Parallel binary reflected Gray code sequence generation on multicore architectures

Md. Mohsin Ali a & Mst. Shakila Khan Rumi b

a The Australian National University (ANU), Canberra, ACT, 0200, Australia
b Khulna University of Engineering and Technology (KUET), Khulna, 9203, Bangladesh

Published online: 13 Aug 2013.

To cite this article: International Journal of Parallel, Emergent and Distributed Systems (2013): Parallel binary reflected Gray code sequence generation on multicore architectures, International Journal of Parallel, Emergent and Distributed Systems, DOI: 10.1080/17445760.2013.822498

To link to this article: http://dx.doi.org/10.1080/17445760.2013.822498

PLEASE SCROLL DOWN FOR ARTICLE
Parallel binary reflected Gray code sequence generation on multicore architectures

Md. Mohsin Ali\textsuperscript{a*} and Mst. Shakila Khan Rumi\textsuperscript{b}

\textsuperscript{a}The Australian National University (ANU), Canberra, ACT 0200, Australia; \textsuperscript{b}Khulna University of Engineering and Technology (KUET), Khulna 9203, Bangladesh

(Received 26 March 2013; final version received 2 July 2013)

We propose a novel parallel algorithm for generating all the sequences of binary reflected Gray code for a given number of bits as input, targeting machines with multicore architectures. A theoretical analysis of work and span, as well as parallelism of this algorithm, is carried out following a multithreaded implementation using Cilk++ on a multicore machine. Theoretical analysis of this algorithm shows a parallelism of $\Theta(2^n/\log n)$ and achieves a linear speedup on 12 cores for input data of sufficiently large size.

\textbf{Keywords:} Gray code; work; span; parallelism; speedup; multithreading; multicore; Cilk++

1. Introduction

Binary reflected Gray code sequence is a kind of binary sequence in which two successive portions of the sequence differ only in 1-bit position. For example, the binary Gray code sequence for 3 bits is 000, 001, 011, 010, 110, 111, 101 and 100. Sometimes this sequence is also known as binary Gray code sequence and these two are used interchangeably in the literature. Historically, the name ‘Gray code’ came from its inventor, Frank Gray, who patented its uses in shaft encoders in 1953 [12]. It was originally designed to prevent spurious output from electromechanical switches. At that time, logic circuits were designed from vacuum tubes and electromechanical relays which enforced time counters to demand more power supply and generating noise spikes since many bits were changed simultaneously. In contrast, in Gray code sequence, only 1 bit is changed whether we increment or decrement the value, regardless of the size of the number. Therefore, it minimises the effect of noise. Applications of Gray code include mechanical position sensors to convert the angular position of a disk to a digital form [12], solving puzzles such as the Tower of Hanoi and the Brain [9], Hamiltonian circuits in hypercubes [10], Cayley graphs of Coxeter groups [4], capanology [23], continuous space-filling curves [11], classification of Venn diagrams [20] and so on. Today, Gray codes are widely used for correcting errors in digital communication systems.

The evolution of computer architecture within the last 10 years enforces a revisit of fundamental algorithms in computer science. Especially, parallelism and data locality are becoming major measures of performance, in addition to the traditional ones, namely serial running time and allocated space. In this paper, we present the design, implementation, analysis and experimentation of a novel parallel algorithm for generating the complete binary Gray code sequence $G(i)$, $0 \leq i \leq 2^n - 1$, for a given number of bits $n$,
targeting multicore architectures. The experimental result of this multithreaded algorithm shows a linear speedup on that architecture for sufficiently larger data as input. Since generating binary reflected Gray code sequence is a subset of one of the particular combinatorial problems, algorithms for efficiently generating such sequences are one of the most interesting research problems. This parallel implementation can encourage those algorithm designers to think a lot to implement them on a multicore parallel environment.

The rest of the paper is organised as follows. Related work is presented in Section 2. Section 3 describes the proposed approach. Theoretical analysis and experimental result of the proposed algorithm are presented in Sections 4 and 5, respectively. Finally, Section 6 concludes the paper.

