \geq j$ is $(m-k+1)/(n-j+1)$ whenever $j > a_{k-1}$. (proof is given in [17])

Let $z = j-a_{k-1}$. Also, let $S(k,z)$ and $S(k,\geq z)$ denote the ratio of the number of $(k-1)$-fixed (m,n)-combinations for which $x_k = b_z$ (and $x_k \geq b_z$ respectively) and the number of $(k-1)$-fixed (m,n)-combinations. Then, from Lemma 1 it follows that $S(k,z)/S(k,\geq z) = (m-k+1)/(n-z-a_{k-1}+1)$ for the case of (m,n)-combinations, and the procedure, SCRS, is obtained, which finds the (m,n)-combination with ordinal number $\lfloor gC(m,n) \rfloor+1$. The procedure is shown in ALGORITHM 12.

```
ALGORITHM 12. SCRS(m,n,g) [17].
BEGIN
   j ← 0; q ← 0;
   For k ← 1 To m Do
     j ← j+1;
     p ← (m-k+1)/(n-j+1);
     While (p ≤ g) Do
        q ← p;
        j ← j+1;
        p ← q+(1-q)(m-k+1)/(n-j+1)
     Endwhile
     x_k ← j;
     g ← (g-q)/(p-q);
   EndFor
END.
```

The relationship between combinations without repetitions and combinations with repetitions was presented in detail in Chapter II. Let $y_i = x_i - i+1$. Then $1 \leq y_1 \leq y_2 \leq \ldots \leq y_m \leq n-m+1$, and $y_1y_2\ldots y_m$ is a combination with repetitions of m out of $n-m+1$ elements. Because of this simple relation, the procedure SCRS(m,n,g) can be used to generate random combinations with repetitions by simply replacing the instruction $x_k \leftarrow j$ with $y_k \leftarrow j-k+1$.

The advantage of this method is that it can be applied to both random combinations generation and dividing all (m,n)-combinations into desirable sized groups. The latter will be useful for designing ***adaptive*** algorithms for generating all (m,n)-combinations on a parallel model of computation. The time complexity of the algorithm is $O(n)$.

3.6 Comparison of parallel algorithms

In this section, the characteristics of the parallel combination generation algorithms, presented in the previous section, are summarized in Table 11. This table contains, for each algorithm, the time complexity, the processor complexity, the memory required per processor, the largest number that each processor must store, and the model of computation required. In addition, the table indicates whether the algorithm: 1) is adaptive or not, 2) is cost optimal or not, 3) lexicographic or not, and 4) generates the combinations from a set of arbitrary elements or not. The algorithms are listed in the table in chronological order. Referring to Table 11, one may discuss the differences among these algorithms as follows. The algorithms in [21,22] are not optimal and do not run on a linear array of processors. However, they produce the combinations in lexicographic order. On the other hand, algorithm [26] is cost optimal and generates combinations in lexicographic order as well. In [28], the algorithm uses an arbitrary number of independent processors, each producing an interval of consecutive combinations. The algorithm is cost optimal and lexicographic, but each processor requires memory of size $O(m)$ and has to deal with large integers. The algorithms in [21,26] use the CREW PRAM computation model but [26] is both cost optimal and lexicographic, the largest number is of order $O(n)$ and the memory required per processor is of order $O(1)$. In [29, 31], each processor is responsible for producing an element of the combination and all of them use a linear array of processors. The algorithms in [29] are lengthy and complex but they are cost optimal and lexicographic. However, [31] is concise and simple as well as cost optimal and lexicographic. There are only four

Table 11. Comparison of parallel algorithms for generating combinations.

Author(s)	Time complexity	Processor complexity	Memory per proc.	Largest number	Cost optimal	Adaptive	Lexicographic	Arbitrary Element	Model
Gupta & Bhatt. [21]	mlog(n-m+1)	C(m,n)	$O(n^2)$	O(C(m,n))	N	N	Y	N	CREW PRAM
Tang, Du & Lee [22]	O(mC(m,n)/p)	$p \le C(m,n)$	O(m)	O(C(m,n))	N	Y	Y	N	p independent
Chan & Akl [26]	O(C(m,n))	m	O(1)	O(n)	Y	N	Y	N	CREW RAM
Chen & Chern [27]	$O(\frac{n}{p}\sum_{i=0}^{k-2}(\frac{n!}{(n-i-1)!}))$	$p \le n$	O(k)[a]	O(n)	Y	Y	N	N	m-linear array[b]
Akl [28]	$O(m\lceil C(m,n)/p \rceil)$	$p \le C(m,n)/n$	O(m)	O(C(m,n))	Y	Y	Y	N	p independent
Er [37]	$O((m^2C(m,n))/p)$	$p \le n$	O(m)	O(n)	N	Y	Y	N	p independent
Lin [29]	O(C(m,n))	m	O(1)	O(n)	Y	N	Y	N	m-linear array
Akl,Gries & Sto. [31]	O(C(m,n))	$m \le p$	O(1)	O(n)[c]	Y	Y	Y	N	m-linear array
stojmenovic [33]	O(C(m,n))	m	O(1)	O(n)	Y	Y	Y	N	m-linear array
Elhage & Sto. [49]	O(C(m,n))	m	O(1)[d]	O(n)	Y	Y	Y	Y	m-linear array

[a] Algorithms produce all subsets of up to k elements.
[b] A selector is also used in the model.
[c] When used adaptively, the algorithm requires numbers $O(C(m,n))$.
[d] Except processor m which has $O(n)$ memory.

algorithms from the list presented in Table 11 that are adaptive [22,28,31, 33]. But only [33] presented an algorithm that is cost optimal, adaptive, lexicographic and runs on a linear array of processors, and therefore satisfies most of the desirable properties presented at the beginning of this chapter. However, this algorithm is restricted to elements that are *integers*.

Chapter IV

Systolic Generation of Combinations From Arbirary Elements

4.1 General

In this chapter, the generation of the $C(m,n)$ combinations of m objects chosen from an arbitrary set $\{p_1, p_2, \ldots, p_n\}$ in lexicographical ascending order is investigated. These combinations are referred as (m,n)-combinations. As presented in earlier chapters, many sequential algorithms were introduced to solve this problem [5, 6, 7, 8, 9, 10, 11, 13, 15, 16, 22]. Since each of these combinations requires $O(m)$ time to be produced as output, the best possible sequential algorithm runs in $O(mC(m,n))$ time when the time to output the combinations is taken into account. Otherwise, there are algorithms for generating combinations without producing them as output, whose running time is $O(C(m,n))$.

