# Constructing Optimal Golomb Rulers in Parallel

PCAP Lab Final Project

***Submitted by***

Sahil Garg - 160905058

Jenit Jain - 160905070

Amrit Goyal - 160905210

Semester - 6
Section - CSE D

Department of Computer Science & Engineering

MANIPAL INSTITUTE OF TECHNOLOGY
A Constituent Institute of Manipal University, Manipal

## Title

Constructing Optimal Golomb Rulers in Parallel using MPI and CUDA

## Research Domain

- CSP - Constraint Satisfaction Problems, Algorithms
- Tree search algorithms
- NP-Hard problems
- Backtracking of OGR-n (optimal golomb ruler of n divisions) problem
- Radio astronomy, information theory and error correction and detection

## Abstract

A CSP (Constraint satisfaction problem) is defined by a set of variables and a set of constraints. A set of allowed values (the domain) is associated to each variable. Solving a CSP means finding an assignment for each variable that satisfies all the constraints. Finite Domains constraint solving is now a well established technique. Typically, these problems require extensive computation to find a solution.

A Constraint Satisfaction Problem P is given as a tuple P = (X,D,C,R), where :
– X = {X1,...,Xn} is a set of n variables.
– D = {D1,...,Dn} is a set of n domains where each Di is asso-
ciated with Xi.
– C = {C1, . . . , Cm} is a set of m constraints where each constraint
Ci is defined by a set of variables {Xi1,...,Xini } ⊆ X.
– R = {R1, . . . , Rm} is a set of m relations where each relation Ri
defines a set of ni-tuples on Di1 × · · · × Dini compatible w.r.t. Ci.

A CSP can be solved using either of the following ways -
- performing constraints propagation before or during search with a number of different filtering techniques
- improving the search by choosing good variable or value ordering heuristics for the next variable to be instantiated and value to be assigned
- improving the search by a more intelligent backtracking, some look-ahead strategies or a combination of them
- performing subproblems decomposition

The golomb rulers problem is one famous example of a CSP. Intuitively, a Golomb Ruler may be thought of as a specific type of ruler: whereas a typical ruler has a mark at every unit measure, a Golomb ruler will only have marks at some subset of these mea-

sures. The distance between any two marks on the ruler is different than any other two marks. Any ruler that possesses this trait is called ‘Golomb’.

More formally, we have that a Golomb ruler is a set,

A = {a_1, a_2, ... ,a_N} where a_1<a_2<...<a_N

having the property that the difference,

a_i - a_j (i < j)

is distinct for all pairs (i,j) .

All known OGRs as of December 2010 are as follows. For *n* ≥ 23 all OGRs have been discovered by the massively distributed computation effort orchestrated by the distributed.net *OGR* project.

| *n* | Optimal Golomb Ruler |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 13 |
| 4 | 146 |
| 5 | 0 1 4 9 11 |
| 6 | 0 1 4 10 12 17 |
| 7 | 0 1 4 10 18 23 25 |
| 8 | 0 1 4 9 15 22 32 34 |
| 9 | 0 1 5 12 25 27 35 41 44 |
| 10 | 0 1 6 10 23 26 34 41 53 55 |
| 11 | 0 1 4 13 28 33 47 54 64 70 72 |
| 12 | 0 2 6 24 29 40 43 55 68 75 76 85 |
| 13 | 0 2 5 25 37 43 59 70 85 89 98 99 106 |

| | |
|---|---|
| 14 | 0 4 6 20 35 52 59 77 78 86 89 99 122 127 |
| 15 | 0 4 20 30 57 59 62 76 100 111 123 136 144 145 151 |
| 16 | 0 1 4 11 26 32 56 68 76 115 117 134 150 163 168 177 |
| 17 | 0 5 7 17 52 56 67 80 81 100 122 138 159 165 168 191 199 |
| 18 | 0 2 10 22 53 56 82 83 89 98 130 148 153 167 188 192 205 216 |
| 19 | 0 1 6 25 32 72 100 108 120 130 153 169 187 190 204 231 233 242 246 |
| 20 | 0 1 8 11 68 77 94 116 121 156 158 179 194 208 212 228 240 253 259 283 |
| 21 | 0 2 24 56 77 82 83 95 129 144 179 186 195 255 265 285 293 296 310 329 333 |
| 22 | 0 1 9 14 43 70 106 122 124 128 159 179 204 223 253 263 270 291 330 341 353 356 |
| 23 | 0 3 7 17 61 66 91 99 114 159 171 199 200 226 235 246 277 316 329 348 350 366 372 |
| 24 | 0 9 33 37 38 97 122 129 140 142 152 191 205 208 252 278 286 326 332 353 368 384 403 425 |
| 25 | 0 12 29 39 72 91 146 157 160 161 166 191 207 214 258 290 316 354 372 394 396 431 459 467 480 |
| 26 | 0 1 33 83 104 110 124 163 185 200 203 249 251 258 314 318 343 356 386 430 440 456 464 475 487 492 |

## Motivation

The problem under discussion is a combinatorial optimisation problem, believed (although not yet proven) to be NP-Hard and even the most efficient parallel algorithms (and their implementations) of today, require years of running time to solve instances with n > 24. Besides exposing ourselves to the already known embarrassingly parallel nature of the OGR-n problem, our parallelisation efforts will additionally allow us for arbitrary selection of the amount of work assigned to each computational node. For the optimal Golomb ruler construction problem, all the sequences are not equivalent : we will have to minimise the length of the constructed solutions as a cost function.

