PARALLEL SEARCH FOR OPTIMAL GOLOMB RULERS

Kiriakos Simon Mountakis
(k.s.mountakis@student.tudelft.nl)

Department of Electronic & Computer Engineering,
Technical University of Crete

ABSTRACT

In this thesis we introduce an exhaustive backtracking search algorithm and it's parallelization, for solving the OGR-n problem: The search for (at least one member of) the set of Optimal Golomb rulers (OGRs) with n marks. The problem under discussion is a combinatorial optimization 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 the already known embarrassingly parallel nature of the OGR-n problem, our parallelization additionally allows for arbitrary selection of the amount of work assigned to each computational node. An experimental evaluation of the practical value of our approach is performed on the massively parallel Nvidia CUDA platform, as well as on the Grid system residing in the Technical University of Crete.

Revised, January 2010.

CHAPTER 1

INTRODUCTION

In this thesis we introduce an exhaustive backtracking search [1][2] algorithm and it's parallelization, for solving the OGR-n problem: The search for (at least one member of) the set of Optimal Golomb rulers (OGRs) with n marks. The problem under discussion is a combinatorial optimization 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 where $n > 24$. Besides exposing the already known embarrassingly parallel nature of the OGR-n problem, our parallelization additionally allows for arbitrary selection of the amount of work assigned to each computational node. An experimental evaluation of the practical value of our approach is performed on the massively parallel Nvidia CUDA platform, as well as on the Grid system residing in the Technical University of Crete.

In mathematics, a Golomb ruler (named after Solomon Golomb) is a set of integers starting from zero ($x_1 = 0 < x_2 < x_3 \cdots < x_n$), selected such that all their cross differences $x_{ij} : i \neq j$ are distinct. In other words, a Golomb ruler can be defined as a set of marks along an imaginary ruler, where two pairs of marks are the same distance apart. Golomb rulers have practical applications in scientific fields such as Radio Astronomy, Information Theory and VLSI. 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.

Both exact and approximation algorithms [3][4][5][1] have been designed for the OGR-n problem. That is, algorithms that yield Golomb rulers with the shortest possible length and algorithms that yield Golomb rulers with a length close to the shortest possible, respectively. Approximate methods satisfy our needs for the practical use of Golomb rulers, as they can produce short enough Golomb rulers for large n values within a small amount of time. On the other hand, exact methods mostly satisfy our curiosity and the need to overcome barriers, such as the overwhelming complexity of discovering the shortest possible Golomb ruler that can be formed with only as few as 25 or 30 marks. The method presented in this thesis is of course a (parallel) exact solution to the OGR-n problem.

In Chapter 2-GOLOMB RULERS, we introduce and provide an overview of Golomb rulers and their properties.

Chapter 3-RELATED WORK, provides a brief description of well known algorithms for solving the OGR-n problem as well as a brief description of two research projects which are closely related to our work: The *Golomb Engine (GE)* project of the Technical University of Crete led by professor Apostolos Dollas (aiming at solving the OGR-n problem on FPGA devices using a parallel version of the Shift algorithm) and the *OGR* project of the distributed.net[1] organization which is responsible for the discovery of all OGRs starting with 20 marks, up to 26 marks so far (by orchestrating a massive volunteer-based distributed computation effort, utilizing their highly efficient *FLEGE* algorithm).

In Chapter 4-BACKTRACKING SEARCH, we cover the design of our sequential OGR-n algorithm in a sequence of steps. We start with the bruteforce search, which is the most naive solution algorithm. We proceed with contrasting it and replacing it with backtracking search. We then introduce further optimizations to backtracking search, such as the midpoint preclusion search space reduction technique and with representing the current state of the search process using a pair of bitvectors. In each following step, we provide an experimental confirmation of induced performance benefits, based on our own C language implementation of the algorithm that each step describes.

[1] http://distributed.net/ogr

In Chapter 5-SEARCH SPACE PARTITIONING, we cover the most important aspect of our work, the parallelization of our backtracking search algorithm. We start by closely investigating the search space of our algorithm. We then proceed on developing an algorithm, that allows us to partition the search space of an instance of the OGR-n problem, into an arbitrary number of pieces, each of arbitrary size. Most importantly, the benefit of our parallelization method over already existing methods (such as the one used by distributed.net or the GE project), is that each produced piece can be arbitrarily large. That is, each piece can correspond to an arbitrarily large amount of work.