As presented in chapter III, the algorithms for generating combinations in parallel have been studied extensively in [19, 20, 21, 22, 26, 27, 28, 29, 31, 33]. The algorithms in [29, 31, 33] satisfy some of the desirable properties presented at the beginning of chapter III. These properties are the following: 1) the combinations are listed in lexicographic order, 2) the algorithm is cost optimal, 3) the time required by the algorithm between any two consecutive objects it produces is constant, 4) the model of parallel computation is as simple

as possible, i.e. a linear array of m processors, and 5) each processor needs as little memory as possible and is capable of storing an integer no larger than n. The algorithms presented in [21, 22, 27, 28] do not meet all these properties. In algorithms [29, 31, 33], each processor is responsible for producing one element of each combination. The algorithms in [29] are rather lengthy and sophisticated while the algorithms in [31, 33] are rather concise. While algorithms [29, 31, 33] satisfy properties 1-5, none of them satisfies the following property:

6) The algorithm generates combinations of a set of arbitrary elements $\{p_1, p_2, \ldots, p_n\}$, on which an order relation $<$ is defined such that $p_1 < p_2 < \ldots < p_n$.

This property is important as it allows us to distinguish between algorithms that generate combinations of any ordered set, and algorithms that can only generate combinations of the set $\{1, 2, \ldots, n\}$, which is the case with algorithms [29, 31, 33].

In this chapter a new combinations generation technique that satisfies all desirable properties 1-6, except a *minor* modification to the architecture, is presented.

4.2 The model architecture

The algorithm uses a linear array of m processors, indexed 1 to m. However, 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. It should be indicated that in practice the network models of parallel computation usually have a master processor that distributes the job and/or data to the other ones, and has a clock to synchronize the execution of all processors. From that point of view the model presented

here is not a restriction with respect to 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.

4.3 Parallel generation of combinations without repetitions

An (m,n)-combination can be represented as a sequence $c_1c_2\ldots c_m$ where $p_1 \leq c_1 < c_2 < \ldots < c_m \leq p_n$. Let $z_1z_2\ldots z_m$ 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. Each processor i is responsible for maintaining c_i and z_i by reading only data from processor $i-1$ and $i+1$. The processors act in ***lock-step*** fashion (i.e. all processors perform the same instructions at any given time), and each step produces a new (m,n)-combination.

The well known sequential algorithm for generating (m,n)-combinations [6] determines the next combination by a backtrack search that finds an element c_t with the greatest possible t such that $z_t < n-m+t$. Processor t is called the ***turning point***. It should be noted 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 during backtracking, one may consider two cases of the backtrack search, as shown in the following.

a) Case 1: $z_t = n\text{-}m+t\text{-}1$.

This is the next to the maximal possible value of the index z_t. Since t is the turning point $z_{t+1} = n\text{-}m+t+1$, 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.

b) Case 2: $z_t \neq n\text{-}m+t\text{-}1$.

In this case the system is supposed to perform a ***major change***. In order to achieve a constant delay in this case, it is decided to start the backtrack search in advance, such that the new values of all indices z_i and element c_i are known at the time when major changes in the system are due. It should be noted that between two major changes the system undergoes a series of minor ones.

Figure 4 shows the indices z_t, z_{t+1}, z_{t+2}, ..., z_{m-1}, z_m of combinations between two major changes. The indices that actually perform minor changes are marked in bold. These indices are $n\text{-}m+t$, $n\text{-}m+t+1$, $n\text{-}m+t+2$, ..., $n\text{-}3$, $n\text{-}2$ and $n\text{-}1$. Also, one may indicate that the turning points for the first and last combinations in the figure are processors $t+1$ and t,

respectively. The corresponding indices of elements, z_t and $n\text{-}m+t\text{-}1$ for processors t and $t+1$, are underlined.

t	$t+1$	$t+2$	$t+3$	...	$m\text{-}2$	$m\text{-}1$	m	*Processors*
z_t	<u>n-m+t-1</u>	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	**n-1**	*minor change*
z_t	n-m+t	n-m+t+1	n-m+t+2	...	n-3	**n-2**	n	*minor change*
z_t	n-m+t	n-m+t+1	n-m+t+2	...	**n-3**	n-1	n	*minor change*
	. . .							
	. . .							
z_t	n-m+t	n-m+t+1	**n-m+t+2**	...	n-2	n-1	n	*minor change*
z_t	n-m+t	**n-m+t+1**	n-m+t+3	...	n-2	n-1	n	*minor change*
z_t	**n-m+t**	n-m+t+2	n-m+t+3	...	n-2	n-1	n	*minor change*
<u>z_t</u>	n-m+t+1	n-m+t+2	n-m+t+3	...	n-2	n-1	n	***major change***

Figure 4. Indices between two major change.

For the special case when $t = m\text{-}1$, Figure 4 will be reduced to the following:

z_{m-1}	<u>n-2</u>
z_{m-1}	**n-1**
z_{m-1}	n

There are at least $m\text{-}t$ minor changes that will take place before a major change with turning point in processor t. In order to perform the major change in the last combination of Figure 4, it was decided to prepare the data starting the process with the second one. More precisely, whenever $z_m = n\text{-}1$ processor m activates the search for the turning point. First a message is sent toward the turning point to find it; this will take $m\text{-}t$ 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 figure, 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.***

Figure 5. Indices after a major change.

The data $p(z_t+1)$, ..., $p(z_t+m-t+1)$ are broadcast from processor m in a pipelined fashion, using the links of the linear array of processors. The data path is illustrated in Figure 6.

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

Figure 6. The data path.

The broadcast presented in Figure 6 takes another m-t steps. Therefore the total number of steps necessary to prepare data for major change in the system is $3(m-t)$. These are to be done during m-t minor changes. This would clearly be 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 is done. This is the case of the second combination in Figure 6. To avoid precise calculation of the number of message steps versus the number of minor changes, a simpler criterion is used 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, PCLES, is coded in the next page.

Summarizing, one may propose the following theorem which is clearly derived from the detailed description presented earlier.

Theorem. The described algorithm generates all combinations of m objects chosen from $\{p_1, p_2, \ldots, p_n\}$ in lexicographic order and with constant delay per combination on a linear array of m processors, thus achieving an optimal cost of $O(mC(m,n))$ (assuming the time to output the combinations is counted). Furthermore, each processor has a memory of constant size, except processor m which has $O(n)$ memory, and can generate elements without the need to deal with large integers such as $C(m,n)$.

