

## GENERATING $n$ -ARY REFLECTED GRAY CODES ON A LINEAR ARRAY OF PROCESSORS<sup>1</sup>

IVAN STOJMENOVIC

*Computer Science Department, University of Ottawa,  
Ottawa, Ontario, Canada K1N 9B4  
E-mail: ivan@csi.uottawa.ca*

Received February 1995

Revised May 1995

Accepted by P. McKenzie

### ABSTRACT

We present a cost-optimal parallel algorithm for generating  $n$ -ary reflected Gray codes, i.e. variations of  $m$  elements out of  $\{0, 1, \dots, n-1\}$  in a Gray code order. It uses a linear array of  $m$  processors, each having constant size memory and each being responsible for producing one part of a given variation. The algorithm is simple and uses a weaker model of computation than a recently published algorithm. In addition, it can be made adaptive (i.e. to run on a linear array with an arbitrary number of processors) and can be generalized to produce variations out of an arbitrary set of elements.

*Keywords:* Gray code, linear array of processors, generating combinatorial objects.

### 1. Introduction

A binary reflected Gray code of length  $m$  is an ordered sequence of  $2^m$  binary codewords such that any two adjacent codewords differ in a single position. It has been studied in the literature [1,2] and corresponds to a Hamiltonian path on a hypercube. The notion is generalized in [3]. An  $n$ -ary Gray code of length  $m$  is an ordered sequence of  $n^m$  codewords for which adjacent codewords differ in exactly one digit and each digit may have any value from 0, 1, ...,  $n-1$ . One special such order of codewords is called  $n$ -ary *reflected* Gray code and is defined in [3]. We will refer to these codewords as being variations. A *variation* of  $m$  out of an  $n$  element set  $S = \{0, 1, \dots, n-1\}$  is a sequence  $x_1, \dots, x_m$  such that  $x_i \in S$ , for all  $i$ ,  $1 \leq i \leq m$ . Note that repeated elements are allowed. For example, 23221 and 21033 are both variations of 5 out of a 4-element set. The number of variations of  $m$  elements out of  $n$  items is  $V(n, m) = n^m$ . Gray codes of variations have application in analogue-to-digital conversion of data. Encoding and decoding for  $n$ -ary Gray codes has been studied in the literature: [3] by circuit and [4] by formulas. In [5] the  $n$ -ary reflected Gray codes are generated by converting each codeword

---

<sup>1</sup> This research was supported by the Natural Sci. and Eng. Res. Council of Canada.

from the corresponding numeral in the  $n$ -ary number systems. Two recursive sequential algorithms for generating  $n$ -ary Gray codes were presented in [6].

There are few parallel algorithms [7,8,9] for generating variations in literature. We begin by listing some desirable properties of parallel generation techniques.

**Property 1.** The objects (e.g. variations) are listed in minimal change order, i.e. two consecutive objects differ as little as possible.

**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. The time to output each object in full is counted. Here, optimal sequential algorithms for generating variations run in  $O(m * V(n,m))$  time, since it takes  $O(m)$  time to produce an object.

**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 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  $m$  processors, indexed  $1$  through  $m$ , where each processor  $i$  ( $2 \leq i \leq m-1$ ) is connected by bidirectional links to its immediate left and right neighbors,  $i-1$  and  $i+1$ , and processors  $1$  and  $m$  are each connected to one neighbor. This model is practical, as it is amenable to VLSI implementation [10].

**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  $V(n,m)$ .

Algorithms exist that satisfy these criteria for generating subsets [8] and permutations [11]. There are a number of known algorithms that satisfy criteria 2-5 but generate combinatorial objects in lexicographic rather than a minimal change order [8,12-17], and a number of parallel techniques for generating combinatorial objects that do not satisfy some of the listed properties.