## Objectives

Our main purpose in this project is to solve the OGR-*n* problem with a parallel algorithm. We first obtain the design of a sequential algorithm for OGR-*n*, which we will subsequently go on to parallelise in the future stages.

In this project, a construction of the Golomb optimal rulers is studied with a tree search approach. Improvements to the basic algorithm are understood and it is parallelised using global as well as shared memory. The application associated to this approach is written in C using the standard CUDA and MPI libraries. The algorithm will use collaborative mechanisms between processors with effective load balancing.

Main metrics for an efficient parallel algorithm we aim to achieve are -
1. Performance - Good sequential performance does not necessarily assure good parallel performance. However if a highly optimized sequential algorithm scales easily to a parallel implementation then it may well provide the best parallel performance also.

2. Scalability - The parallel implementation of the algorithm should take maximum advantage of all available processors and not limit itself to fixed number.

3. Portability - The algorithm should not be dependent on any specific processor architecture or feature.

4. Low Communications Overhead - Since the processor groups that are used by PVM might not b e tightly coupled there may be significant overhead involved in interprocess communications Any parallel implementation should strive to minimize this overhead.

## Introduction

Golomb rulers are numerical sequences. The Optimal Golomb Rulers (OGR) construction is a hard combinatorial problem that consumes a high computation time. It represents a real challenging problem especially for parallel combinatorial search.

The term Golomb Ruler derived its name from the relevant work by Professor Solomon W Golomb of the University of Southern California. The search for Golomb Rulers is a combinatorial algorithm whose bounds can be shown to grow geometrically with respect to the solution size. This has been a major limitation in the discovery of new rulers since each new ruler is by necessity larger that its predecessor The search is bounded however and therefore solvable. By carefully reviewing existing sequential algorithms and applying parallel programming techniques the problem can be reduced to a workable solution.

Golomb rulers have a wide variety of other applications, including radio astronomy and information theory. In radio astronomy astronomers use arrays of radio telescopes in a single line to collect data on particular stars or celestial regions. More information may be extracted if there are multiple difference measurements between the telescopes. By placing the telescopes at the marks of a Golomb ruler, the number of distance measurements is maximized.

In information theory, Golomb rulers are used for error detection and correction. Golomb rulers of the same length are used to generate self-orthogonal codes, codes that do not share common differences. Such codes allow for easier error checking since redundancies between codes will show up as errors and may then be resent or corrected.

An optimal Golomb ruler is a Golomb ruler of shortest possible length dependant on the number of marks. G(n) denotes this length for a ruler with n marks. In mathematics, a Golomb ruler (named after Solomon Golomb) is a set of integers starting from zero ($x1 = 0 < x2 < x3 \ldots < xn$), selected such that all their cross differences $xij : i \neq j$ are distinct. A Golomb ruler is said to be Optimal when it has been formed so that x_n (the ruler's length) is as small as possible.

## Literature review

A Golomb ruler can be described as a collection of markings spaced apart at integer multiples of a unit length, so that all distances between pairs of markings are distinct. This is equivalent to saying that, when a Golomb ruler is superimposed to a shifted version of itself (by an integer multiple of the unit length), at most one pair of markings will coincide: in engineering terminology, Golomb rulers have an 1D ideal thumbtack autocorrelation (or auto-ambiguity).

Golomb rulers are the same objects as Sidon sets (which historically predate Golomb rulers), namely sets of integers such that all pairwise sums of elements of the set are distinct.

A 2D object related to Golomb rulers is the Costas array, namely a square arrangement of dots and blanks such that there is exactly one dot per row and column (i.e. a permutation array), and such that no four dots form a parallelogram and no three dots lying on the same straight line are equidistant.

There exist multiple papers with relevant definitions and construction methods for Golomb rulers, and summarising these findings, we obtain the following derived theorems and rules -

- Let $m,n \in N$, and let $f : [m-1] \rightarrow [n]$ be injective with $f(0) = 0$, $f(m-1) = n$ (whence $m \leq n+1$); f is a Golomb ruler of length n with m markings if and only if ,$\forall$ $i,j,k,l \in [m-1]$, $f(i)-f(j)=f(k)-f(l) \Leftrightarrow i=k \wedge j=l$. If, in addition, there exists some $N \in N$ such that $\forall$ $i,j,k,l \in [m-1]$, $f(i)-f(j) \equiv f(k)-f(l) \mod N \Leftrightarrow i=k \wedge j=l$, f will be called a Golomb ruler modulo N, or simply a modular Golomb ruler.