ALGORITHM. PCLES(n, m). *(without repetitions)*
BEGIN

```
1.  For i ← 1 To m Do in parallel {
2.      read p_i, p_{n-m+i}, p_{n-m+i-1};
3.      z_i ← i; c_i ← p_i; t_i ← 0; x_i ← 0; v_i ← 0; cn_i ← 0;
4.      Repeat {
5.          output c_i;
6.          If z_i = n-m+i Then cn_i ← cn_i+1;   /* cn counts repetitions of the maximal value in i */
7.          If i = m and z_i = n-1 Then  t_i ← -1;   /* initiate search for turning point t  */
8.          For j ← 1 to 3 Do {
9.              If i < m and t_i = 0 and t_{i+1} = -1 Then {
10.                 t_i ← -1;      /* continue search for t */
11.                 t_{i+1} ← 0;
12.                 If z_i < n-m+i-1 Then {
13.                     x_i ← z_i;     /* turning point t found */
14.                     t_i ← i; }
15.                 Else If i = 1 Then t_i ← 2n; }   /* initiate termination message */
16.             If i > 1 and t_i = 0 and t_{i-1} > 0 Then {
17.                 t_i ← t_{i-1};
18.                 x_i ← x_{i-1}; }
                /* distribute t and z_i to proc. t, t+1,..., n; they are saved as t_i and x_i, resp. */
19.             If i = m and t_i > 0 and v_i = v_{i-1} and x_i < n Then {
20.                 x_i ← x_i+1;
21.                 v_i ← x_i;
22.                 r_i ← p(v_i); }
                /* generating new values, for next major change, for processors t, t+1, ..., m */
23.             If i < m and t_i > 0 and v_{i+1} > 0 and v_i ≠ x_i+i-t_i+1 Then {
24.                 r_i ← r_{i+1};
25.                 v_i ← v_{i+1}; }}
            /* pipeline to the left new values v_i and r_i for z_i and c_i, respectively */
26.         If i < m and z_i = n-m+i-1 and z_{i+1}= n-m+i+1 Then {
27.             z_i ← n-m+i;
28.             c_i ← p_{n-m+i} }
29.         If i = m and z_i < n Then {      /* minor change */
30.             z_i ← z_i+1;
31.             c_i ← p(z_i); }        /* minor change in processor m */
32.         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 {
33.             z_i ← v_i;
34.             c_i ← r_i;
35.             t_i ← 0;
36.             x_i ← 0;
37.             v_i ← 0;
38.             cn_i ← 0; }          /* major change  */
        Until  t_i = 2n and cn_i = i }}
END.
```

4.4 Example using algorithm PCLES

The algorithm is applied to generate the combinations for $n = 6$ and $m = 4$ from the set $P = \{A, B, C, D, E, F\}$. First, in this example, the communication steps among the processors throughout the execution of the algorithm is presented in Figure 7. Next, the values for the variables that have changed during the execution are shown in each step in Table 12. The content of each variable in each processor may change several times in one step. All new values that are assigned to each variable that changes in one step are shown in Table 12 after executing each instruction. Once a step is completed, the content of the variable in each processor is the value on the right of the corresponding cell. The number of communication steps needed is equal to three times the number of minor changes between two major changes. The variables used by the algorithm and shown in Table 12 are:

z_i = Index of an element in P used in processor i.
c_i = Element of a combination produced by processor i, $c_i = p(z_i)$.
t_i = Turning point processor number used in processor i.
x_i = Index of the element in the turning point used in processor i.
v_i = Index of the element for major change used in processor i.
r_i = Element of a combination to be produced during a major change by processor i.
cn_i = Counter for repetitions of the maximal value in processor i.

To find the turning point t, a message is sent to the left by processor $m = 4$. Once the turning point t is found, 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. Finally, processor m sends the new data v_i and r_i to all processors between t and m in a pipelined fashion. The process is illustrated for this example in the following.