2. Related work

The study of Gray code sequences and the algorithms for generating such sequences are found in [1–3,5–7,15,16,18,19,21,22]. Among these, [5,7] are dedicated to extensive study of Gray code sequences; [1–3,6,15,16,18,19] are for the serial version of algorithms for generating such sequences and, lastly, [21,22] describe parallel approaches for computing such sequences. Algorithm in [3] requires a total of $O(n^2)$ operations for the generation of $2^n$ n-bit sequence. Lenstra and Kan [15], Peelle [18] and Er [6] proposed algorithms based on the straightforward recursive definition of the binary Gray code. Simple iterative solutions for generating Gray code sequence are discussed in [2,16]. On the other hand, Phillips and Wick [19] also proposed a recursive approach for generating a complete binary Gray code sequence. This approach is very efficient for generating only one Gray code value with running time of $O(n)$, but for $n$-bit sequence generation it needs $O(n^2)$ time, similar to that of [3]. To improve the efficiency of this approach, Phillips and Wick [19] proposed a dynamic programming approach for generating the same sequence. Both the space and time complexity of this approach is within a constant multiple of the optimal time and space complexity of $O(2^n)$. In order to improve the efficiencies of this approach, Phillips and Wick [19] and Ali et al. [1] proposed space and time optimal approaches with space and time complexity of $O(2^n)$, which is optimal for the generation of a complete $n$-bit Gray code sequence. On the other hand, parallel algorithms for generating $n$-ary reflected Gray code sequence on a linear array of processors, say $m$ processors, are discussed in [21,22]. Until now there is no algorithm for generating Gray code sequence targeting multicore architectures and our main contribution is to propose such an algorithm.

3. Proposed approach

It is observed from Figure 1 that the 4-bit binary reflected Gray code sequence can be represented by a binary tree data structure [1]. The properties of this tree are as follows. Assume that a leaf is considered to be a sub-tree.

- Visiting the right sub-tree from the root generates ‘0’ in its right sub-tree.
- Visiting the left sub-tree from the root generates ‘1’ in its left sub-tree.
- Visiting the right sub-tree from the root of any right sub-tree generates ‘0’ in its right sub-tree.
- Visiting the left sub-tree from the root of any right sub-tree generates ‘1’ in its left sub-tree.
- Visiting the left sub-tree from the root of any left sub-tree generates ‘0’ in its left sub-tree.
- Visiting the right sub-tree from the root of any left sub-tree generates ‘1’ in its right sub-tree.
Data, either ‘0’ or ‘1’, in any level of the tree can be generated concurrently without any dependency in the corresponding level.

As for example, node 1 is the root of the original tree and visiting its right sub-tree, rooted at node 2, generates all ‘0’s in its right sub-tree. Similarly, node 2 is the root of the right sub-tree with respect to node 1, and visiting its right sub-tree, rooted at node 3, generates all ‘0’s in its right sub-tree. Alternatively, node 9 is the root of the left sub-tree with respect to node 1, and visiting its left sub-tree, rooted at node 13, generates all ‘0’s in its left sub-tree. If visiting the right sub-tree generates ‘0’s, then visiting the left sub-tree of the corresponding node generates ‘1’s. These properties are also applicable for nodes 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14 and 15.

These properties can be used to design a multithreaded parallel algorithm such as Algorithm 1. This algorithm takes the following as input:

\[\text{ParaGrayCode}(A, m, n, \text{direction})\]

\[\text{Input:}\] First call ParaGrayCode\((A, 1, n, \text{right})\), where \(A\) is a 0-initialised \(2^n \times n\) matrix and \(\text{direction} \in \{\text{left}, \text{right}\}\).

\[\text{Output:}\] Return \(2^n \times n\) matrix \(A\) with starting index \((1, 1)\) containing \(n\)-bit binary Gray code sequences.

1: if \(n = 0\) then
2: \(\text{return} \);
3: else
4: if \(\text{direction} = \text{right}\) then
5: \(A_{m, m+2^{n-1}} = 0\);
6: \(A_{m+2^n, m+2^{n-1}} = 1\); \(\text{in parallel}\)
7: else
8: \(A_{m, m+2^{n-1}} = 1\);
9: \(A_{m+2^n, m+2^{n-1}} = 0\); \(\text{in parallel}\)
10: \(\text{spawn ParaGrayCode}(A, m, n-1, \text{right})\)
11: \(\text{spawn ParaGrayCode}(A, m+2^n-1, n-1, \text{left})\)