- Geometrically, the Golomb ruler can be interpreted as a sequence of dots (called the markings of the ruler) and blanks, so that all distances between pairs of dots are distinct; the positions of the dots in the sequence (which we will also be calling a Golomb ruler, as there is no danger of confusion) correspond to the range of f.
- Erdo¨s-Turan construction : For every odd prime p, the sequence - 2pk+(k^2 mod p), k∈[p−1] forms a Golomb ruler.
- For any n ∈ N∗ (positive natural numbers), and for a fixed c ∈ {1, 2}, the sequence cnk^2+k, k∈[n−1], forms a Golomb ruler.
- Focusing on optimal Golomb rulers of more than 10 markings, we learn, in particular, that those for 11, 12, 14, 18, 19, 20, 21, 23, and 24 markings are obtained by the Singer construction (possibly by truncating the final marking), those for 17 and 22 markings by the Bose-Chowla construction, while only those for 13, 15, and 16 markings were new rulers found by computer search.
- The construction algorithms for Optimal Golomb rulers are the following -
    - <u>Scientific American algorithm</u> : The basic algorithm is composed of two sections a ruler procedure The generation procedure constructs a sequence of rulers each of which are passed to the verification procedure which determines whether or not the sequence meets the criteria of a Golomb Ruler.
    - <u>Exhaust</u> : The generation procedure, called Exhaust, requires two parameters The first parameter is the number of marks contained in the desired Golomb Ruler. The second parameter sets an upper bound on the length of the ruler being searched for. If this boundary is too low, no ruler will be found. The Exhaust procedure is recursive in nature. A Golomb Ruler candidate is constructed by taking an existing n mark Golomb Ruler and appending a new mark onto the right side of the ruler generating an n+1 mark ruler.
    - <u>Token passing algorithm and the shift algorithm</u> : They are modified versions of the first 2 algorithms mentioned above to improve efficiency by restricting the number of combinations generated for the ruler.
    - <u>Tree Algorithm</u> : The search space is represented as a general tree structure with nodes representing marks and the edges representing distances. Finding a valid mark consists of locating a node at the proper depth. The tree is built up by applying the Shift Algorithm to each of the nodes of the tree, checking to see if new nodes can be added.

## Methodology (Algorithm)

This project aims to develop an application using parallel processing APIs - MPI and CUDA, that can generate an optimal golomb ruler for a particular number of divisions in the most optimum and effective manner with the help of load balancing and distributed memory. The tree search based algorithm is used for this.
The construction of optimal Golomb rulers can be modelized as a tree search problem, with the view to minimize the length of the constructed sequence:

- as the length of an optimal n marks ruler does not exceed 1+2+4+⋯+2^n−2 = 2^(n−1)−1, it is possible to consider that the root value is 0 and that the values of the nodes are between 1 and 2^(n−1) – 2;
- every leaf of the tree symbolizes a sequence which is a solution if the values are ordered and if all the differences are different.

It is now sufficient to have a walk through the search tree, in depth first, keeping the best observed length, to get a simple sequential algorithm finding an optimal Golomb ruler.
This constructive algorithm belongs to the backtrack algorithms class; to be efficient, any recursive tree traversal must be avoided and the use of arrays as elementary data structure is preferred.

The search space can be reduced by applying some simple construction remarks:

- the best possible length best_length is set to initial_limit = 2^n – 1 at the beginning (or it can be fixed at the execution time)
- when placing the k-th mark at position pos(k), there are still r = n – k remaining marks to be placed, which can't have a length remaining_length(r) less than 1+2+⋯+r = r(r+1)/2 or than G(r + 1): it induces the following new constraint:
    - pos(k) + max( r(r + 1) , G(r + 1)) < best_length
- mirror solutions can be avoided and thus the search tree reduced, by considering only sequences whose last distance is greater than the first one: if the second mark (first except 0) has already been set with the a value, and in order for the last distance b to satisfy b ≥ a + 1, another constraint has to be introduced :
    - pos(k)+max( (r – 1)r , G(r))+a+1 < best_length
- as the best observed length decreases, more and more inconsistent values can be ignored.

The algorithm can be summarized in c-like mode by:

```
nbTasks = generateTasks(n,max,k);

best_length = initial_limit;

for(task=0 ; task<nbTasks ; task++)

if ( useful(task,best_length) )

 best_length = solveTask(task,best_length);
```

where nbTasks is the number of initially useful tasks, deduced by the development of all subtrees to the depth k by the function generateTasks for n marks with initial_limit as initial best length. The function solveTask is in charge of traversing the subtree associated with the task numbered task. At the end, the variable best_length contains the optimal Golomb ruler length for the given value of n.