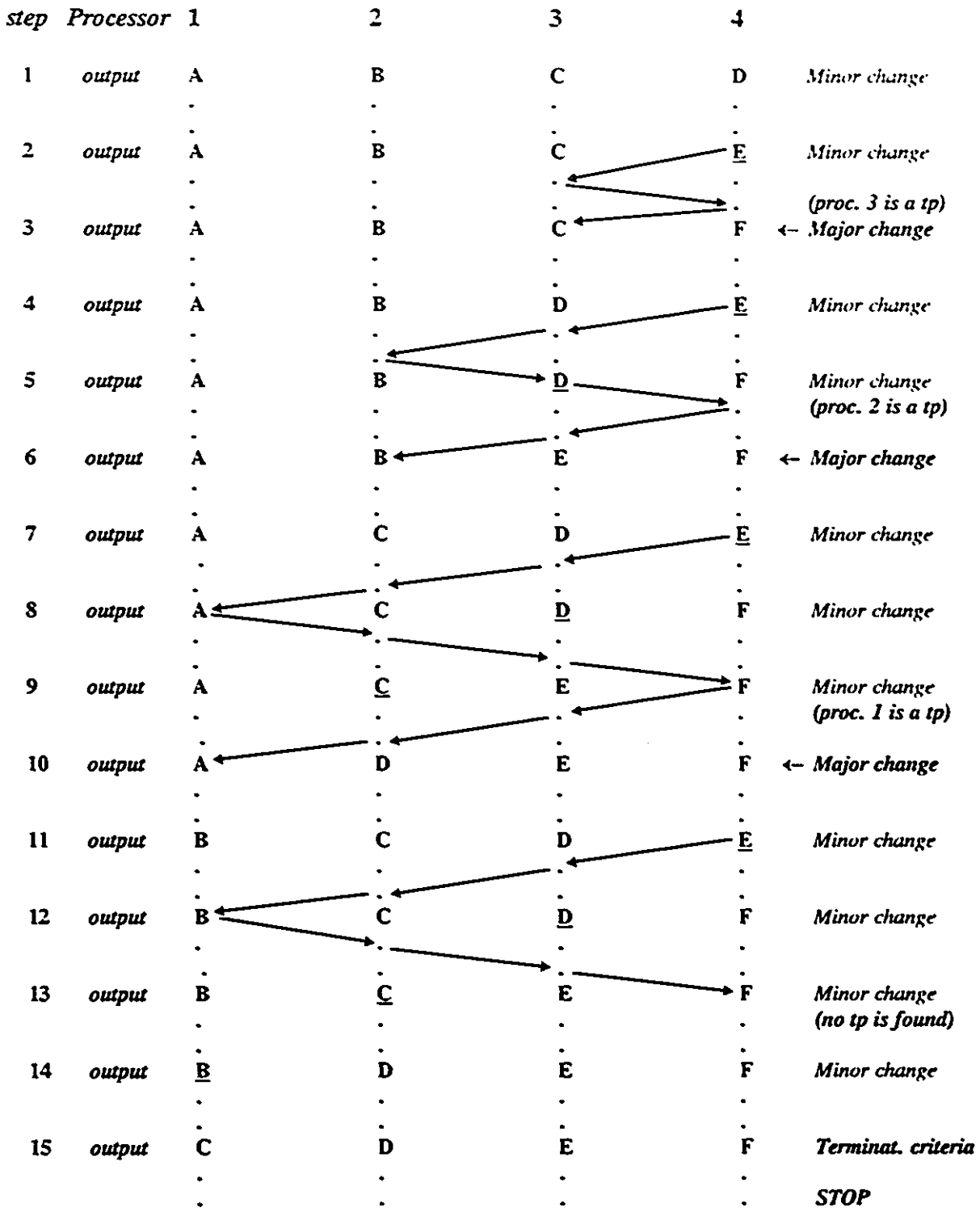

Figure 7. Communication steps using algorithm PCLES.

Table 12. Algorithm PCLES execution steps for $n = 6$ and $m = 4$.

Proc. i	P_i	P_{n-m+i}	$P_{n-m+i-1}$	$n-m+i$
1	P_1=A	P_3=C	P_2=B	3
2	P_2=B	P_4=D	P_3=C	4
3	P_3=C	P_5=E	P_4=D	5
4	P_4=D	P_6=F	P_5=E	6

Step	*Processor*	1	2	3	4	*Comments*
1	*output*	A	B	C	D	
1.1		t_1=0 x_1=0 v_1=0 r_1= cn_1=0 z_1=1 c_1=A	t_2=0 x_2=0 v_2=0 r_2= cn_2=0 z_2=2 c_2=B	t_3=0 x_3=0 v_3=0 r_3= cn_3=0 z_3=3 c_3=C	t_4=0 x_4=0 v_4=0 r_4= cn_4=0 z_4=4 c_4=D	*Initialization*
1.2						
1.3					z_4=4.5 c_4=E	
2	*output*	A	B	C	E	*Minor change (proc. 3 is tp)*
2.1				t_3=0,-1,3 x_3=0,3 v_3=0,4 r_3=D	t_4=0,-1,0,3 x_4=0,3,4 v_4=0,4 r_4=D	
2.2					x_4=4,5 v_4=4,5 r_4=D,E	
2.3					z_4=5,6 c_4=E,F	
3	*output*	A	B	C	F	*Major change*
3.1					cn_4=0,1	
3.2						

3.3				t_3=3.0 x_3=3.0 v_3=4.0 z_3=3.4 c_3=C.D	t_4=3.0 x_4=4.0 v_4=4.0 z_4=6.5 c_4=F.E cn_4=1.0	
4	*output*	A	B	D	E	*Minor change*
4.1				t_3=0.-i	t_4=0.-1.0	
4.2			t_2=0.-1.2 x_2=0.2	t_3= -1.0.2 x_3=0.2		
4.3					t_4= 0.2 x_4=0.2.3	*(proc. 2 is tp)*
				v_3=0.3 r_3=D.C	v_4=0.3 r_4=E.C z_4=5.6 c_4=E.F	
5	*output*	A	B	D	F	*Minor change*
5.1					cn_4=0.1 x_4=3.4	
			v_2=0.3 r_2=C	v_3=3.4 r_3=C.D	v_4=3.4 r_4=C.D	
5.2					x_4=4.5 v_4=4.5 r_4=D.E	
5.3				z_3=4.5 c_3=D.E		
6	*output*	A	B	E	F	*Major change*
6.1				cn_3=0.1	cn_4=1.2	
6.2						
6.3			z_2=2.3 c_2=B.C t_2=2.0 x_2=2.0 v_2=3.0	z_3=5.4 c_3=E.D t_3=2.0 x_3=2.0 v_3=4.0 cn_3=1.0	z_4=6.5 c_4=F.E t_4=2.0 x_4=5.0 v_4=5.0 cn_4=2.0	

7	*output*	A	C	D	E	*Minor change*
7.1				t_3=0,-1	t_4=0,-1,0	
7.2			t_2=0,-1	t_3= -1,0		
7.3		t_1=0,-1,1 x_1=0,1	t_2= -1,0,1 x_2=0,1			
					z_4=5,6 c_4=E,F	

8	*output*	A	C	D	F	*Minor change*
8.1					cn_4=0,1	
				t_3=0,1 x_3=0,1		
8.2					t_4=0,1 x_4=0,1,2	*(proc. 1 is tp)*
				v_3=0,2 r_3=D,B	v_4=0,2 r_4=E,B	
8.3					x_4=2,3	
			v_2=0,2 r_2=C,B	v_3=2,3 r_3=B,C z_3=4,5 c_3=D,E	v_4=2,3 r_4=B,C	

9	*output*	A	C	E	F	*Minor change*
9.1				cn_3=0,1	cn_4=1,2 x_4=3,4	
		v_1=0,2 r_1=B	v_2=2,3 r_2=B,C	v_3=3,4 r_3=C,D	v_4=3,4 r_4=C,D	
9.2					x_4=4,5 v_4=4,5 r_4=D,E	
9.3			z_2=3,4 c_2=C,D			

10	*output*	A	D	E	F	*Major change*
10.1			cn_2=0,1	cn_3=1,2	cn_4=2,3	
10.2						
10.3		z_1=1,2 c_1=A,B t_1=1,0 x_1=1,0 v_1=2,0	z_2=4,3 c_2=D,C t_2=1,0 x_2=1,0 v_2=3,0 cn_2=1,0	z_3=5,4 c_3=E,D t_3=1,0 x_3=1,0 v_3=4,0 cn_3=2,0	z_4=6,5 c_4=F,E t_4=1,0 x_4=5,0 v_4=5,0 cn_4=3,0	

