Parallel Algorithms for Generating Integer Partitions and Compositions[1]

Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario
Canada K7L 3N6

Ivan Stojmenović
Computer Science Department
University of Ottawa
Ottawa, Ontario
Canada K1N 9B4

Abstract. We present cost-optimal parallel algorithms for generating partitions and compositions of an integer n in lexicographic order. The algorithms use a linear array of n processors, each having constant size memory and each being responsible for producing one part of a given partition or composition.

1. Introduction

Given an integer n, it is possible to represent it as the sum of one or more positive integers a_i, i.e. $n = a_1 + a_2 + \cdots + a_m$. This representation is called a partition if the order of the a_i is of no consequence. Thus two partitions of an integer n are distinct if they differ with respect to the a_i they contain. For example, there are seven distinct partitions of the integer 5:

$$5, 4+1, 3+2, 3+1+1, 2+2+1, 2+1+1+1, 1+1+1+1+1.$$

For a given integer n, the representation $n = a_1 + a_2 + \cdots + a_m$ is said to be a composition if the order of the a_i is important. Thus two compositions of an integer n are distinct if they differ with respect to the a_i they contain and the order in which the a_i are listed. For example, there are sixteen compositions of the integer 5:

$$5, 4+1, 1+4, 3+2, 2+3, 3+1+1, 1+3+1, 1+1+3, 2+2+1,$$
$$2+1+2, 1+2+2, 2+1+1+1, 1+2+1+1, 1+1+2+1,$$
$$1+1+1+2, 1+1+1+1+1.$$

The partitions and compositions of an integer have been the subject of extensive study for over 300 years, since Leibniz asked Bernoulli if he had investigated $P(n)$, the number of partitions of an integer n. Details of the history and the state of the art as of 1920 can be found in Chapter 3 of [D]. Additional details and

[1]This research was supported by the Natural Sciences and Engineering Research Council of Canada

later results can be found in most combinatorics texts; in particular, see **[H, Li, Ri]**. This interest is partly motivated by the important role played by partitions and compositions in many problems of combinatorics and algebra. In general, a list of all combinatorial objects of a given type might be used to search for a counter-example to some conjecture, or to test and analyze an algorithm for its correctness or computational complexity. For computational purposes one is often interested in generating all the partitions of an integer, or sometimes just those satisfying various restrictive conditions. Several such algorithms, dealing with both the unrestricted **[An, Le, Mk, Mk1, NW, PW, RND, RJ, W]** and restricted **[An, Le, NMS, RJ, W, Wh]** cases, have appeared in the literature. Algorithms for integer compositions are given in **[F, Le, NW, PW, RND]**.

In this paper, we describe algorithms for generating partitions and compositions of integers using a linear array of processors. The algorithms that we present satisfy several optimality criteria. We begin by listing some desirable properties of parallel generation techniques.

Property 1. *The objects are listed in lexicographic order, i.e. if* $A = (a_1, a_2, \ldots, a_s)$ *and* $B = (b_1, b_2, \ldots, b_s)$ *are representations of objects, then A precedes B lexicographically if and only if, for some* $j \geq 1$, $a_i = b_i$ *when* $i < j$, *and* a_j *precedes* b_j. *Lexicographic order is desirable as it is the natural (dictionary) order, and can be easily characterized.*

Integer partitions and compositions are represented by integer sequences. Depending on the definition of 'precedes' among integers, various lexicographic orders can be obtained. In ordinary lexicographic order, the relation a 'precedes' b means $a < b$ (i.e. a is smaller than b). The algorithms in this paper generate objects in reverse lexicographic order, in which a 'precedes' b means $a > b$ (i.e. a is larger than b).

Property 2. *The algorithm is cost-optimal, i.e. the number of processors it uses multiplied by its running time matches—up to a constant factor—a lower bound on the number of operations required to solve the problem.*

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

a) The time required to "create" the objects, without actually "producing" as output the s elements of each object, is counted. Optimal sequential algorithms in this sense generate objects in $O(B(s))$ time, i.e. time linear in the number of objects of s elements.
b) The time to output each object in full is counted. Here, optimal sequential algorithms run in $O(s * B(s))$ time, since it takes $O(s)$ time to produce an object. In this paper we adopt this measure; designing optimal parallel algorithms under measure (a) remains an open problem.