The algorithm presented in [7] generates variations in the lexicographic order, while satisfying all other properties 2-5. Another algorithm with the same properties is given in [8] but it works correctly only for  $2n \geq m$ . The algorithm described in [9] satisfies all properties except property 4. Namely, [9] uses a linear array equipped with a reconfigurable bus system. We will show in this paper that the reconfigurable bus system is not needed to solve the problem at hand, and that the generation of  $n$ -ary reflected Gray codes can be done with an algorithm which is no more sophisticated than the algorithm [9]. The algorithm in this paper modifies the algorithm [7] to generate variations in a minimal change rather than lexicographic order.

The algorithm presented in this paper satisfies all properties 1-5.

## 2. Generating Variations in Parallel

The  $n$ -ary reflected Gray code of variations is defined formally in [3,4,6,9]. We shall define the same sequence in a different manner (the definition is equivalent to a theorem in [6]). Let  $x=x_1x_2\dots x_m$  and  $y=y_1y_2\dots y_m$  be two variations. Then  $x < y$  iff there exist  $i$ ,  $0 \leq i \leq m$ , such that  $x_j = y_j$  for  $j < i$  and either  $x_1 + x_2 + \dots + x_{i-1}$  is even and  $x_i < y_i$  or  $x_1 + x_2 + \dots + x_{i-1}$  is odd and  $x_i > y_i$ .

We will now prove that the order is a minimal change order (i.e. satisfied property 1). Let  $x$  and  $y$  be two consecutive variations in given order,  $x < y$ , and let  $x_j = y_j$  for  $j < i$  and  $x_i \neq y_i$ . There are two cases. If  $x_i < y_i$  then  $X_i = x_1 + x_2 + \dots + x_{i-1}$  is even and  $y_i = x_i + 1$ . Thus  $X_{i+1}$  and  $Y_{i+1}$  have different parity, since  $Y_{i+1} = X_{i+1} + 1$ . It means that either  $x_{i+1} = y_{i+1} = 0$  or  $x_{i+1} = y_{i+1} = n-1$  (the  $(i+1)$ -th element in  $x$  is the maximum at that position while the  $(i+1)$ -th element in  $y$  is the minimum at given position, and they are the same because of different parity checks). Similarly we conclude  $Y_j = X_j + 1$  and  $x_j = y_j$  for all  $j > i+1$ . The case  $x_i > y_i$  can be analyzed in analogous way, leading to the same conclusion.

As an example, 3-ary reflected Gray code order of variations out of  $\{0,1,2\}$  is as follows (the variations are ordered columnwise):

000	122	200
001	121	201
002	120	202
012	110	212
011	111	211
010	112	210
020	102	220
021	101	221
022	100	222

Our parallel generation algorithm runs on a linear array of  $m$  processors, as shown in Fig. 1. Processors work synchronously, in SIMD (single instruction multiple data) fashion.

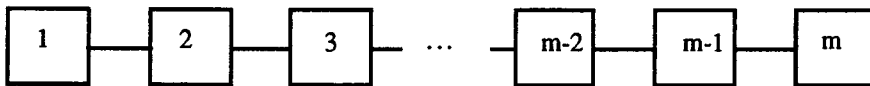


Figure 1. Linear array of processors

Each processor  $i$  is in charge of producing element  $x_i$  of the  $m$ -variation,  $1 \leq i \leq m$ ,  $0 \leq x_i \leq n-1$ . By a *block* of processor  $i$  we mean the output of the processor when  $x_1, \dots, x_i$  are fixed. One block of processor  $i$  consists of element  $x_i$  repeated  $n^{m-i}$  times.