11	*output*	B	C	D	E	*Minor change*
11.1				t_1=0,-1	t_2=0,-1,0	
11.2			t_2=0,-1	t_3=0,-1,0		
11.3		t_1=0,-1,12	t_2=0,-1,0,12		z_4=5,6 c_4=E,F	
12	*output*	B	C	D	F	*Minor change*
12.1				t_3=0,12	cn_4=0,1	
12.2				v_3=0,1 r_3=D,A	t_4=0,12 x_4=0,1 v_4=0,1 r_4=E,A	*(no tp is found)*
12.3			v_2=0,1 r_2=C,A	v_3=1,2 r_3=A,B z_3=4,5 c_3=D,E	x_4=1,2 v_4=1,2 r_4=A,B	
13	*output*	B	C	E	F	*Minor change*
13.1		v_1=0,1 r_1=B,A	v_2=1,2 r_2=A,B	cn_3=0,1 v_3=2,3 r_3=B,C	cn_4=1,2 x_4=2,3 v_4=2,3 r_4=B,C	
13.2		v_1=1,2 r_1=A,B	v_2=2,3 r_2=B,C	v_3=3,4 r_3=C,D	x_4=3,4 v_4=3,4 r_4=C,D	
13.3		v_1=2,3 r_1=B,C	v_2=3,4 r_2=C,D z_2=3,4 c_2=C,D	v_3=4,5 r_3=D,E	x_4=4,5 v_4=4,5 r_4=D,E	
14	*output*	B	D	E	F	*Minor change*
14.1			cn_2=0,1	cn_3=1,2	cn_4=2,3 x_5=5,6 v_4=5,6 r_4=E,F	
14.2						

14.3		$z_1=2,3$ $c_1=B,C$				
15	*output*	C	D	E	F	
15.1		$cn_1=0.1$	$cn_2=1.2$	$cn_3=2.3$	$cn_4=3.4$	*termination criteria*
15.2						
15.3						*STOP*

When the cell is blank, means that non of the variables in the corresponding processor have changed their values.

4.5 Parallel generation of combinations with repetitions

A *(m,n)*-combination from the set of arbitrary elements $P = \{p_1, p_2, \ldots, p_n\}$ can be represented as $c_1c_2\ldots c_m$ where $p_1 \le c_1 < c_2 < \ldots < c_m \le p_n$. Let $z_1z_2\ldots z_m$ be the corresponding array of indices, i.e. $c_i = p(z_i)$, $1 \le i \le m$, where the notation $y(r)=y_r$ is used to avoid double indices. As described in earlier chapters, there is a one-to-one correspondence between the *(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 $h_i = z_i - i + 1$ and $x_i = p(h_i)$. Then, $p_1 \le x_1 \le x_2 \le \ldots \le x_m \le p_{n-m+1}$, and $x_1x_2\ldots x_m$ is a combination with repetitions of m out of $n-m+1$ elements; we call it a *(m,n-m+1)-r_*combination.

Because of the simple relationship, any algorithm that generates the *(m,n+m-1)*-combinations from a set of arbitrary elements may be used to generate the *(m,n)-r_*combinations, and vice versa. The only difference in the case of a parallel algorithm is in the way the processors pick their elements for each combination. If z_i is the index of c_i in P

of a *(m,n+m-1)*-combination, replacing z_i by z_i-i+1 will yield to the *(m,n)-r_*combination, while the generating algorithm remains the same. In the case of the algorithm presented in section 4.3, the processor picks its element using the instructions at lines 2, 3, 22, 28 and 31. To generate the *(m,n)-r_*combinations, the instructions: *read* p_i p_{n-m+i} $p_{n-m+i-1}$ at line 2, $c_i \leftarrow p_i$ at line 3, $r_i \leftarrow p(v_i)$ at line 22, $c_i \leftarrow p_{n-m+i}$ at line 28, and $c_i \leftarrow p(z_i)$ at line 31 must be replaced by the instructions: *read* p_1 p_{n-m+1} p_{n-m}, $c_i \leftarrow p_1$, $r_i \leftarrow p(v_i-i+1)$, $c_i \leftarrow p(z_i-i+1)$, and $c_i \leftarrow p(z_i-i+1)$ respectively. Incorporating these modifications in algorithm PCLES, algorithm PCLES2 to generate the *(m,n)-r_*combinations can be obtained and coded as shown in the next page.

4.6 Making the algorithm adaptive

The algorithm presented in this chapter can be made adaptive (i.e. to run on a linear array consisting of an arbitrary number of k processors) according to the definition presented in Chapter III. The approach applied here is that m processors produce the combination $c_1c_2...c_m$ of m elements out of n elements such that processor i is responsible for producing element c_i. For example, combination *1 2 4 5* is produced in the following way: processor *1* produces *1*; processor *2* produces *2*; processor *3* produces *4*; processor *3* produces *5*. This approach is used to generate combinations of m elements out of n elements in [19, 26, 29, 31, 33]. There are two cases to consider [17]:

1) $k < m$. In this case each processor will do the job of m/k processors, with m/k rounded appropriately if not an integer, so that the last processor does slightly less work. This will obviously require $O(m/k)$ memory per processor.

```
ALGORITHM. PCLES2(n, m). (with repetitions)
BEGIN
1.  For i ← 1 To m Do in parallel {
2.      read p_1, p_{n-m+1}, p_{n-m};
3.      z_i ← i; c_i ← p_1; t_i ← 0; x_i ← 0; v_i ← 0; cn_i ← 0;
4.      Repeat {
5.          output c_i;
6.          If z_i = n-m+i Then cn_i ← cn_i+1;   /* cn counts repetitions of the maximal value in i */
7.          If i = m and z_i = n-1 Then  t_i ← -1;   /* initiate search for turning point t  */
8.          For j ← 1 to 3 Do {
9.              If i < m and t_i = 0 and t_{i+1} = -1 Then {
10.                 t_i ← -1;      /* continue search for t */
11.                 t_{i+1} ← 0;
12.                 If z_i < n-m+i-1 Then {
13.                     x_i ← z_i;     /* turning point t found */
14.                     t_i ← i; }
15.                 Else If i = 1 Then t_i ← 2n; }   /* initiate termination message */
16.             If i > 1 and t_i = 0 and t_{i-1} > 0 Then {
17.                 t_i ← t_{i-1};
18.                 x_i ← x_{i-1}; }
                /* distribute t and z_i to proc. t, t=1,..., m; they are saved as t_i and x_i, resp. */
19.             If i = m and t_i > 0 and v_i = v_{i-1} and x_i < n Then {
20.                 x_i ← x_i+1;
21.                 v_i ← x_i;
22.                 r_i ← p(v_i-i+1); }
                /* generating new values, for next major change, for processors t, t+1, ..., m */
23.             If i < m and t_i > 0 and v_{i+1} > 0 and v_i ≠ x_i+i-t_i+1 Then {
24.                 r_i ← r_{i+1};
25.                 v_i ← v_{i+1}; }}
            /* pipeline to the left new values v_i and r_i for z_i and c_i, respectively */
26.         If i < m and z_i = n-m+i-1 and z_{i+1} = n-m+i+1 Then {
27.             z_i ← n-m+i;
28.             c_i ← p(z_i-i+1) }
29.         If i = m and z_i < n Then {      /* minor change */
30.             z_i ← z_i+1;
31.             c_i ← p(z_i-i+1); }  /* minor change in processor m */
32.         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 {
33.             z_i ← v_i;
34.             c_i ← r_i;
35.             t_i ← 0;
36.             x_i ← 0;
37.             v_i ← 0;
38.             cn_i ← 0; }          /* major change  */
        Until  t_i = 2n and cn_i = i }}
END.
```

2) $k \geq m$ and $r = k/m$ (integer division). In this case the processors are divided into r groups of m processors each, such that each group produces an interval of consecutive combinations. If k/n is not an integer, then $k-rm$ processors may either be left without any job which will not change the asymptotic behavior of the adaptive algorithm, or assume the appropriate portion of the combinations to generate in the way given in 1).