Property 3. *The time required by the algorithm between any two consecutive objects it produces is constant. A constant time delay between outputs is particularly important in applications where the output of one computation serves as input to another. As usual in sequential computation, we assume that a processor requires constant time to perform an elementary operation. Examples of such elementary operations are adding or comparing two numbers of* $\log n$ *bits each.*

Property 4. *The model of parallel computation should be as simple as possible. Arguably, the simplest such model is a linear array of* p *processors, indexed* 1 *through* p*, where each processor* i $(2 \leq i \leq p-1)$ *is connected by bi-directional links to its immediate left and right neighbors,* $i-1$ *and* $i+1$*, and processors* 1 *and* p *are each connected to one neighbor. This model is practical, as it is amenable to VLSI implementation* [A1].

It should be noted here that, from both the theoretical and practical points of view, an algorithmic result is more valuable if it is derived on a parallel model of computation that makes the fewest assumptions possible about processor connectivity. Indeed, the weaker the model, the stronger an optimality result, since a more powerful model can simulate the algorithm with no increase in running time. In this respect, the linear array of processors (the model for which all of the algorithms in this paper are defined) is weaker than the two-dimensional array or hypercube models, which are in turn weaker than the Parallel Random Access Machine (PRAM) [A1]. The latter consists of a set of processors sharing a common memory from which they can read and to which they can write. Depending on whether more than one processor can gain access to a given memory location simultaneously, we distinguish between the EREW PRAM (no simultaneous reads or writes), CREW PRAM (simultaneous reads, but not writes), etc.

Property 5. *Each processor needs as little memory as possible, preferably a constant number of words, each of* $\log n$ *bits and hence capable of storing an integer no larger than* n*. This implies that no processor can store an array of size* n*, or a counter up to* $n!$.

Property 6. *The algorithm should produce all objects of a given kind, not just a subset.*

Algorithms exist that satisfy these criteria for generating permutations [AMS], combinations [AGS,LT], derangements [ACS], subsets and equivalence relations [St]. There a number of parallel techniques for generating combinatorial objects that do not satisfy all listed properties: for example, such algorithms exist for permutations [CC, L, AMS, A1, A2], combinations [CA, CC, A1, A2], and derangements [GB].

The algorithms presented in this paper satisfy all properties 1–6, and generate partitions or compositions of a given integer (therefore they provide the first

parallel analogues to known sequential techniques). More precisely, we describe algorithms for generating the following objects:

(1) partitions of n in the multiplicity representation,
(2) partitions of n in the standard representation,
(3) partitions of n whose largest part is k in the multiplicity representation,
(4) partitions of n whose largest part is k the standard representation,
(5) partitions of n into m parts in the multiplicity representation,
(6) compositions of n into m parts,
(7) compositions of n into any number of parts.

The number of processors used is $O(n^{1/2})$ for problems (1), (3) and (5), $O(m)$ for (6), and $O(n)$ for (2), (4), and (7).

2. Generating partitions in parallel

In standard representation, a partition of n is given by a sequence $x_1 \ldots x_m$, where $x_1 \geq x_2 \geq \cdots \geq x_m$, and $x_1 + x_2 + \cdots + x_m = n$. In the sequel x will denote an arbitrary partition and m will denote the number of parts of x (m is not fixed). The number of parts of x greater than unity will be denoted by h, i.e. $x_i > 1$ for $1 \leq i \leq h$, and $x_i = 1$ for $h < i \leq m$. Similarly let t denote the number of parts of x greater than 2. It is sometimes more convenient to use a multiplicity representation for partitions in terms of a list of the distinct parts of the partition and their respective multiplicities. Let $y_1 > \cdots > y_d$ be all distinct parts in a partitions, and $c_1, \ldots, c_d$ their respective (positive) multiplicities. Clearly $c_1 y_1 + \cdots + c_d y_d = n$.

The number of partitions $P(n)$ of n can be determined using the following recurrence relation: $P(n, l) = P(n-l, l) + P(n, l-1)$ $(n \geq l \geq 1)$, where $P(n, l)$ is the number of partitions of n such that the largest part x_1 is no larger than l. Using boundary conditions $P(n, 1) = P(1, l) = P(n, 0) = P(0, l) = 1$ and $P(n) = P(n, n)$, $P(n)$ can be determined in $O(n^2)$ using the technique of dynamic programming [RND].