It is easy to check that each block of processor  $i$  has the same size  $n^{m-i}$ . In lexicographic order, blocks producing 0 ( $n^{m-i}$  times), 1 ( $n^{m-i}$  times), ...,  $n-1$  ( $n^{m-i}$  times) follow each other in a cyclic manner. This order of blocks will be referred to as the increasing order. In a reflected Gray code order, if  $X_i = x_1 + x_2 + \dots + x_{i-1}$  is even then blocks are listed in increasing order, since, according to the definition,  $x_1 x_2 \dots x_{i-1} x'_i \dots < x_1 x_2 \dots x_{i-1} x''_i \dots$  iff  $x'_i < x''_i$ . On the other hand, if  $X_i = x_1 + x_2 + \dots + x_{i-1}$  is odd then blocks are listed in decreasing (reverse) order:  $n-1$  ( $n^{m-i}$  times),  $n-2$  ( $n^{m-i}$  times), ..., 0 ( $n^{m-i}$  times), since, according to the definition,  $x_1 x_2 \dots x_{i-1} x'_i \dots < x_1 x_2 \dots x_{i-1} x''_i \dots$  iff  $x'_i > x''_i$ . Each series of blocks in increasing order is followed by a series of the same blocks in the reverse (decreasing) order, and vice versa. This follows from the change of parity of the sum  $X_i$  which, for the next block of processor  $i-1$ , becomes  $X_i \pm 1$ .

Moreover, the output of processor  $i$  can be described as being independent from the data in other processors. Such a behavior can be used to write down a simple parallel algorithm for generating variations. It is derived from the algorithm [7] by generating variations  $x_1 x_2 \dots x_m$  in lexicographic order, and producing as output the sequence  $y_1 y_2 \dots y_m$  where  $y_i = x_i$  or  $y_i = n-1-x_i$  depending on whether the blocks are currently generated in the increasing or reverse order, respectively. The algorithm [7] will be reviewed first, and then we will show how to modify it to generate variations in a minimal change rather than lexicographic order.

Let  $s = m - \lceil \log_n m \rceil$ . Processor  $s$  has a special role in the algorithm. For  $i \geq s$  we have  $n^{m-i} \leq n^{m-s} = \frac{\lceil \log_n m \rceil}{n} \log_n m \cdot n^\epsilon = m n^\epsilon < mn$  (for some  $\epsilon$ ,  $0 \leq \epsilon < 1$ ). Therefore the number of repetitions of  $x_i$  in each block of processor  $i$ , for  $i \geq s$ , is less than  $mn$ , and the processors may use local counters of size  $O(\log mn)$  to produce such blocks. Because  $m = O(n^c)$ , the amount of space required per processor for this purpose is  $O(\log n)$ , thus satisfying property 5. In this way the generation problem is solved for processors  $i \geq s$ .

By contrast, processors  $i$ , where  $i < s$  do not use local counters for repetitions. Instead, they continually produce the same element unless they are informed to change their output. A message for a major update in processors  $i$ ,  $i < s$ , is initiated by processor  $s$  whenever it produces the value  $n-1$  for the first time. Processor  $s$  then begins a block with  $n^{m-s}$  repetitions of  $n-1$ . The message is forwarded toward processors  $s-1$ ,  $s-2$ , ..., 1, advancing one step per variation. Processors  $i$  receiving the message will forward it if  $x_i = n-1$ , and stop forwarding if  $x_i < n-1$ . This satisfies the property that the variation which follows  $v \ n-1 \ n-1 \dots \ n-1$  in lexicographic order is  $v+1 \ 0 \ 0 \dots \ 0$ , for  $v < n-1$ . The message will leave the new value in each processor, as well as the waiting time, i.e. information about when the major update becomes effective. Because  $n^{m-s} = \frac{\lceil \log_n m \rceil}{n} \log_n m \cdot n^\epsilon = m n^\epsilon \geq m > s$ , there is enough time for the message to visit all processors  $i < s$  before the end of current block of processor  $s$ .

What remains is to show how to terminate the algorithm simultaneously in all processors, immediately after the last variation is produced. A termination flag is initiated by processor 1 when it produces the value  $n-1$  for the first time. Each processor

$i, i > 1$ , will "accept" the termination flag when it also produces the value  $n-1$  for the first time. This way the termination flag is spread over all processors, and they will terminate simultaneously when they are supposed to produce 0 for the first time after producing  $n-1$ . Note that the termination flag has enough time to reach all processors since  $n^{m-1} > m$ . The termination procedure will remain unchanged in our modified algorithm.