Hence, in both cases the set of all combinations is divided into r (approximately) equal groups. Each of the r groups will produce (approximately) $\lfloor C(m,n)/r \rfloor$ combinations. The first and last combination in each group can be determined in a preprocessing step by applying the known unranking function presented in Chapter II to map integers between 1 and $C(m,n)$ onto the set of (m,n)-combinations. However, this function involves very large integers, such as $C(m,n)$, and for practical purposes, is inefficient.

A new numbering system that does not deal with large integers and yet divides the job evenly into groups is described in Chapter III. The new system finds for a given g such that $1 \leq g \leq n$, the combination $X = x_1x_2...x_m$ such that the ratio of the number of combinations preceding X and the total number of combinations is less then or equal to g. In other words, it finds the combination with the ordinal number $\lfloor gC(m,n) \rfloor + 1$. The group j, $1 \leq j \leq r$, will produce combinations numbered from $\lfloor (j-1)C(m,n)/r \rfloor + 1$ to $\lfloor jC(m,n)/r \rfloor$. Thus we apply the mentioned algorithm for $g = j/r$, $0 \leq j \leq r$, to find the beginning and end of each group.

The above procedure is sequential and is supposed to be done by one processor in each of the groups. For $k < m$, the processors can simulate the work of k/m processors. This can be done sequentially by each processor. For $k > m$, the processors are divided into $r = k/m$ groups. Each group will have one last processor. One of these last processors (the last one) will send the data in a pipelined fashion to other processors. Each last processor in each group will pick its own elements that are needed for that group. Then, the last processor will generate the first combination using the unranking algorithm and calculate the values for the variables in the algorithm. Furthermore, it will pass the elements and values to all other processors in the group and the process will continue as usual. This will cause major delays. As an alternative approach, the last processor in each group will generate the first few combinations and send the elements to other processors until a major change is due in the system where all variables are set to zero. Then the process will continue as usual. Hence, using the method and results presented in [17], algorithms PCLES and PCLES2 can be made adaptive without using calculations with large numbers such as $C(m,n)$, while at the same time keeping all the desirable properties presented in Chapter III. However, although each processor produces its combinations in lexicographic order, the overall *chronological* order of outputting combinations is not lexicographic.

4.7 Application of algorithm

Using the algorithms presented in this chapter, it is possible to generate all subsets of a given set $\{p_1, p_2, \ldots, p_n\}$ such that the most desirable properties presented in Chapter III are satisfied in the following way: For $m = 1, 2, \ldots, n$ list all combinations of m items taken

from the set $\{p_1, p_2, \ldots, p_n\}$ using the algorithm presented in this chapter. All subsets will clearly be listed but not in lexicographic order. It can be mentioned that in [45] an algorithm is given for generating subsets that enjoys all desirable properties of Chapter III, but the algorithm does not deal with a set of arbitrary elements.

4.8 Architecture of a processor

The architecture of each processor in the linear array is very simple. Each processor consists of a central processing unit (CPU), an arithmetic/logical unit (ALU), a number of words of memory, several registers (LR,RR), some of which are responsible for all communications with left and right neighbors, and an input and output port (I/O) for producing the current element.

4.9 Conclusion

In this chapter, we presented a simple parallel algorithm to generate all combinations of m elements from an arbitrary set of n elements, on a linear array of m processors. Each processor has constant size memory, except processor m which has $O(n)$ memory, and each is responsible for producing one element of a given combination. There is a constant delay per combination, leading to an $O(C(m,n))$ time solution, where $C(m,n)$ is the total number of combinations. The algorithm is cost optimal, assuming the time to output the combination is counted, and does not deal with large integers, such as $C(m,n)$. The algorithm can be made adaptive while keeping all desirable optimality properties satisfied at the same time. Using the simple one-to-one correspondence between (m,n)-combinations and combinations

with repetitions of m out of n-m+1 elements, the algorithm was modified for generating combinations with repetitions drawn from an arbitrary set of elements.

In conclusion, the algorithm presented in this chapter is an improvement over all previous parallel combination generation algorithms in that it satisfies all desirable properties, listed in chapter III, and generates combinations from a set of arbitrary elements $\{p_1, p_2, \ldots, p_n\}$, on which an order relation $<$ is defined such that $p_1 < p_2 < \ldots < p_n$. This property is important, as it allows us to distinguish between algorithms that generate combinations of any ordered set, and algorithms that can only generate combinations of the set $\{1, 2, \ldots, n\}$.

Chapter V

Conclusions

5.1 Discussion of results

In this thesis we presented four different results in the area of combinations generation algorithms. First, we described the most important sequential algorithms for generating the combinations of m elements out of a set of n elements, i.e. the (m,n)-combinations. Then, the algorithms that generate the (m,n)-combinations in lexicographic order were implemented in *C*, measured and compared for different values of m and n. It was concluded that Semba's algorithm is the fastest sequential algorithms for large m. The time complexity of the best sequential algorithm is $O(m(m,n))$, assuming the time to output combinations is taken into account; otherwise, the time complexity of the algorithm, without producing the combinations as output, is $O(C(m,n))$.

Secondly, a comprehensive survey of all known parallel combinations generation algorithms is provided. The algorithms were evaluated according to the desirable properties for any parallel combinatorial generation algorithm in a tabular form.

The main contribution of this thesis is an original parallel algorithm for generating all combinations ***without repetitions*** of m elements from an arbitrary set of n elements. The

algorithm is lexicographic, uses a linear systolic array of m processors each having constant size memory (except processor m, which has $O(n)$ memory), has only constant delay between successive combinations, is cost-optimal (assuming the time to output the combination is counted), and does not deal with large integers (such as $C(m,n)$). Thus the algorithm satisfies all desirable properties, which was the case for the best parallel combinations algorithm [33]. However, the algorithm is an improvement over all previous work because it generates combinations out of the set of arbitrary elements $\{p_1, p_2, \ldots, p_n\}$, on which an order relation $<$ is defined such that $p_1 < p_2 < \ldots < p_n$. This property is important since it allows us to distinguish between algorithms that generate combinations out of any ordered set, and algorithms that can only generate combinations out of the set $\{1, 2, \ldots, n\}$.