The **algorithm using a single processor (Serial implementation using a CPU)** is -

**Problem 4.1 (OGR-*n*)** Given positive integer $n$, find and return an Optimal Golomb

**Algorithm 4.1 (General GR-*n* algorithm)**

```
1. Given positive integer n:
2. Calculate L(n), some lower bound for the length of a Golomb ruler with n marks
3. For each integer K ≥ L(n) :
4. Solve problem instance GR-n, K (find if a golomb ruler is possible with length k for n divisions) - using backtracking search
5. If a Golomb ruler has been found : return found ruler, quit search
```

**Problem 4.2 (GR-*n*, *K*)** Given positive integers $n$, $K$ , find and return a Golomb ruler with $n$ marks and length $K$ , or failure if that is not possible.

**Algorithm 4.2 (General GR-*n*, *K* algorithm)** Given positive integers $n$, $K$ :

```
1. Let the first mark fixed at distance zero
2. Let the n-th mark fixed at distance K

3. For each possible configuration of marks between the first and the n-th mark:

4.If a Golomb ruler has been formed: return ruler, quit search

5. return failure
```

To find L(n), the lower bound for the length of the Golomb ruler with n marks,

$L(n) = \max(G(n-1) + 1, n(n-1)/2 + 1, n^2 - 2n\sqrt{n} + \sqrt{n} - 2)$

The **algorithm in a parallel processing interface such as CUDA** can be described as-

```
algorithm CUDA-OGR-SEARCH(n, G[])
1. Initialize K according to Equation 1 as given below.
2. do
        1. Create P pieces
        2. Launch CUDA kernel; each thread runs Algorithm PIECE-GR-n,k on it's piece
        3. if search space of rulers with length K exhausted
        4. Increase K
3. while Golomb ruler of length K not found
4. — Found Golomb ruler is guaranteed to be optimal
5. Output found OGR
```

Where, equation 1 is -

$K = x[n] \geq L(n) = \max(G(n-1)+1, n(n-1)/2+1, n^2 - 2n\sqrt{n} + \sqrt{n} - 2)$

And **algorithm PIECE-GR-n,K** is -

```
Given positive integers n, K and rulers start[1, . . . , n], end[1, . . . , n]:
1. Let the first mark fixed at distance zero
2. Let the n-th mark fixed at distance K
3. For each possible configuration of marks between the first and the n-th mark, within piece boundaries:
        4.If a Golomb ruler has been formed:
                5. return ruler, quit search
6. return failure
```

The **algorithm in a parallel processing interface such as MPI** can be described as-

```
MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &nbp);

MPI_Comm_rank(MPI_COMM_WORLD, &p); if ( p == 0 ) { /* server */

nbTasks = generateTasks(n, max,k);

global_best = initial_limit; initialSends();
while ( (nextTask < nbTasks)) {

/* get the result */

MPI_Recv(&noClient, 1,MPI_INT, MPI_ANY_SOURCE,1, MPI_COMM_WORLD, &ierr);
MPI_Recv(&best, 1, MPI_INT, noClient, 2, MPI_COMM_WORLD, &ierr);

if (best < global_best) global_best = best ;

/* send the next task */
nextTask = nestUseful(nextTask);

MPI_Send(&nextTask, 1, MPI_INT, noClient, 0, MPI_COMM_WORLD);
MPI_Send(&global_best, 1, MPI_INT, noClient, 0, MPI_COMM_WORLD);

  } /* end while */
  finalRecieves();
  MPI_Finalize();
} /* end of server */
else { /* client */
nbTasks = generateTasks(n, max,k); best = initial_limit;

MPI_Recv(&task, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &ierr);

  while ( task < nbTasks ) {
    best = solveTask(task,best);
/* send the result */

MPI_Send(&p, 1, MPI_INT, 0,1, MPI_COMM_WORLD);

MPI_Send(&best, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);

    /* receive next task */
    MPI_Recv(&task, 1,MPI_INT,0 ,0, MPI_COMM_WORLD,&ierr);
    MPI_Recv(&best, 1,MPI_INT,0 ,0, MPI_COMM_WORLD,&ierr);
  } /* finish while */
  MPI_Finalize();
} /* end of client */
```

## Program Codes

The codes using the above mentioned algorithms are as follows -

**Serial Implementation - C**

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>


int isGolomb(int x[],int in){
        int map[1000];
        for(int i=0;i<1000;i++) map[i] = 0;


        for(int i=0;i<in;i++){
                for(int j=i+1;j<in;j++){
                        int diff = x[j]-x[i];
                        if(map[diff]==1) return 0;
                        map[diff]=1;
                }
        }
        return 1;
}


int recurse(int n,int k,int x[],int in){
        int isg = isGolomb(x,in);
        if(!isg) return 0;


        if(in==n && isg){
                printf("%d  %d  ",n,x[n-1]);
                for(int i=0;i<n;i++) printf("%d ",x[i]);
                printf("\n");
                return 1;
```