Following, Chapter 6-PARALLEL AND DISTRIBUTED COMPUTING, introduces the practices of parallel computing and distributed computing. An effort is made to provide a brief overview of each, explain what is the difference between them, and describe which types of problems each is more suitable for. Finally, we introduce the concept of SpeedUp for measuring induced performance benefits of parallel and distributed computation platforms and explain how the maximum anticipated SpeedUp is bound from above according to Amdahl's law.

Chapter 7-NVIDIA CUDA, introduces the massively parallel NVIDIA CUDA computation platform[2], which is currently the most prevalent solution for utilizing the Graphics Processing Unit (GPU) as a highly parallel co-processor to the CPU. In this chapter, we give an overview of the NVIDIA CUDA architecture, describe it's programming model, and explain best practices for avoiding common bottlenecks in performance.

In Chapter 8-PARALLEL SEARCH WITH CUDA, we cover our utilization of the NVIDIA CUDA platform, for evaluating our parallelized OGR-n algorithm, in terms of gained SpeedUp with respect to running it's sequencial version on a single CPU core. We describe how we chose to implement concepts introduced in chapters 4 and 5 on the NVIDIA CUDA platform in accordance with the best practices described in chapter 6 and present resulting running times for solving various instances of the OGR-n problem.

In Chapter 9-PARALLEL SEARCH WITH THE T.U.C. GRID, we cover our utilization of the Grid computation system residing in the Technical University of Crete[3], for evaluating our parallelized OGR-n algorithm on a distributed computation platform, again with respect to gained SpeedUp with respect to running it's sequencial version on a single CPU core. We start with briefly introducing the concept of Grid systems. We then describe the design and implementation of the algorithm executed by each node on the Grid for orchestrating the parallel solution of the OGR-n problem using our algorithm and present resulting running times for solving various instances of the OGR-n problem.

In Chapter 10-FUTURE WORK, we briefly describe a set of ideas which can possibly help in further enhancing our work described here. We describe how the value of the lowest possible length of an OGR with n marks (defined as $L(n)$ in Chapter 4) can be substantially increased (and thus be brought closer to the actual length), by use of the Linear Programming [6] formulation described in [7] Additionally, we introduce our own Dynamic Programming [1] OGR-n solution algorithm. We discuss possible disadvantages of such an approach and we further investigate how our parallelization of the search space (described in Chapter 4) can be used in conjuction with this algorithm and what are the expected performance benefits of this scenario.

[2] http://www.nividia.com/cuda

[3] http://www.grid.tuc.gr/

CHAPTER 2

GOLOMB RULERS

In mathematics, a Golomb Ruler [5][8] with n marks (named after Solomon Golomb), can be defined as a set of n integer values, chosen so that all resulting cross differences (e.g., differences between each pair of values) are distinct. As illustrated in Figure 2.1, for n chosen values, there are

$$0 + 1 + 2 + 3 + \cdots + n - 1 = \sum_{i=0}^{n-1} i = n(n-1)/2 = \binom{n}{2}$$

resulting cross differences between them. The number of marks of a Golomb Ruler define it's *order* . The aforementioned definition of a Golomb Ruler, poses no restriction as to whether the chosen values should be positive or negative. However, as we will be seeing over the following sections, Golomb Rulers of a certain form; the *canonical* form, find actual uses in various scientific fields. According to this form, a Golomb Ruler corresponds to a desk ruler used for measuring distances. Letting the smallest of the set of integer values be equal to zero, then all other (positive) $n-1$ values correspond to marks on a desk ruler. The largest of those $n-1$ values (e.g., the last mark on the desk ruler) defines the ruler's *length* . The resulting desk ruler would then have the property that between each pair of marks, some of the $n(n-1)/2$ distinct cross differences can be measured as a distance. For the remaining of this document, we will be assuming to work with Golomb Rulers in the canonical form.