Lastly, this algorithm was modified to generate all combinations ***with repetitions*** of m elements chosen from an arbitrary set of n elements. The algorithm satisfies the same desirable properties as in the case of the algorithm for generating combinations ***without repetitions***.

5.2 Open problems

The algorithm presented in Chapter IV assumes that the n elements are all stored in one processor. It remains 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, the exception made on the size of one processor can be lifted.

It remains an open problem to design a *cost-optimal* parallel combinations generation algorithm that creates all (m,n)-combinations without actually producing them as output. An optimal sequential algorithm in this sense, generates all (m,n)-combinations in $O(C(m,n))$, i.e. time linear in the number of combinations.

There exist algorithms for generating permutations [39] and derangements [46] satisfying the same desirable properties as the algorithm presented in Chapter IV. It remains an open problem to design algorithms satisfying the same desirable properties for generating subsets [36], set partitions [36], and other kinds of combinatorial objects.

Bibliography

[1] H. T. Kung, Why systolic architectures, Transactions on Computers Vol. 15, 37-46 (1982).

[2] H. T. K Kung, The structure of parallel algorithms, In Advances in Computers, pp. 65-112. Academic Press, New York (1980).

[3] H. S. Stone, Parallel computers, In Introduction to Computer Architectures, pp. 363-425. Science Research Associates, Chicago, (1980).

[4] V. Zakharov, Parallelism and array processing, IEEE Transactions on Computers Vol. 33, 45-78 (1984).

[5] J. Kurtzberg, Combinations (Algorithm 94), Commun. ACM Vol. 5, No. 6, pp. 344, (1962).

[6] C. J. Mifsud, Combination in lexicographical order (Algorithm 154), Commun. ACM Vol. 6, No. 3, pp. 103, 1963.

[7] P. J. Chase, Algorithm 382, Combinations of m out of n objects, Commun. ACM, Vol. 13, No. 6, pp. 368.

[8] G. Ehrlich, Four combinatorial algorithms (Algorithm 466), Commun. ACM Vol. 16, No. 11, pp. 690-691, (1973).

[9] C. N. Liu and D. T. Tang, Enumerating combinations of m out of n objects (Algorithm 452), Commun. ACM, Vol. 16, No. 8, 485, (1973).

[10] A. Nijenhuis, H. S. and Wilf, Combinatorial algorithms, Academic Press, New York (1975).

[11] J. R. Bitner, G. Ehrlich, and E. M. Reingold, Efficient generation of the binary reflected Gray code and its applications, Commu. ACM, Vol. 19, No. 9, pp. 517-521 (1976).

[12] E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial algorithms, Prentice-Hall, N. J. Englewood Cliffs, pp. 179-181, (1977).

[13] W. H. Payne, and F. M. Ives, Combination generators, ACM Transactions on Mathematical Software, Vol. 5, No. 2, pp. 163-172, (1979).

[14] S. G. Akl, A comparison of combination generation methods, ACM Transactions on Mathematical Software, Vol. 7, No. 1, pp. 42-45, (1981).

[15] I. Semba, A note on enumerating combinations in lexicographic order, Journal of Information Processing, Vol. 4, No. 1, pp. 35-37, (1981).

[16] F. Ruskey, Adjacent interchange generation of combinations, Journal of Algorithms, Vol. 9, pp. 162 180, (1988).

[17] I. Stojmenovic, On random and adaptive parallel generation of combinatorial objects, Int. J. Computer Mathematics, Vol. 42, pp. 125-135, (1991).

[18] J. M. Calvert, The parallel generation of permutation and derangements, Queens University Master Thesis, (1991).

[19] S. G. Akl, The design and analysis of parallel algorithm, Prentice Hall, Eaglewood Cliffs, (1989).

[20] S. G. Akl and I. Stojmenovic, Generating combinatorial objects on a linear array of processors, Computer Science Department, University of Ottawa, Ottawa, TR-93-10, (1993).

[21] P. Gupta and G. P. Bhattacharjee, Parallel Generation of Lexicographic Combination, Conference on Foundation of Software Technology and Theoretical Computer Science, Bangalore India, (1981).

[22] C. Y. Tang, M. W. Du and R. T. C. Lee, Parallel generation of combinations, Proc. int. Computer Sump., Taipei, Taiwan, pp. 1006-1010, (1984).

[23] L. E. Moses and R. V. Oakland, Tables of random permutations, Stanford University press, Stanford, Ca. (1963).

[24] G. De Balbine, Note on random permutations, Math. Comp., Vol. 21, pp. 710-712, (1967).

[25] M. C. Pike, Remark on algorithm 235, Comm. ACM, Vol. 8, pp. 445-445, (1965).

[26] B. Chan, and S. G. Akl, Generation combinations in parallel, BIT Vol. 26, pp. 2-6, (1986).

[27] G. H. Chen, and M. S. Chern, Parallel generation of permutations and combinations, BIT, Vol. 26, pp. 277-283, (1986).

[28] S. G. Akl, Adaptive and optimal parallel algorithms for enumerating permutations and combinations, The Computer journal, Vol. 30, No. 5, pp. 433-436, (1987).

[29] C. J. Lin, A parallel algorithm for generating combinations, Computers and Mathematics with Applications, Vol. 17, No. 12, pp. 1523-1533, (1989).

[30] C. J. Lin and J. C. Tsay, A systolic generation of combinations, BIT, Vol. 29, pp. 23-36, (1989).

[31] S. G. Akl, D. Gries, and I. Stojmenovic, An optimal parallel algorithm for generating combinations, Information Processing Letters, Vol. 33, No. 3, pp. 135-139, (1989).

[32] J. C. Tsay and C. J. Lin, A systolic design for generating combinations in lexicographic order, Parallel Computing Vol. 13, pp. 119-125, (1990).

[33] I. Stojmenovic, A simple systolic algorithm for generating combination lexicographic order, Computers and Mathematics with Applications, Vol. 24, No. 4, pp. 61-64 (1992).

[34] D. Sarkar. Cost and time-cost effectiveness of multiprocessing. IEEE Trans. on Parallel and Distributed Systems. (1992).

[35] G. D. Knott. A numbering system for combinations. Comm. ACM. Vol. 17. No. 1. pp. 45-46. (1974).