The algorithm [7] uses the following variables. The variation which is generated is denoted  $x = x_1 x_2 \dots x_m$ . Counter for repetitions, from 0 to  $n^{m-i}$ , is denoted  $r_i$ . The future value of  $x_i$  and the waiting time for it are named  $f_i$  and  $w_i$ , respectively. The termination flag is denoted  $t_i$ . This algorithm introduces new variables  $p_i$  and  $fp_i$ . Variable  $p_i$  indicates whether the current blocks of processor  $i$  are produced in increasing  $(0, 1, \dots, n-1, p_i = \text{true})$  or decreasing  $(n-1, n-2, \dots, 0, p_i = \text{false})$  order. Variable  $fp_i$  denotes the future value of  $p_i$ .

This algorithm, modified to generate variations in a minimal change rather than lexicographic order, can be formally written as follows. Each processor  $i$ , from 1 to  $m$ , runs the following program.

```

{ calculate  $s$ ;
 $x_i \leftarrow 0$ ;  $r_i \leftarrow 0$ ;  $w_i \leftarrow 0$ ;  $f_i \leftarrow n$ ;  $t_i \leftarrow 0$ ;  $p_i \leftarrow \text{true}$ ;
if  $i \geq s$  then calculate  $n^{m-i}$ ;
repeat
0.      if  $p_i = 0$  then print out  $x_i$  else print out  $n-1-x_i$  ;
1.      if  $i \geq s$  then {  $r_i \leftarrow r_i + 1$ ; if  $r_i = n^{m-i}$  then
           {  $x_i \leftarrow x_i + 1 \pmod n$ ;
             if  $x_i = 0$  then  $p_i \leftarrow \text{not } p_i$  ;  $r_i \leftarrow 0$  };
2.      if  $i < s$  and  $f_i = n$  and  $f_{i+1} < n$  and  $x_{i+1} = n-1$ 
           then {  $f_i \leftarrow x_i + 1 \pmod n$ ;
                 if  $f_i > 0$  then  $fp_i \leftarrow p_i$  else  $fp_i \leftarrow \text{not } p_i$  ;  $w_i \leftarrow w_{i+1}$  };
3.      if  $i = s$  and  $x_i = n-1$  and  $f_i = n$ 
           then {  $f_i \leftarrow 0$ ;  $fp_i \leftarrow \text{not } p_i$  ;  $w_i \leftarrow n^{m-i} + 1$  };
4.      if  $i = 1$  and  $x_i = n-1$  then  $t_i \leftarrow 1$ ;
5.      if  $w_i = 1$  then {  $x_i \leftarrow f_i$ ;  $p_i \leftarrow fp_i$ ;  $w_i \leftarrow 0$ ;  $f_i \leftarrow n$  };
6.      if  $i > 1$  and  $t_{i-1} = 1$  and  $x_i = n-1$  then  $t_i \leftarrow 1$ ;
7.      if  $w_i > 1$  then  $w_i \leftarrow w_i - 1$ ;
until  $t_i = 1$  and  $x_i = 0$  }

```

In the above algorithm **if** statements numbered 1-7 correspond to updating values in processors  $i \geq s$ , forwarding the message, initiating the message, initiating the termination flag  $t_i$  by processor 1, setting the new value for  $x_i$  (major update),

“accepting” the termination flag, and decreasing the waiting time, respectively. These statements correspond to statements from algorithm in [7].

The modifications to algorithm in [7] are the following. If statement numbered 0 is introduced which determines whether to print  $x_i$  or  $n-1-x_i$ , which depends on whether the current blocks of processor  $i$  are generated in increasing ( $p_i=true$ ) or decreasing ( $p_i=false$ ) order, respectively. The instruction **if**  $x_i = 0$  **then**  $p_i \leftarrow \text{not } p_i$  is added to **if** statement numbered 1 to determine, for processors  $i \geq s$ , when the blocks change their “orientation”, i.e. become decreasing after being increasing and vice versa (it occurs whenever, in lexicographic order, the value of  $x_i$  changes from  $n-1$  to 0).