```
	}

	x[in]=x[in-1]+1;
	while(x[in]<=k-n+in){
		int res = recurse(n,k,x,in+1);
		if(res==1) return 1;
		x[in]++;
	}
	return 0;
}

int main(){
	int n;
	scanf("%d",&n);
	int x[n];

	int k = (n*(n-1))/2;
	int k2 = (int)((double)(n*n)-2*n*pow(n,0.5)+pow(n,0.5)-2);
	if(k2>k) k = k2;

	x[0] = 0;

	while(1){
		x[n-1] = k;
		if(recurse(n,k,x,1)) break;
		k++;
	}

	return 0;
}
```

**Parallel Implementation - CUDA**

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__device__ int isGolomb(int *x, int in) {
	int map[100];
	for (int i = 0; i<100; i++) map[i] = 0;

	for (int i = 0; i<in; i++) {
		for (int j = i + 1; j<in; j++) {
```

```
                    int diff = x[j] - x[i];
                    if (map[diff] == 1) return 0;
                    map[diff] = 1;
                }
        }
        return 1;
}


__device__ int recurse(int n, int k, int *x,int *p, int in) {
        int isg = isGolomb(x, in);
        if (!isg) return 0;


        if (in == n-1) {
                if (isGolomb(x, n) && (x[n-1]<p[n-1] || p[n-1]==0)) { for (int i = 0; i < n; i++)
p[i] = x[i];return 1; }
                else if(isGolomb(x,n)) return 1;
                else return 0;
        }


        x[in] = x[in - 1] + 1;
        while (x[in] <= k - n + in) {
                int res = recurse(n, k, x,p,in + 1);
                if (res == 1) return 1;
                x[in]++;
        }


        return 0;
}


//algo 5.3
__global__ void kernel(int *n, int *k,int *p, int *res) {
        int i1 = blockIdx.x + 1;
        int i2 = threadIdx.x + 1;

        int x[15];
        x[0] = 0;
        x[1] = i1;x[2] = i2;x[*n - 1] = *k;


        if (*n>3) {
                if (i2 <= i1 || i2 >= *k) return;
                int r = recurse(*n, *k, x, p,3);
                if (r == 1) res[0] = 1;
```

```
	}
	else if (*n == 3) {
		if (i2 <= i1 || i2 >= *k) return;
		x[1] = i1;
		x[2] = i2;
		if (!isGolomb(x,3)) return;
		else {
			if (x[*n - 1] < p[*n - 1] || p[*n - 1] == 0) for(int i = 0; i < *n; i++) p[i] =
x[i];
			res[0] = 1;
		}
	}
	else if (*n == 2) {
		if (!isGolomb(x,2)) return;
		else {
			for (int i = 0; i < *n; i++) p[i] = x[i];
			res[0] = 1;
		}
	}
	else{
		res[0] = 1;
	}
}


int main() {
	int n, r = 0;
	int p[100];
	scanf("%d", &n);
	int *d_res,*d_p,*d_n,*d_k;
	cudaMalloc((void **)&d_res, sizeof(int));
	cudaMalloc((void **)&d_n, sizeof(int));
	cudaMalloc((void **)&d_k, sizeof(int));
	cudaMalloc((void **)&d_p, sizeof(int)*n);

	int k = (n*(n - 1)) / 2;
	int k2 = (int)((double)(n*n) - 2 * n*pow(n, 0.5) + pow(n, 0.5) - 2);
	if (k2>k) k = k2;


	cudaMemcpy(d_n, &n, sizeof(int), cudaMemcpyHostToDevice);
	while(1) {
		cudaMemcpy(d_k, &k, sizeof(int), cudaMemcpyHostToDevice);
		kernel<<<k-n+3,25>>>(d_n,d_k,d_p, d_res);
		cudaDeviceSynchronize();
		cudaMemcpy(&r, d_res, sizeof(int), cudaMemcpyDeviceToHost);
		cudaMemcpy(p, d_p, sizeof(int)*n, cudaMemcpyDeviceToHost);
```