As for example, node 1 is the root of the original tree and visiting its right sub-tree, rooted at node 2, generates all ‘0’s in its right sub-tree. Similarly, node 2 is the root of the right sub-tree with respect to node 1, and visiting its right sub-tree, rooted at node 3, generates all ‘0’s in its right sub-tree. Alternatively, node 9 is the root of the left sub-tree with respect to node 1, and visiting its left sub-tree, rooted at node 13, generates all ‘0’s in its left sub-tree. If visiting the right sub-tree generates ‘0’s, then visiting the left sub-tree of the corresponding node generates ‘1’s. These properties are also applicable for nodes 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14 and 15.

These properties can be used to design a multithreaded parallel algorithm such as Algorithm 1. This algorithm takes the following as input:
• a variable \( n \), representing number of bits in the binary Gray code sequence,
• a 2D array \( A = [A_{ij}]_{1 \leq i \leq 2^n, 1 \leq j \leq n} \), initialised to 0,
• a status variable \( \text{direction} \), initialised to 0, contains only 0 or 1, controlling if it is 
right sub-tree or left sub-tree,
• a variable \( m \), initialised to 1, used for specifying array index,

and returns the array \( A = [A_{ij}]_{1 \leq i \leq 2^n, 1 \leq j \leq n} \) with \( n \)-bit complete binary Gray 
code sequence as output. Although it seems that the variable \( n \) inside the algorithm is 
decreased by 1, this algorithm proceeds in a divide-and-conquer manner with respect to 
the generation of the sequence. As a result, this is one kind of divide-and-conquer 
multithreaded parallel algorithm.

4. Theoretical analysis
Theoretical performance of a multithreaded algorithm is analysed by its work, span and 
parallelism based on fork-join parallelism model which are described as follows.

4.1 The work law
The first important measure is the work which is defined as the total amount of time 
required to execute all the instructions of a given program.

Let \( T_P \) be the fastest possible execution time of the application on \( P \) processors. 
Therefore, we denote the work by \( T_1 \) as it corresponds to the execution time on one 
processor. Moreover, we have the following relation

\[
T_P \geq \frac{T_1}{P},
\]

which is referred as the work law. In our simple theoretical model, the justification of this 
relation is easy: each processor executes at most one instruction per unit time and, 
therefore, \( P \) processors can execute at most \( P \) instructions per unit time. Therefore, the 
speedup on \( P \) processors is at most \( P \) since we have

\[
\frac{T_1}{T_P} \leq P.
\]

4.2 The span law
The second important measure is based on the program’s critical-path length denoted by 
\( T_\infty \). This is actually the execution time of the application on an infinite number of 
processors or, equivalently, the time needed to execute threads along the longest path of 
dependency. As a result, we have the following relation, called the span law:

\[
T_P \geq T_\infty.
\]

4.3 Parallelism
In the fork-join parallelism model, parallelism is defined as the ratio of work to span, or 
\( T_1/T_\infty \). Thus, it can be considered as the average amount of work along each point of the 
critical path. Specifically, the speedup for any number of processors cannot be greater than 
\( T_1/T_\infty \). Indeed, Equations (2) and (3) imply that speedup satisfies \( T_1/T_P \leq T_1/T_\infty \).
4.4 Performance bounds

For an application running on a parallel machine with $P$ processors with work $T_1$ and span $T_\infty$, the Cilk++ work-stealing scheduler [13,14] achieves an expected running time as follows:

$$T_P = \frac{T_1}{P} + O(T_\infty) \quad (4)$$

under the following three hypotheses:

- Each strand executes in unit time.
- For almost all ‘parallel steps’ there are at least $P$ strands to run.
- Each processor is either working or stealing.

See [8] for details.

If the parallelism $T_1/T_\infty$ is so large that it sufficiently exceeds $P$, which is $T_1/T_\infty \gg P$, or equivalently $T_1/P \gg T_\infty$, then from Equation (4) we have $T_P \approx T_1/P$. From this, we easily observe that the work-stealing scheduler achieves a nearly perfect linear speedup of $T_1/T_P \approx P$.

Under the discussed fork-join parallelism model, the work, span and parallelism of Algorithm 1 is summarised in Proposition 4.1.