The instruction **if**  $f_i > 0$  **then**  $fp_i \leftarrow p_i$  **else**  $fp_i \leftarrow \text{not } p_i$  is added to **if** statement numbered 2. It determines the future value of  $p_i$ ; the value changes only when  $f_i$  becomes 0. Similarly, the instruction  $fp_i \leftarrow \text{not } p_i$  is added to **if** statement numbered 3. In addition to the new value for  $x_i$ , the new value of  $p_i$  is also due ( $p_i \leftarrow fp_i$ ) in **if** statement numbered 5. **If** statements 4, 6 and 7 and the termination criteria from [7] remain unchanged.

### 3. Adaptive Algorithm

The variation generation algorithm can be made adaptive, i.e. to run on a linear array consisting of an arbitrary number  $k$  of processors, in a similar way as other algorithms for generating combinatorial objects in parallel (e.g. [8,12-17]). If  $k < m$  then each processor will do the job of  $m/k$  processors in the original algorithm (all numbers here are rounded, with some details left out). Otherwise, for  $k \geq m$ , the processors are divided into  $r=k/m$  groups of  $m$  processors each, such that each group produces an interval of consecutive variations. The first and the last variation in each group can be determined in a preprocessing step by applying known unranking functions (i.e. functions mapping integers 1, 2, ...,  $n^m$  to variations). After determining the initial variations, the preprocessing should include finding the values of all variables at the time of generating the initial variation for the group. We omit the details since they are straightforward. However, the obvious unranking function involves very large integers. A scheme that does not deal with large integers and yet divides the job evenly among groups is described for variations in lexicographic orders in [18]. Here we describe analogous procedure for variations in reflected  $n$ -ary Gray code order.

The group  $j$ ,  $1 \leq j \leq r$ , will produce objects numbered from  $\lfloor (j-1)V(n,m)/r \rfloor + 1$  to  $\lfloor jV(n,m)/r \rfloor$ . Thus we need to find the variation indexed  $\lfloor gV(n,m) \rfloor + 1$  for  $g=j/r$ ,  $0 \leq j \leq r$ , to find the beginning and the end of each group. Let  $g=0.a_1a_2\dots a_m a_{m+1}\dots$  be the number  $g$  written in the number system with the base  $n$ , i.e.  $0 \leq a_i \leq n-1$  for  $1 \leq i \leq m$ . Then the variation indexed  $\lfloor gV(n,m) \rfloor + 1$  is coded as  $b_1b_2\dots b_m$  where  $b_1=a_1$ ,  $b_i=a_i$  if  $a_1+a_2+\dots+a_{i-1}$  is even and  $b_i=n-1-a_i$  otherwise ( $2 \leq i \leq m$ ).

### 4. Generating Variations from an Arbitrary Set

As a further generalization of our algorithm, we propose the problem of

generating variations of  $m$  elements out of  $n$  items of any kind (the solution presented here runs on the set  $\{0, 1, \dots, n-1\}$  only) such that properties 1-5 are preserved. In [19] such an extension is described for the case of variations in lexicographic order. The same extension can be applied for the case of variations in the reflected Gray code order. The extension is as follows [19].

To generate variations out of an arbitrary set  $S = \{s_0, s_1, \dots, s_{n-1}\}$ , we assume that processor  $m$  stores the whole set  $S$  and supplies other processors with the elements that they should produce. Processor  $m$  sends all data from  $S$  to processors  $m-1, m-2, \dots, 1$  in a cyclic manner. At each step (that corresponds to producing a variation) processor  $m$  sends the datum from  $S$  whose index is one greater than in the previous step (modulo  $n$ ). Processor  $i$ , for  $i < m$ , will read at each step the element from processor  $i+1$ . This process enables the circulation of elements from  $S$ , with period  $n$ . The number of variations in a given block of processor  $i$  is at least  $n$  for  $i < m$  (more precisely, it is  $n^{m-i}$ ). Thus there is enough time for each processor to receive the element it is supposed to produce after its current block is completed. The algorithm is omitted, since the extension does not differ from the extension presented in [19].