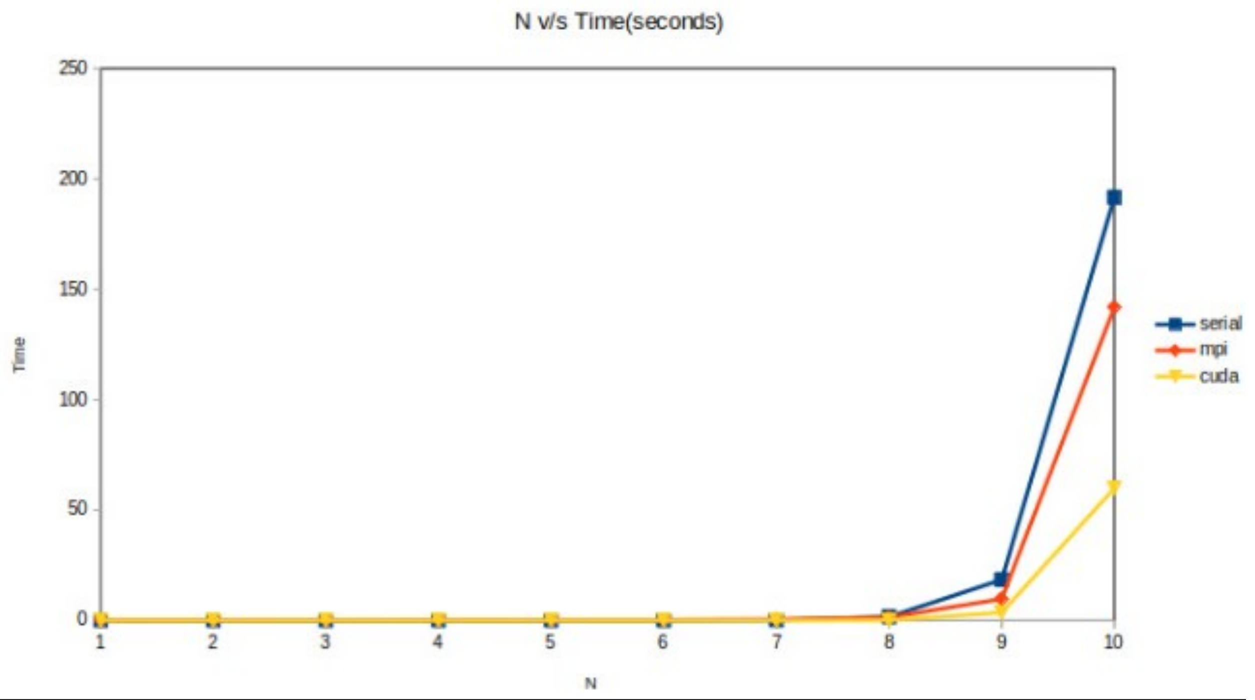
```
                cudaDeviceSynchronize();
                if (r==1) {
                        printf("%d  %d  ", n, p[n-1]);
                        for(int i = 0;i < n;i++) printf("%d ", p[i]);
                        printf("\n");
                        break;
                }else k++;
        }


        return 0;
}
```

**Parallel Implementation - MPI**

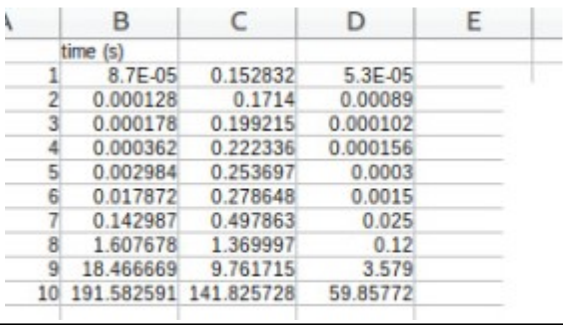
```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "mpi.h"


int isGolomb(int x[],int in){
        int map[100];
        for(int i=0;i<100;i++) map[i] = 0;


        for(int i=0;i<in;i++){
                for(int j=i+1;j<in;j++){
                        int diff = x[j]-x[i];
                        if(map[diff]==1) return 0;
                        map[diff]=1;
                }
        }
        return 1;
}


int recurse(int n,int k,int x[],int in){
        int isg = isGolomb(x,in);
        if(!isg) return 0;


        if(in==n && isg){
                return 1;
        }


        x[in]=x[in-1]+1;
        while(x[in]<=k-n+in){
                int res = recurse(n,k,x,in+1);
                if(res==1) return 1;
                x[in]++;
        }
        return 0;
}


int main(int argc, char *argv[]){
        int s,r;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&s);
        MPI_Comm_rank(MPI_COMM_WORLD,&r);

        int n,res=1000,tn=1000;
        int x[30];

        if(r==0){
                scanf("%d",&n);
        }


        MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);


        int k = (n*(n-1))/2;
        int k2 = (int)((double)(n*n)-2*n*pow(n,0.5)+pow(n,0.5)-2);
        if(k2>k) k = k2;


        x[0] = 0;


        if(r>=k){
                x[n-1] = r;
                if(recurse(n,r,x,1)){
                        res = r;
                }
        }


        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Reduce(&res,&tn,1,MPI_INT,MPI_MIN,0,MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        if(r==0){
                if(n==1) printf("\n%d\n",tn);
                else printf("\n%d\n",tn-1);
        }
        MPI_Finalize();
        return 0;
}
```

## Results

It is inferred that for smaller inputs upto **n = 5, the execution time is fairly small** and is **almost the same** (is of the same order) for all the three different implementations. It is also seen that the time taken by **MPI is a little more** for smaller values, because of the time involved in sending, receiving and transferring data among processes. However, for **larger values of n (n>5) , the time increases exponentially for serial** implementation, whereas remains **polynomial for the parallel** implementations. This is because the parallel code runs different permutations and checks for their validity as Optimal Golomb rulers concurrently and parallel to each other, instead of sequentially checking the next permutation only if the first one proved to be invalid. The smallest ruler is obtained in the parallel code by giving preference to smaller value even if a ruler has already been found. So, for larger input values, especially for **n > 9, the time reduces** by a large margin using CUDA. For n = 10 and greater, the serial code takes over 10 minutes to generate the output, wherever the CUDA program can do so within a tenth of this time (approx 1 minute).