A Golomb Rulers of a certain order, is said to be *optimal* , if there exists no other ruler of that same order with a smaller length. That is, the last mark on the ruler (e.g., the n-th mark) is as small as possible. For the remaining of this document we will be refering to Optimal Golomb Rulers using the abbreviation OGRs. An example OGR of order 5 can be seen in Figure 2.2 A table with all OGRs discovered so far can be seen in Table 2.1. Furthermore, when all resulting $n(n-1)/2$ distances cover the set $\{1, 2, \ldots, n(n-1)/2\}$, the ruler is said to be *perfect* . In other words, a perfect ruler can be used to measure all the distinct distances up to it's length. It is trivial to see that a perfect ruler of a certain order is always

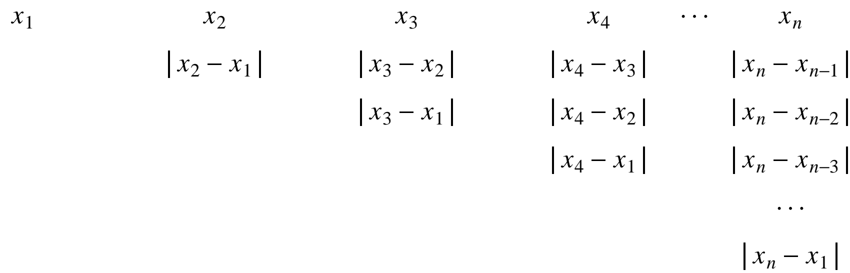

Figure 2.1. Visual representation of a set of n integer values $x_1, x_2, .., x_n$ and the resulting $n(n-1)/2$ cross differences.

Figure 2.2. An Optimal Golomb Ruler (OGR) of order 5 in it's corresponding desk ruler representation (with a length of 11).

n	Optimal Golomb Ruler
1	0
2	0 1
3	0 1 3
4	0 1 4 6
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

Table 2.1. All known OGRs as of December 2010. For $n \geq 23$ all OGRs have been discovered by the massively distributed computation effort orchestrated by the distributed.net *OGR* project.

optimal. The search for an optimal ruler of a certain degree would then be equivalent to finding a perfect ruler of that same degree. However, it has been proven[9] there can be no perfect ruler with more than 5 marks. Thus, perfection for a Golomb Ruler of any order n, is a sufficient condition for optimality, but not necessary.

2.1. Trivial Construction of a Golomb Ruler

Constructing any (not necessarily optimal) Golomb Ruler with n marks is a trivial task. Consider the following procedure for doing so, where deciding for the value of one mark at a time we can make sure there are no produced conflicting (e.g., equal) differences:

We set the first mark to zero. For each next i-th mark we set, we introduce i new differences. The smallest of those newly introduced differences, is the one between the i-th and the $i-1$-th mark, (e.g., $x_i - x_{i-1}$). If for every mark x_i, we make sure this difference is bigger than the previously introduced biggest difference (e.g., $x_{i-1} - x_1 = x_{i-1}$), the resulting ruler will be a Golomb Ruler. A recursive formula for deciding the values of marks x_i of a Golomb Ruler produced in this trivial manner, can be expressed as:

$$x_i = \begin{cases} 0 & i = 1 \\ x_{i-1} + x_{i-1} + 1 = 2x_{i-1} + 1 & i > 1 \end{cases}$$

This formula however, besides contributing to our insight relative to the problem of constructing a Golomb Ruler, it does not guarantee optimality.

2.2. The Function $G(n)$

The function $G(n)$ denotes the length of any optimal Golomb Ruler with n marks (e.g., of order n). Fact is, we do not know much about this function and the only way to determine it's value for a certain n, is to actually find any of the optimal Golomb Rulers with n marks. According to the list of OGRs found so far (mostly attributed to the work of distributed.net), we can see a plot of $G(n)$, up to $n = 26$ in Figure 2.3.

As observed in [10] by Dimitromanolakis, $G(n)$ seems to be bounded above by n^2. At the same time, we can show that $G(n)$ is also bounded from below by $n(n-1)/2$:

Lemma 2.1. $G(n) \geq n(n-1)/2$

Proof. Between the marks of a Golomb Ruler with n marks, there are exactly $n(n-1)/2$ distinct positive integer differences. The biggest of those differences equals $G(n) = |x_n - x_1|$. If we assume $G(n) < n(n-1)/2$, then according to the pigeonhole principle[11], some differences have to appear more than once. This, in turn, contradicts with our initial assumption that all differences are distinct. □

2.3. The Scientific American Algorithm

The first of the algorithms for constructing Optimal Golomb rulers was presented in a Scientific American article in December 1985[8]. Besides the number of marks n, this algorithm is also given as input an upper bound to the ruler's length. This second input parameter is supposed to express the caller's belief that the Optimal Golomb ruler to be found will certainly not be longer than that.