The well known sequential algorithm for generating all partitions in (reverse) lexicographic order is due to G. Ehrlich (cf. [RND]) and is described in [NW, RND, PW]. It finds the next partition from a given one by decreasing element x_h by one and updating $x_{h+1} = \cdots = x_{h+r} = x_h - 1$ where $r = (n - x_1 - \cdots - x_h + 1)/(x_h - 1)$ (integer division), $x_{h+r+1} = n - x_1 - \cdots - x_{h+r}$, and $x_{h+r+2} = 0$. For example, the partition that follows $8+5+1+1+1+1+1+1$ is $8+4+4+3$. In other words, a partition is derived from the previous one by subtracting 1 from the rightmost part greater than 1, and distributing the remainder as quickly as possible. The delay between the generation of two consecutive partitions in the algorithm is $O(n)$ in the worst case.

2.1. Generating partitions in the multiplicity representation

If the above method is implemented in the multiplicity representation [NW, RND,

PW], a loop-free algorithm can be obtained, i.e. all partitions can be produced with constant delay per partition (if the time to output partitions is not taken into consideration). The following code is adapted from [NW].

Algorithm 1. Sequential generation of partitions of n in the multiplicity representation with nonincreasing parts.

$y_1 \leftarrow n$;
$c_1 \leftarrow 1$;
$d \leftarrow 1$;
print out c_1, y_1;
repeat
  if $y_d = 1$ **then** $\{\text{rem} \leftarrow y_{d-1} + c_d; d \leftarrow d - 1\}$ **else** $\text{rem} \leftarrow y_d$;
  $\text{lim} \leftarrow y_d - 1$;
  if $c_d > 1$ **then** $\{c_d \leftarrow c_d - 1; d \leftarrow d + 1\}$;
  $y_d \leftarrow \text{lim}$;
  $c_d \leftarrow \lfloor \text{rem}/\text{lim} \rfloor$;
  $\text{dif} \leftarrow \text{rem} - c_d * \text{lim}$;
  if $\text{dif} > 0$ **then** $\{d \leftarrow d + 1; y_d \leftarrow \text{dif}; c_d \leftarrow 1\}$;
  print out c_i, y_i for $i = 1, 2, \ldots, d$
until $y_1 = 1$

The calculation is applied only on indices d, $d-1$, $d+1$, which means that, if the time required to produce a partition as output is not counted, the algorithm has constant delay between partitions. This procedure is implementable on a linear array of n processors in a straightforward way, and the resulting algorithm satisfies all properties 1–6. The number of processors is, in the multiplicity representation, actually less than n. From $y_1 > \cdots > y_d$ it follows that $1 + 2 + \cdots + d \leq y_d + \cdots + y_1 \leq n$, i.e. $d(d-1)/2 \leq n$. This in turn implies $d \leq (2n)^{1/2}$, which is also the number of processors that are needed in a linear array.

In addition to registers y_i, c_i, rem_i, lim_i, and dif_i, each processor has a register m_i for message exchange ($m_d = 1$; $m_i = 2$ if processor i desires to communicate a message to the right neighbor (processor $i + 1$); $m_i = 3$ if processor i desires to communicate a message to processor $i - 1$; $m_i = 0$ otherwise), and a register term_j for termination. All processors will terminate simultaneously when the termination register receives the value 1. We obtain the parallel Algorithm 2.

Theorem 1. *Algorithm 2 generates, in the multiplicity representation, all partitions of n in reverse lexicographic order and with constant delay per partition on a linear array of $n^{1/2}$ processors, thus achieving an optimal cost of $O(n^{1/2} P(n))$; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as $P(n)$.*