**Proposition 4.1.** If $n$ is the number of bits in the binary Gray code sequence, then for an input data of size $n$, the work, span and parallelism of Algorithm 1 are $\Theta(n2^n)$, $\Theta(n \log n)$ and $\Theta(2^n/\log n)$, respectively.

**Proof.** Let us denote by $W(n)$ (resp. $S(n)$) the work (resp. span) of Algorithm 1 on input data of order $n$.

Then the following two classes of executions have happened for the serial case (work) and parallel case (span).

- Serial case: lines 5 and 6 or lines 8 and 9 are executed in $\Theta(2^n)$ time with two recursive calls, one at line 10 and another at line 11, of input data size $(n-1)$.
- Parallel case: assume that execution is provided by the Cilk++ runtime [13,14]. Then, lines 5 and 6 or lines 8 and 9 are executed in $\log n$ time. Moreover, there are two recursive calls, one at line 10 and another at line 11, of input data size $(n-1)$, but can be considered as only one call due to spawn.

So, we can write the equations for work and span as follows:

$$W(n) = 2W(n - 1) + \Theta(2^n),$$
$$= 2[2W(n - 2) + \Theta(2^{n-1})] + \Theta(2^n)$$
$$= 2^2W(n - 2) + 2\Theta(2^{n-1}) + \Theta(2^n)$$
$$\vdots$$
$$= 2^kW(n - k) + 2^{k-1}\Theta(2^{n-k+1}) + \cdots + \Theta(2^n),$$
since \( W(0) = \Theta(1) \) for \( k = n \), we deduce:

\[
W(n) = 2^n \Theta(1) + 2^{n-1} \Theta(2) + 2^{n-2} \Theta(2^2) + \cdots + 2 \Theta(2^{n-1}) + \Theta(2^n)
\]

\[
= n \Theta(2^n)
\]

\[
= \Theta(n2^n),
\]

\[\text{(5)}\]

\[
S(n) = S(n - 1) + \Theta(\log n)
\]

\[
= S(n - 2) + \Theta(\log(n - 1)) + \Theta(\log n)
\]

\[
\vdots
\]

\[
= S(n - k) + \Theta(\log(n - k + 1)) + \cdots + \Theta(\log n),
\]

since \( S(0) = \Theta(\log 1) \) for \( k = n \), we deduce:

\[
S(n) = \Theta(\log 1) + \Theta(\log 2) + \cdots + \Theta(\log n)
\]

\[
= n \Theta(\log n)
\]

\[
= \Theta(n \log n).
\]

\[\text{(6)}\]

Finally, from the definition of parallelism in [8], we get the following from Equations (5) and (6):

\[
\text{Parallelism} = \frac{W(n)}{S(n)} = \Theta\left(\frac{2^n}{\log n}\right).
\]

This completes the proof.

5. Experimental result

Since the data of the 2D array, \( A \), of Algorithm 1 are generated in parallel in columnwise manner from left to right, we consider storing the data in that array as column-major order for implementation. This technique actually achieves highest spatial locality of the data, which is most important for multithreaded implementation. However, we have implemented our algorithm in the Cilk++ concurrency platform [13,14]. The speedup estimate (scalability analysis) of this algorithm is obtained by CilkView, a feature of Cilk++, by executing them in an Intel(R) Xeon(R) (64 bit) Machine, with CPU (X5680) speed 3.33 GHz, 24.0 GB of RAM, and having a total of 12 cores available in one of the OPL clusters [17].

Figure 2 shows the speedup estimate of Algorithm 1 for generating a complete 25-bit binary reflected Gray code sequence obtained from the median of five repetitions. It is observed that the speedup is almost linear with the number of cores of the machine. The reason is that this algorithm is efficient in terms of data locality and parallelism.

6. Conclusion

In this paper, we propose a novel multithreaded parallel algorithm for generating a complete binary reflected Gray code sequence targeting machines with multicore architectures. In this proposal, we design a divide-and-conquer algorithm, compute its theoretical performance and implement it in a parallel programming language in order to
experiment its actual performance behaviour on multicore machines. The experimental result of this algorithm shows that it achieves almost linear parallelism on multicore machines for sufficiently larger data as input.

References


[17] OPL: Open peta-scale library. Available at http://opl.dl.ac.uk