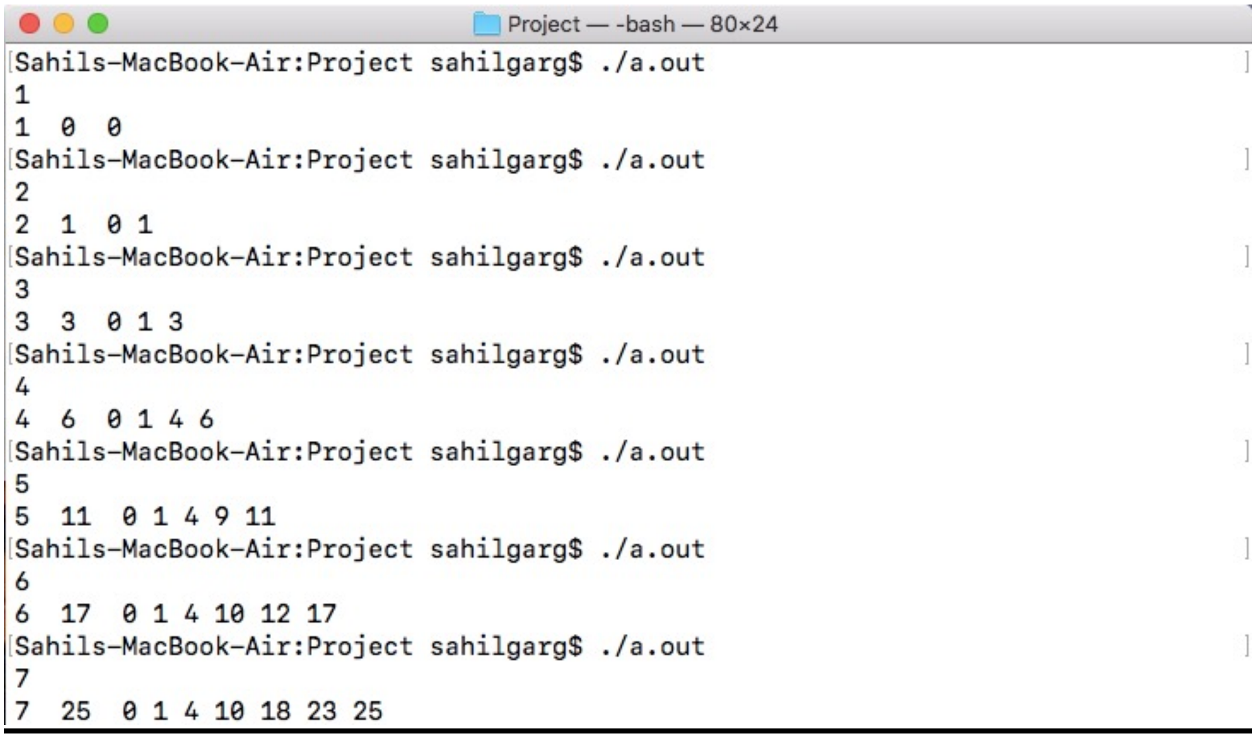
**Graph showing the difference in execution time (in seconds) serially/ using MPI and Cuda to generate Optimal Golomb rulers**



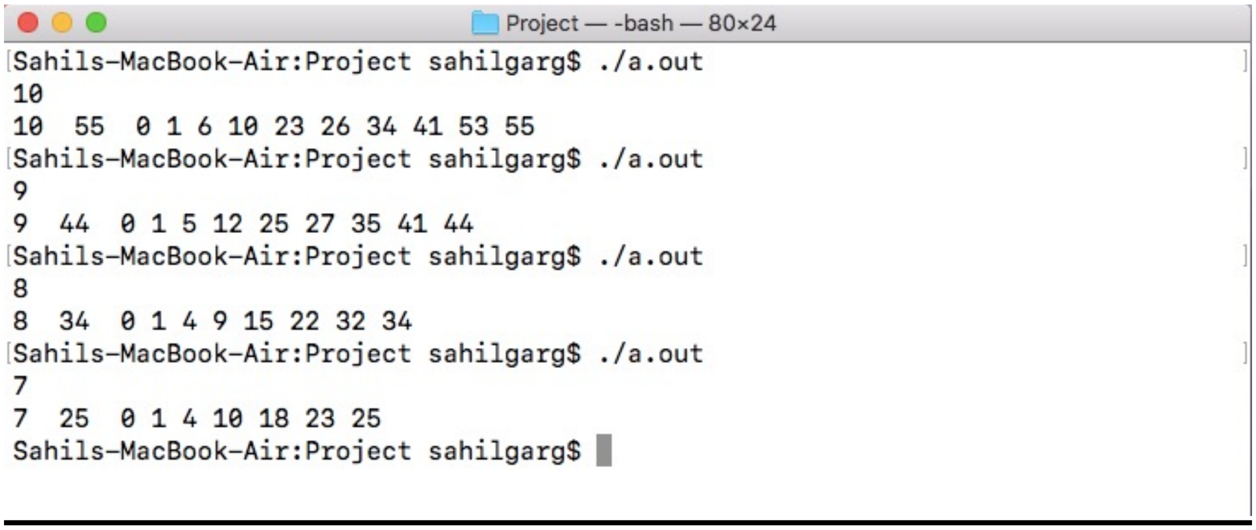
| A | B | C | D | E |
|---|---|---|---|---|
| | time (s) | | | |
| 1 | 8.7E-05 | 0.152832 | 5.3E-05 | |
| 2 | 0.000128 | 0.1714 | 0.00089 | |
| 3 | 0.000178 | 0.199215 | 0.000102 | |
| 4 | 0.000362 | 0.222336 | 0.000156 | |
| 5 | 0.002984 | 0.253697 | 0.0003 | |
| 6 | 0.017872 | 0.278648 | 0.0015 | |
| 7 | 0.142987 | 0.497863 | 0.025 | |
| 8 | 1.607678 | 1.369997 | 0.12 | |
| 9 | 18.466669 | 9.761715 | 3.579 | |
| 10 | 191.582591 | 141.825728 | 59.85772 | |