This algorithm consists of a procedure called "exhaust" which methodically generates candidate rulers and of a second procedure called "checker" which checks if any of those rulers are Golomb. Whenever a Golomb ruler is found, it gets output and the search goes on in hope that another shorter ruler will be

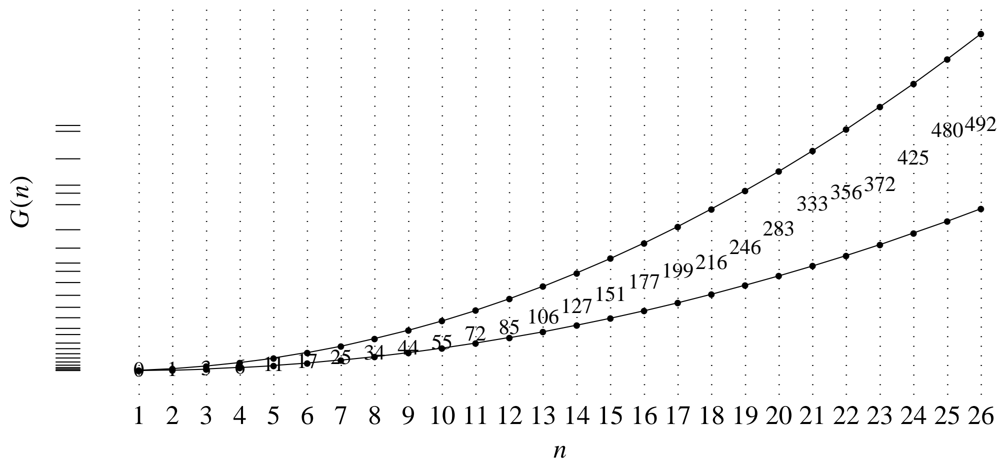

Figure 2.3. Plot of the function $G(n) = x_n - x_1$ according to the list of OGRs found as of 2010. Note how $G(n)$ appears to be bounded from above by n^2 and bounded from below by $n(n-1)/2$.

formed. That is, on input n and K, the exhaust procedure exhausts the search space of all rulers defined by n marks and a length of at most K.

The scientific american algorithm essentially performs a backtracking search over all members of said search space. It tries to construct a Golomb ruler by placing each next mark to the shortest possible distance from it's previous adjacent mark, so that no conflicting cross differences are introduced by this new mark.

2.4. The Token Passing Algorithm

This algorithm [12] was designed by professor Apostolos Dollas at the Duke University. As with the Scientific American algorithm, this algorithm also consists of a procedure for generating candidate rulers and of a procedure for checking if any candidate is a Golomb ruler.

In contrast to the Scientific American algorithm however, this algorithm does not try to construct a Golomb ruler incrementally by appending one mark at a time at a suitable position. Rather, it tries all possible configurations of all n marks in a brute-force manner, until one configuration is found that forms a Golomb ruler. The advantage of this algorithm is that it does not require an upper bound value for the length. The methodical way of searching through configurations guarantees that the first Golomb ruler to be formed will be also be Optimal.

2.5. The Shift Algorithm

This algorithm has been designed by professor Apostolos Dolas and his students at the Duke University. In essence, this algorithm is an enhanced version of the Scientific American algorithm with respect to the "checker" procedure. One enhancement is the fact that when the next, l-th mark is appended, only the l new cross differences it introduces have to be checked against all preexisting cross differences, instead of recomputing and checking all $l(l-1)/2$ cross differences. A second enhancement is the fact that the currently constructed segment of the ruler is represented by means of a pair of bit vectors which allows for checking if a candidate ruler is Golomb in a very efficient way.

CHAPTER 3

RELATED WORK

Our work is directly related to other scientific efforts aimed in contributing to the efficient construction of Optimal Golomb rulers. Most notable parallel methods for solving OGR-n, include the work of W. Ranking in [12] , the Golomb Engine (GE) project of the Technical University of Crete [13] and most importantly the OGR project of the distributed.net organization.

The purpose of Rankin's work was to design, implement and experimentally evaluate a parallel algorithm for finding Optimal Golomb rulers. In particular, he initially performed an experimental evaluation of the Token Passing algorithm and the Shift Algorithm. Based on his comparison of these two algorithms, he then goes on to design a parallel, fault-tollerant and restartable version of the Shift algorithm. Finally, he then proceeds into implementing his parallel algorithm using the Message Passing Interface (MPI)[14] protocol. Finally, he then runs his implementation on a cluster at Duke University and manages to pioneer the discovery of a 19 marks Optimal Golomb ruler.