Algorithm 2. Parallel generation of partitions of n in the multiplicity representation with nonincreasing parts.

```
for each processor j (1 ≤ j ≤ (2n)^{1/2}) do in parallel
    y_j ← 0; c_j ← 0; m_j ← 0; term_j ← 0;
    if j = 1 then {y_j ← n; c_j ← 1; m_j ← 1; print out c_j, y_j};
    repeat
      if m_j = 1 then if y_j = 1 then
                {read y_{j-1}; rem_j ← y_{j-1} + c_j; m_j ← 3; c_j ← 0}
                                  else rem_j ← y_j;
      read m_{j+1};
      if m_{j+1} = 3 then {read rem_{j+1}; rem_j ← rem_{j+1}; m_j ← 1};
      if m_j = 3 then m_j ← 0;
      if m_j = 1 then {lim_j ← y_j − 1; if c_j > 1 then {c_j ← c_j − 1; m_j ← 2}}
      read m_{j-1}; if m_{j-1} = 2 then {read lim_{j-1}, rem_{j-1};
                                         lim_j ← lim_{j-1}; rem_j ← rem_{j-1}; m_j ← 1};
      if m_j = 2 then m_j ← 0;
      if m_j = 1 then {y_j ← lim_j; c_j ← ⌊rem_j / lim_j⌋;
                       dif_j ← rem_j − c_j * lim_j; if dif_j > 0 then m_j ← 2};
      read m_{j-1}; if m_{j-1} = 2 then {m_j ← 1; y_j ← dif_j; c_j ← 1};
      if m_j = 2 then m_j ← 0;
      if term_j = 0 then
                    {read term_{j-1}; if term_{j-1} > 0 then term_j ← term_{j-1} − 1}
                  else term_j ← term_j − 1;
      if j = 1 and y_j = 2 and term_j = 0 then term_j ← ⌊n/2⌋ + 1;
      if c_i > 0 then print out c_i, y_i;
    until term_j = 1
```

2.2. Generating partitions in the standard representation

If the output is desired in the standard representation, then the sequential algorithm has a linear time delay in the worst case, even when the time required for output is not taken into consideration. The same sequential algorithm can be used, with minor modifications, to produce partitions in the standard representation. A direct implementation of the sequential algorithm on parallel models will give the following results: constant delay on the CREW PRAM, $O(\log n)$ time on the EREW PRAM, and linear time (per partition) on a linear array of processors. Thus, designing a cost-optimal partition procedure on a linear array of processors, with standard output, is a nontrivial problem, the solution to which is presented below. For illustration, Table 1 gives all partitions of 7 in reverse lexicographic order.

Processor i is responsible for producing part x_i. For a partition into m parts,

7						
6	1					
5	2					
5	1	1				
4	3					
4	2	1				
4	1	1	1			
3	3	1				
3	2	2				
3	2	1	1			
3	1	1	1	1		
2	2	2	1			
2	2	1	1	1		
2	1	1	1	1	1	
1	1	1	1	1	1	1

Table 1

let $x_{m+1} = \cdots = x_n = 0$ for convenience. Each processor i also keeps a register $r_i = n - x_1 - \cdots - x_i$, namely the remainder to be distributed once x_i is established.

Each processor produces the same part unless it is informed to change it. Consider first some trivial cases. If $x_k = 2$ and $x_{k+1} = 1$, in the following step we set $x_k = 1$; this means that one more part of size 1 should be added. This is done automatically unless a message to stop adding new unit parts is received. Thus if $x_k = 1$ and $x_{k+1} = 0$, then in the next step we have $x_{k+1} = 0$. Also, when $x_k = 1$ then in the next step we repeat $x_k = 1$ until a message has arrived telling when to stop producing 1.

Whenever $x_k > 2$ and $x_{k+1} \leq 1$, processor k subtracts 1 from x_k. However, some processors with index greater than k have to adjust their value at the same moment, which will not be possible unless they are prepared in advance to do so. It is a trivial case if $r_k \leq 1$: in the next step, x_{k+1} increases by 1, and it is the only adjustment in the system. However, when $x_{k+1} = 1$ and $r_k > 1$, the system undergoes a major change (major changes are underlined in Table 1). Processor k begins preparing processors $k+1, k+2, \ldots$, for a major change whenever the two conditions $x_k > 2$ and $x_{k+1} = 2$ are true for the first time. It sends a message toward processors $k+1, k+2, \ldots$ informing them about their new value and the moment when the new value will take effect.