### Acknowledgement

The author is grateful to referees for careful reading and comments that greatly improved the clarity of the presentation of this paper.

### References

- [1] J.R. Bitner, G. Ehrlich and E.M. Reingold, Efficient generation of the binary reflected Gray code and its application, *Comm. ACM* 19, 1976, 517-521.
- [2] J. Misra, Remark on Algorithm 246, *ACM Trans. Math. Software*, 1, 1975, 185.[2] M. Cohn, Affine  $m$ -ary Gray codes, *Inform. Control* 6, 1963, 70-78.
- [3] M. Chon, Affine  $m$ -ary Gray codes, *Inform. Control* 6, 1963, 70-78.
- [4] B.D. Sharma, and R.K. Khanna, On  $m$ -ary Gray codes, *Information Sci.* 15, 1978, 31-43.
- [5] I. Flores, Reflected number systems, *IRE Trans. Electron. Comput.*, EC-5, 1956, 79-82.
- [6] M.C. Er, On generating the  $n$ -ary reflected Gray codes, *IEEE Trans. Comput.* 33, 8, 1984, 739-741.
- [7] S.G. Akl, T. Duboux, and I. Stojmenovic, Constant delay parallel counters, *Parallel Processing Letters*, 1, 2, 1991, 143-148.
- [8] I. Stojmenovic, An optimal algorithm for generating equivalence relations on a linear array of processors, *BIT* 30, 3 (1990) 424-436.
- [9] Thangavel P., and Muthuswamy V.P., A parallel algorithm to generate  $N$ -ary reflected Gray codes in a linear array with reconfigurable bus system, *Parallel Processing Letters*, 3, 2, 1993, 157-164.
- [10] S.G. Akl, *The Design and Analysis of Parallel Algorithms* (Prentice Hall, Englewood Cliffs, New Jersey, 1989).
- [11] S.G. Akl and I. Stojmenovic, A simple optimal systolic algorithm for generating permutations, *Parallel Processing Letters*, 2, 2/3, 1992, 231-239.
- [12] S.G. Akl, D. Gries, and I. Stojmenovic, An optimal parallel algorithm for generating combinations, *Inform. Process. Lett.*, 33 (1989/90) 135-139.
- [13] H. Elhage and I. Stojmenovic, Systolic generation of combinations from arbitrary elements, *Parallel Processing letters* 2, 2&3, 1992, 241-248.
- [14] S.G. Akl, J. Calvert, and I. Stojmenovic, Systolic generation of derangements, in:

Algorithms and Parallel VLSI Architectures II, Elsevier Sci. Publ., 1992, 59-70.

- [15] S.G. Akl, and I. Stojmenovic, Generating binary trees in parallel, Allerton Conference on Communication, Control and Computing, Monticello, Ill., USA, 1992, 225-233.
- [16] S.G. Akl, and I. Stojmenovic, Parallel algorithms for generating integer partitions and compositions, *J. Combinatorial Math. and Combinatorial Comput.*, 13, 1993, 107-120.
- [17] S.G. Akl, H. Meijer, and I. Stojmenovic, An optimal systolic algorithm for generating permutations in lexicographic order, *J. Parallel and Distr. Comput.*, 20, 1, 1994, 84-91.
- [18] I. Stojmenovic, On random and adaptive parallel generation of combinatorial objects, *Int. J. Computer Math.*, 42, 4, 1992, 125-135.
- [19] S.G.Akl and I. Stojmenovic, Generating combinatorial objects on a linear array of processors, in: *Parallel Computations: Paradigms and Applications* (A.Y. Zomaya, ed.), Chapman & Hall, to appear.