[36] I. Stojmenovic. An optimal algorithm for generating equivalence relations on a linear array of processors. BIT Vol. 30. pp. 424-436. (1990).

[37] M. C. Er, A parallel algorithm for cost optimal generation of permutations of *r* out of *n* items. Journal of Information & Optimization Sciences (India). Vol. 9. No. 1. pp. 53-56, (1988).

[38] M. Cosnard and A. G. Ferreira, Generating permutations on a VLSI suitable linear network, The computer Journal, Vol. 32, No. 6, pp. 571-573, (1989).

[39] S. G. Akl, H. Meijer, and I. Stojmenovic, An optimal systolic algorithm for generating permutations in lexicographic order, Journal of Parallel and Distributed Computing, Vol. 20, pp. 84-91, (1994).

[40] P. Gupta and G.P. Bhattacharjee, Parallel generation of permutations with repetitions lexicographically, Second Conference on Foundation of Software Technology and Theoretical Computer Science, Bangalore India, (1982).

[41] S. G. Akl, T. Duboux and I. Stojmenovic, Constant delay parallel counters, Parallel Processing Letters, Vol. 1, No. 2, pp. 143-148, (1991).

[42] H. T. Kung, The structure of parallel algorithms, Advances in Computers, M. C. Yovits, Ed., Academic Press, New York, pp. 65-112, (1980).

[43] H. S. Stone, Parallel computers: Introduction to Computer Architectures, H. S. Stone, Ed., Science Research Associates, Chicago, pp. 363-458, (1980).

[44] G. D. Knott, A numbering system for combinations, Comm. ACM, Vol. 17, No. 1, pp. 45-46, (1974).

[45] I. Stojmenovic, An optimal algorithm for generating equivalence relations on a linear array of processors, BIT, Vol. 30, pp. 424-436, (1990).

[46] S.G. Akl, J.M. Calvert and I. Stojmenovic, Systolic generation of derangements, Proceeding of the workshop Algorithms and Parallel VLSI Architectures II, (P. Quiton, Y. Roberts, ed.), Chateau de Bonas, France, June 1991, Elsevier Sci. Publ., pp. 59-70, (1992).

[47] S. G. Akl, T. Duboux and I. Stojmenovic, Constant delay parallel counters, Parallel Processing Letters, 1, 2, pp. 143-148, (1991).

[48] D. H. Lehmer, The machine tools of combinatorics, Applied combinatorial mathematics (E. F. Beckenbach ed.), Wiley, New York, pp. 5-31, (1964).

[49] H. Elhage and I. Stojmenovic, Systolic generation of combinations from arbitrary elements, Parallel Processing Letters, Vol. 2, No. 2, pp. 241-248, (1992).

Appendix

Programs for implemented sequential algorithms

```
/*************************************************************************
*
* Objective: Generate combinations in lexicographic order.
* Algorithm: Charles J. Mifsud.
*
* Author   : Hassan Elhage.
* Date     : October 28, 1993.
*
*************************************************************************/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <limits.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

#define CONST1 12
#define CONST2 25

int combi(int n, int r, int I[]);

int first = TRUE;
FILE *outptr;

main()
{
  int n,j,r,I[100], k;
  struct time t;
  long i;
  printf("Please enter n: \n");
  scanf("%d", &n);
  printf("Please enter starting r: \n");
  scanf("%d", &r);
  outptr = fopen("mifsud3.out","w");
  for(k = r; k <= n; k++) {
      gettime(&t);
      printf("The current time is: %2d:%02d:%02d:%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
      fprintf(outptr,"The current time is: %2d:%02d:%02d:%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
      for(i = 1; i <= k; i++)
          I[i] = i ;
      first = FALSE;
      i = 1;
      while (first == FALSE)
          combi(n,k,I);
      }
  gettime(&t);
```

```
  printf("The current time is: %2d:%02d:%02d:%02d\n",
      t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
  fprintf(outptr,"The current time is: %2d:%02d:%02d:%02d\n",
      t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
  fclose(outptr);
}

/* This function calculates the next combination for given n and r */

int combi(int n, int r, int I[])
{
  int s,j;

  if (first == TRUE)
      for (j = 1; j <= r; j++) {
          I[j] = j;
      first = FALSE;
      return 1;
  }
  if (I[r] < n) {
      I[r] = I[r] + 1;
      return 1;
  }
  for (j = r; j >= 2; j--)
      if (I[j-1] < n-r+j-1) {
          I[j-1] = I[j-1] + 1;
          for (s = j; s <= r; s++)
              I[s] = I[j-1]+s-(j-1);
      return 1;
  }
  first = TRUE;
  return 1;
}
```

```
/*************************************************************************
*
* Objective: Generate combinations in lexicographic order.
* Algorithm: Iciro Semba
*
* Author  : Hassan Elhage.
* Date    : October 28, 1993.
*
*************************************************************************/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <limits.h>
#include <math.h>

#define TRUE 1
#define FALSE 0

int combi(int x[], int n, int k);

int nk,h;
FILE *outptr;

main()
{
  int n,j,k,x[100], kk;
  long i;
  struct time t;

  printf("Please enter n: \n");
  scanf("%d", &n);
  printf("Please enter starting k: \n");
  scanf("%d", &k);
  outptr = fopen("simba3.out","w");

  for (kk=1; kk<= n; kk++) {
      printf("The combinations of %d out of %d are:\n", kk, n);
      fprintf(outptr,"The combinations of %d out of %d are:\n", kk, n);*/
      i = 1;
      h = n+1;
      gettime(&t);
      printf("The current time is: %2d:%02d:%02d:%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
      fprintf(outptr,"The current time is: %2d:%02d:%02d:%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
      while(h != 0) {
          combi(x,n,kk);
      }
  }
  gettime(&t);
```

```
  printf("The current time is: %2d:%02d:%02d:%02d\n",
     t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
  fprintf(outptr,"The current time is: %2d:%02d:%02d:%02d\n",
     t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
}

/* This function calculates the next combination for given n and k */

int combi(int x[], int n, int k)
 {
  int j;

  if (k-h < 0) {
     for (j = 1; j <= k; j++)
        x[j] = j;
     nk = n-k;
     h = k;
     if (n == k)
        h = 0;
     }
  else
     if (k-h == 0) {
        x[h] = x[h] + 1;
        if (x[h] == n)
           h = h-1;
        }
     else {
        x[h] = x[h]+1;
        if (x[h] < nk+h) {
           h = h+1;
           x[h] = x[h-1]+1;
           while (h < k) {
              h = h+1;
              x[h] = x[h-1]+1;
           }
        }
        else
           h = h-1;
        }
  return 1;
}
```