After subtracting 1 from x_k the amount $r_k + 1$ should be distributed over processors $k+1, k+2, \ldots, k+t$ such that each gets the maximal possible value no greater than $x_k - 1$. Clearly $t = \lceil (r_k + 1)/(x_k - 1) \rceil$. On the other hand, when 2 is encountered in x_{k+1} for the first time, the amount $r_k - 1$ is partitioned as $2+2+\cdots+2+1+\cdots+1$ in all possible ways. The number of such partitions

is $\lfloor r_k/2 \rfloor + 1$. If the message is communicated with unit speed (from a processor s to processor $s+1$ between the productions of two partitions), there is enough time to communicate the message to processors $k+1, \ldots, k+t$ before the change becomes effective, because $\lceil (r_k+1)/(x_k-1) \rceil \leq \lfloor r_k/2 \rfloor + 1$ is always satisfied. However, there are r_k 1s in the last partition $\cdots + x_k + 1 + \cdots + 1$ before the major change, and a message with unit speed is not able to reach the later 1s in the partition to inform them to stop producing 1s at a certain moment. Thus we decide to double the speed of the message: it advances by two processors between the productions of two partitions. The total path length of a message is $2\lfloor r_k/2 \rfloor$ and the distance from processor k to the last 1 is rem_k. Because this can still be short by one we actually let processor $k+1$ start the message by getting data from processor k. The contents of the message is in fact the new value in the processors. Together with the message the waiting time is also communicated, which determines when the new value takes effect. The message passing is indicated in bold in the examples of Tables 1 and 2.

8	**2**	**2**	**2**	**2**	**1**				
8	2	**2**	**2**	**1**	**1**	**1**			
8	2	2	**1**	**1**	**1**	**1**	**1**		
8	2	1	1	1	1	**1**	**1**	**1**	
8	1	1	1	1	1	1	1	1	1
7	7	3							

Table 2

The example in Table 1 also shows how the algorithm terminates. The termination message is originated by processor 1 when it produces 2 for the first time, and communicated with double speed as before.

In the Algorithm 3, the eight numbered statements inside the loop correspond to printing the partition, forwarding messages (w denotes the waiting time, m is the new value for x, rd is the new value for r, and t is the termination flag), initiating the messages, initiating the termination flag, updating the values x in obvious cases, assigning new values for x and r (major change), decreasing the waiting time, and decreasing the termination time, respectively.

Theorem 2. *Algorithm 3 generates, in the standard representation, all partitions of n in reverse lexicographic order and with constant delay per partition on a linear array of n processors, thus achieving an optimal cost of $O(nP(n))$; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as $P(n)$.*

Algorithm 3. Parallel generation of partitions of n in the standard representation.

for each processor k from 1 to n do
$x_k \leftarrow 0; r_k \leftarrow 0; m_k \leftarrow 0; w_k \leftarrow 0;$ **if** $k = 1$ **then** $x_k \leftarrow n; t_k \leftarrow 0;$
repeat
1. **if** $x_k > 0$ **then** print out x_k;
2. **for** $i \leftarrow 1$ **to** 2 **do** $\{t_k \leftarrow t_{k-1}$; **if** $w_{k-1} > 0$ and $w_k = 0$ **then** $\{w_k \leftarrow w_{k-1}$;
   if $rd_{k-1} > m_{k-1}$ **then** $\{m_k \leftarrow m_{k-1}; rd_k \leftarrow rd_{k-1} - m_k\}$
   else $\{m_k \leftarrow rd_{k-1}; rd_k \leftarrow 0\}\}\}$;
3. **if** $x_{k-1} > 2$ and $x_k = 2$ and $w_k = 0$ **then** $\{$
   $m_{k-1} \leftarrow x_{k-1} - 1; rd_{k-1} \leftarrow r_{k-1} + 1; w_k \leftarrow \lfloor r_{k-1}/2 \rfloor + 1; w_{k-1} \leftarrow w_k;$
   if $rd_{k-1} > m_{k-1}$ **then** $\{m_k \leftarrow m_{k-1}; rd_k \leftarrow rd_{k-1} - m_k\}$
   else $\{m_k \leftarrow rd_{k-1}; rd_k \leftarrow 0\}\}$;
4. **if** $k = 1$ and $x_k = 2$ and $t_k = 0$ **then** $t_k \leftarrow n/2 + 2$;
5. **if** $x_k = 0$ and ($x_{k-1} > 1$ or ($x_{k-1} = 1$ and $x_{k-2} < 3$ and $w_{k-1} <> 1$))
   then $x_k = 1$ **else** $\{$**if** $x_k > 1$ and $x_{k+1} < 2$ **then** $\{x_k \leftarrow x_{k-1}; r_k \leftarrow r_{k+1}\}$
   else $\{$**if** $x_k = 1$ and $x_{k-1} > 2$ and $x_{k+1} = 0$ **then** $x_k = 2\}\}$;
6. **if** $w_k = 1$ **then** $\{x_k \leftarrow m_k; m_k \leftarrow 0; w_k \leftarrow 0; r_k \leftarrow rd_k; rd_k \leftarrow 0\}$;
7. **if** $w_k > 1$ **then** $w_k \leftarrow w_{k-1}$;
8. **if** $t_k > 1$ **then** $t_k \leftarrow t_{k-1}$;
until $t_k = 1$