**Table showing execution time in seconds for serial code, MPI code and CUDA respectively**

## Sample Input / Output



```
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
1
1  0  0
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
2
2  1  0 1
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
3
3  3  0 1 3
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
4
4  6  0 1 4 6
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
5
5  11  0 1 4 9 11
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
6
6  17  0 1 4 10 12 17
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
7
7  25  0 1 4 10 18 23 25
```



```
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
10
10  55  0 1 6 10 23 26 34 41 53 55
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
9
9  44  0 1 5 12 25 27 35 41 44
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
8
8  34  0 1 4 9 15 22 32 34
Sahils-MacBook-Air:Project sahilgarg$ ./a.out
7
7  25  0 1 4 10 18 23 25
Sahils-MacBook-Air:Project sahilgarg$ 
```

## Limitations and Possible Improvements

- The algorithm runs till only n=10 because of single GPU, Golomb's ruler for larger values of n are computed using cluster computers so we were unable to implement the function for larger values of n.
- Also, we are currently using global memory in kernel and each kernel defines an array to store the ruler which can be optimized by using shared memory in the future releases of the code.
- For efficient use of the GPU under heavy computation, since we require to check a large number of permutations for the optimal ruler, we must run the parallelized form of the algorithm on clusters, and take advantage of the shared memory available for quick and efficient memory access.

## Conclusion

Over the course of this project we have successfully managed to develop and evaluate a method for solving OGR-n by utilizing multiple computational nodes in parallel.
The heart of this project is to understand a generic approach for the parallel resolution of combinatorial optimization problems and come up with an approach using GPU parallelisation for a specific problem, which is the Golomb rulers. We applied this theoretical approach to implement the solution in parallel using two standards - MPI and CUDA, and showed that the efficiencies are good in practice. The applications have been written in C and parallelized using OpenMP or MPI. The key advantage of OpenMP (MPI) is to offer a user-friendly tool to parallelize the sequential algorithm: it enables the programmer to develop a parallel application from the sequential one with minimum changes. Computational nodes can be any kind of processing cores, such as CPU cores distributed over asset of hosts, or GPU cores residing within an Nvidia CUDA device. Our parallelization method consists of a pair of algorithms that run recursiely to allow for solving OGR-n for any n value upto an upper limit, by utilizing multiple computational nodes in parallel in CUDA language. Our algorithm for finding Golomb rulers within the boundaries of a given search space piece is simple and easy to understand. Although there are a few limitations, we have provided possible improvements along with the graphs for linear versus parallel execution of the Optimal Golomb's ruler for further study and implementation.

## References

- [1] Jaillet, Christophe & Krajecki, Michael. (2019). Constructing Optimal Golomb Rulers in Parallel.

- [2] Mountakis, Kiriakos Simon. (2000). PARALLEL SEARCH FOR OPTIMAL GOLOMB RULERS.

- [3] T. Rankin, William. (1995). Optimal Golomb Rulers: An Exhaustive Parallel Search Implementation. pp. 13-21.

- [4] Krajecki, Michael & Jaillet, Christophe & Bui, Alain. (2008). Parallel Tree Search for Combinatorial Problems: a Comparative Study between OpenMP and MPI, .. Stud. Inform. Univ.. 6. 274-312. pp. 17-22.

- [5] Soliday, Stephen & Homaifar, Abdollah & Lebby, Gary. (2000). Genetic Algorithm Approach To The Search For Golomb Rulers.

- [6] J. Colannino, "Modular and Regular Golomb Rulers", *Cgm.cs.mcgill.ca*, 2003. [Online]. Available: http://cgm.cs.mcgill.ca/~athens/cs507/Projects/2003/Justin-Colannino/. [Accessed: 01- Apr- 2019].