3. Partitions whose largest part is k

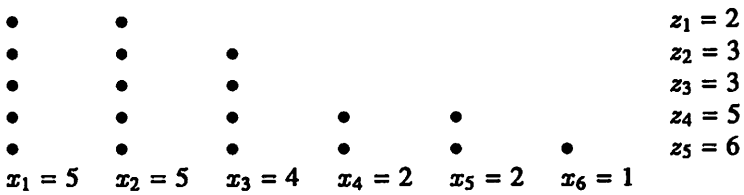
In this and following sections we consider some cases of restricted partitions. We begin by investigating partitions of n in which the largest part is k, i.e. $x_1 = k$ in the standard representation, and $y_1 = k$ in the multiplicity representation. The case of partitions whose largest part is k is given in [NW] as a special case of their general method for listing, ranking and unranking combinatorial objects. Note that the case of partitions of n whose largest part is smaller than or equal to k is, by adding one more part of size k, equivalent to the case of partitions of $n + k$ with largest part exactly k.

Sequential generation algorithms for partitions whose largest part is k can be obtained from the algorithm for the case of unrestricted partitions of n by "cutting" the execution, i.e. by choosing the first and last partitions. The first partition is $y_1 = k, c_1 = \lfloor n/y_1 \rfloor, y_2 = n - c_1 y_1, c_2 = 1$ if $y_2 > 0, c_2 = 0$ otherwise, and the last partition appears when y_1 becomes $k - 1$, in the multiplicity representation. Similarly one can obtain the first and last partitions for the standard representation. This can be easily incorporated in the corresponding parallel algorithms for generating partitions with largest part k in both the multiplicity and standard representations. The first partition is defined as above before entering the loop. The

termination criteria can be also modified easily by "pretending" that partitions of $n-k$ into any number of parts are being generated (the termination condition for such partitions and for the partitions under consideration coincide).

4. Partitions of n into m parts

Using the Ferrers graph (cf. [PW]) a one-to-one correspondence between partitions of n into m parts and partitions of n whose largest part is m is established. Let $z_1 \dots z_m$ be a partition into m nondecreasing parts, $z_1 \leq \cdots \leq z_m$, and $x_1 \dots x_k$, $x_1 \geq \cdots \geq x_k$, be a partition of n into any number of nonincreasing parts (i.e. k varies) with largest part $x_1 = m$. The following Ferrers graph illustrates the relationship between the two kinds of partitions.



The following relation follows from the Ferrers graph: $z_{m-i+1} = \max\{j \mid x_j \geq i\}$. Consider now the corresponding multiplicity representation of partitions with largest part m: $c_1, \dots, c_d$ are the multiplicities of $y_1, \dots, y_d$, where $m = y_1 > \cdots > y_d$, and $c_1 d_1 + \cdots + c_d y_d = n$. For the partitions into m parts, let $e_1, \dots, e_d$ be the multiplicities of $w_1, \dots, w_d$ (clearly the two sequences have the same number d of different parts), where $w_1 < \cdots < w_d$, $e_1 w_1 + \cdots + e_d w_d = n$, and $e_1 + \cdots + e_d = m$. Then it easily follows that $w_i = c_1 + c_2 + \cdots + c_i$, and $e_i = y_i - y_{i+1}$ where $y_{d+1} = 0$. The sums for w_i can be easily maintained during the execution of the program for generating partitions of n with largest part m in the multiplicity representation (in nonincreasing order of parts); recall that at any step in the algorithm all changes are done in processor d and its immediate neighborhood, and the partial sum $c_1 + \cdots + c_i$ is affected only in these processors. The necessary modifications to Algorithms 1 and 2 can be done in a straightforward way.

There exists another solution for the case of partitions of n into m parts in the multiplicity representation. In **[Le,RND]** algorithms are presented for the restricted case of partitions in which the number of parts is exactly m, and in lexicographic order for each fixed m, but considering the parts of the partition in increasing order. In fact, this algorithm was discovered by K.F. Hindenburg in 1778 (cf. **[RND]**). To obtain the next partition from the current one, the elements are scanned from right to left, stopping at the rightmost x_i such that $x_m - x_i \geq 2$. Replace x_j by x_{i+1} for $j = i, i+1, \dots, m-1$ and then replace x_m by the remainder, to get the sum n. For example, in the partition 11334, $i = 2$ and the next partition is 12225.

If the multiplicity representation is used, the algorithm is again loop free and works on the last indices only, enabling a parallel implementation satisfying properties 1–6; a linear array of m (more precisely, $(2m)^{1/2}$) processors suffices.

Finding a corresponding parallel algorithm for partitions of n into m parts in the standard representation is an open problem for further research.

5. Compositions of n into m parts

Compositions of n into exactly m parts can be written in the form $x_1 + \cdots + x_m = n$, where there is no requirement for the decreasing order of parts (i.e. $1+2+1$ and $2+1+1$ are two different compositions). Sequential algorithms for generating compositions appear in [NW, PW, RND].

There are two cases:

a) $x_i > 0$ for each i. Let $y_1, \ldots, y_m$ be the following sequence:

$$y_i = x_1 + \cdots + x_i.$$

The sequence $y_1, \ldots, y_m$ is a combinations of m out of n elements such that $y_m = n$; in other words, it is a combinations of $m-1$ out of $n-1$ elements from $\{1, \ldots, n-1\}$, and the number of compositions in question is $R(m,n) = C(m-1, n-1)$, where the latter is the binomial coefficient (proofs can be found in [NW,RND]). Thus, this case can be solved using the solution for combinations [AGS,ST2]. Each sequence $x_1 \ldots x_m$ is easily obtained from $y_1 \ldots y_m$ by the relation: $x_i = y_i - y_{i-1}$ (with $y_0 = 0$).

b) $x_i \geq 0$. By adding 1 to each part we get compositions of the number $n+k$ and proceed as in case (a).

Thus we obtain the following theorem.

Theorem 3. *It is possible to generate all compositions of n into m parts in lexicographic order and with constant delay per composition on a linear array of n processors, thus achieving an optimal cost of $O(nR(m,n))$; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as $R(m,n)$.*

6. Compositions of n into any number of parts

Let the sequence $y_1 \ldots y_m$ be defined as in 5(a). Here, however, m is not fixed. This case corresponds to the generation of all subsets of $\{1, 2, \ldots, n-1\}$. A solution to this problem on a linear array of processors appears in [S]. The number of such compositions is $R(n) = 2^n - 1$.

Theorem 4. *It is possible to generate all compositions of n into any number of parts in lexicographic order and with constant delay per composition on a linear array of n processors, thus achieving an optimal cost of $O(nR(n))$; furthermore each processor has a memory of constant size and can generate elements without the need to deal with large integers such as $R(n)$.*

7. Conclusion

We derived cost-optimal algorithms for generating integer partitions and compositions on a linear array of processors. The algorithms produce partitions and compositions with constant delay.

The partition and composition generation algorithms can be made adaptive (i.e. to run on a linear array consisting of an arbitrary number k of processors) if the processors are divided into k/n groups of n processors each such that each group produces an interval of consecutive partitions or compositions. The first and last partitions/compositions in each group can be determined in a preprocessing step by applying known unranking functions. Since compositions are generated by means of combinations, the unranking function described in [K] that maps integers between 1 and $C(m, n)$ onto the set of combinations of m out of n elements can be used. However, the function involves very large integers. Another scheme that does not deal with large integers and yet divides the job evenly among groups is described in [St1].

A general method for unranking combinatorial objects is given in [NW], with partitions and compositions given as special cases. The method deals with large integers. However, we are aware neither of an unranking procedure for partitions, nor of a way of dividing partitions into groups of roughly equal size, that do not use large integers.

Finding a parallel algorithm for the case of partitions of n into m parts that satisfies properties 1–6 and uses the standard representation of partitions, remains an open problem for further research. The case of compositions of n whose largest part is k is not studied here.

References

A1 S.G. Akl, "The Design and Analysis of Parallel Algorithms", Prentice Hall, Englewood Cliffs, New Jersey, 1989.

A2 S.G. Akl, *Adaptive and optimal parallel algorithms for enumerating permutations and combinations*, The Computer Journal **30, 5** (1987), 433–436.

AGS S.G. Akl, D. Gries, and I. Stojmenović, *An optimal parallel algorithm for generating combinations*, Information Processing Letters **33** (1989/90), 135–139.

ACS S.G. Akl, J. Calvert, and I. Stojmenović, *Systolic generation of derangements*, Proc. of the Workshop on Algorithms and Parallel VLSI Architectures II, France, Elsevier (1992), 59–70.

AMS S.G. Akl, H. Meijer, and I. Stojmenović, *Optimal parallel algorithms for generating permutations*, Technical Report No. 90-270, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada (1990).

An G.E. Andrews, "The theory of partitions", Addison-Wesley, Reading, Ma, 1976.

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

CC G.H. Chen and M.-S. Chern, *Parallel generation of permutations and combinations*, BIT **26** (1986), 277–283.

D L.E. Dickson, "History of the theory of numbers, Vol. II, Diophantine Analysis", Chelsea Publishing Co., New York, 1971.

F W. Feller, "An Introduction to Probability Theory and its Applications", Wiley, NY, 1951.

FL T.I. Fenner and G. Loizou, *A binary tree representation and related algorithms for generating integer partitions*, The Computer J. **23**, 4 (1980), 332–337.

GB P. Gupta and G.P. Bhattacharjee, *A parallel derangement generation algorithm*, BIT **29** (1989), 14–22.

H M. Hall, "Combinatorial Theory", Blaisdell, Waltham, Mass., 1967.

K G.D. Knott, *A numbering system for combinations*, Comm. ACM **17**, 1 (1974), 45–46.

Kn D.E. Knuth, "The Art of Computer Programming, Vol. 1: Fundamental Algorithms", Addison-Wesley, Reading, Ma, 1968.

Le D.H. Lehmer, *The machine tools of combinatorics*, in "Applied Combinatorial Mathematics", Beckenbach (ed.), Wiley, NY, 1964.

L C.J. Lin, *Parallel generation of permutations on systolic arrays*, Parallel Computing **15**, 1 (1990), 267–276.

LT C.J. Lin and J.C. Tsay, *A systolic generation of combinations*, BIT **29** (1989), 23–36.

Li C.L. Liu, "Introduction to Combinatorial mathematics", McGraw Hill, 1968.

Mk J.K.S. McKay, *Partition generator, Algorithm 263*, CACM **8** (1965), 493.

Mk1 J.K.S. McKay, *Partitions in natural order, Algorithm 371*, CACM **13** (1970), 52.

NMS T.V. Narayana, R.M. Mathsen, and J. Saranji, *An algorithm for generating partitions and its applications*, J. Comb. Theory **11** (1971), 54–61.

NW A. Nijenhius and H.S. Wilf, "Combinatorial Algorithms", Academic Press, NY, 1975.

NW1 A. Nijenhius and H.S. Wilf, *A method and two algorithms on the theory of partitions*, J. Comb. Theory A **18** (1975), 219–222.

PW E.S. Page and L.B. Wilson, "An Introduction to Computational Combinatorics", Cambridge Univ. Press, 1979.

R R.C. Read, *A survey of graph generation techniques*, Lect. Notes in Math. **884** (1980), 77–89.

Ri J. Riordan, "An introduction to Combinatorial Analysis", John Wiley, 1958.

RND E.M. Reingold, J. Nievergelt, and N. Deo, "Combinatorial Algorithms", Prentice Hall, Englewood Cliffs, New Jersey, 1977.

RJ W. Riha, and K.R. James, *Efficient algorithms for doubly and multiply restricted partitions, Algorithm 29*, Computing **16** (1976), 163–168.

Sa C.D. Savage, *Gray code sequences of partitions*, J. of Alg. **10** (1989), 577–595.

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

St1 I. Stojmenović, *On random and adaptive parallel generation of combinatorial objects*, Int. J. Computer Math.. (to appear).

ST2 I. Stojemonić, *A simple systolic algorithm for generating combinations in lexicographic order*, J. of Comb. Math. and Comb. Computing. to appear.

W M.B. Wells, "Elements of Combinatorial Computing", Pergamon Press, Oxford, 1971.

Wh J.S. White, *Restricted partition generator, Algorithm 374*, CACM **13** (1970), 120.