Research on constructing Optimal Golomb rulers in parallel has also been conducted under the context of the Golomb Engine (GE) project of the MHL laboratory of Technical University of Crete[1]. The GE project focuses on designing and implementing a parallel algorithm for OGR-n on Field-Programmable Gate Array (FPGA)[15] machines. The latest development in that direction is the GE3 project, conducted by Malakonakis under the context of his undergraduate thesis [15]. In his work, Malakonakis first introduces a hardware (destined for an FPGA) architecture that implements the well-known Shift algorithm for solving OGR-n. He then enhances this architecture by introducing the ability to perform multiple shift operations (see our description of the Shift algorithm) in parallel, in essence, introducing an FPGA design of a parallel version of the Shift algorithm. Finally, he concludes with measuring delivered performance of his approach on certain FPGA models, for solving OGR-n, with up to $n = 15$ marks. In particular, running his parallel version of the Shift algorithm on a Spartan XC3S1000, Malakonakis was able to solve OGR-14 and OGR-15 in about 7 and 36 minutes respectively.

The most fruitful effort regarding discovery of Optimal Golomb rulers so far, comes from the OGR project of distributed.net. The distributed.net organization is internet's first general purpose distributed computing project, founded in 1997, with the following mission statement[2]:

> "Three independent goals: development, deployment & advocacy to be pursued in the advancement of distributed computing."

The OGR subproject of this organizatin is a massively distributed computing project, which is based on the contribution of computational nodes by volunteers for solving the next biggest OGR-n instance. On February 24 - 2009, distributed.net announced the discovery of an Optimal Golomb ruler with 26 marks. Volunteers around the world are able to download a program from the organization's website. This program contacts specific distributed.net servers located all around the world and downloads the next "stub" (essentially a search space piece of the current OGR-n instance to be solved). Upon downloading a stub, this program then processes it with the very efficient Feiri-Levet GARSP Engine (FLEDGE) search algorithm, developed by researchers of distributed.net. After processing a stub, this program answers back to some server whether a Golomb ruler has been possible to form or not. As of February 2008, the task

[1] http://www.mhl.tuc.gr

[2] http://distributed.net

currently undertaken by project OGR is the discovery of a Golomb ruler with 27 marks and is expected to last for about 7 years in total.

CHAPTER 4

BACKTRACKING SEARCH

Our main purpose in this thesis is to solve the OGR-n problem with a parallel algorithm. This chapter covers the design of our sequencial algorithm for OGR-n, which we will subsequently go on to parallelize in the following chapter.

In section 1 we formally state the OGR-n problem and introduce the GR-n,K subproblem and proceed to show how solving OGR-n is essentially accomplished by solving multiple instances of GR-n,K. That is, the core of this chapter is about solving the GR-n,K subproblem efficiently.

In sections 2 and 3 we design an efficient backtracking search algorithm for solving GR-n,K. We start by describing a brute force search algorithm, for better explaining the logic behind solving GR-n,K. We then introduce a backtracking search algorithm with lower complexity than brute force and display it's superiority in performance through experimental measurements. Finally, we further enhance this backtracking search algorithm with a search space reduction technique called midpoint preclusion and with a technique where search state is represented by means of a pair of bitvectors, as is the case with the Shift algorithm explained in chapter 2. Again we present any performance enhancements through experimental measurements.

4.1. The OGR-n and GR-n,K problems

Problem 4.1 (OGR-n) Given positive integer n, find and return an Optimal Golomb ruler with n marks.

Algorithm 4.1 (General OGR-n algorithm)

Given positive integer n:

1 Calculate $L(n)$, some lower bound for the length of a Golomb ruler with n marks

2 **For** each integer $K \geq L(n)$:

3   Solve problem instance GR-n,K. **If** a Golomb ruler has been found :

4   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:

5   return ruler, quit search

6 return failure

In Algorithm 4.1 we try a range of increasing candidate values for $G(n)$ (an Optimal Golomb ruler's length) starting from $L(n)$. $L(n)$ could be any value that is proven to be a lower bound for the length of a Golomb ruler with n marks. Thus, $L(n)$ defines the starting point of the search process and is chosen so that the Golomb ruler to be eventually found is guaranteed to be Optimal.

Note that Algorithm 4.1 will always return a ruler. On the other hand, if a Golomb ruler with n marks and length K does not exist, then Algorithm 4.2 will return failure.

In Algorithm 4.1, for as long as K remains too small, calls to Algorithm 4.2 will return failure. When K gets large enough, we will have the first successful formation of a Golomb ruler by Algorithm 4.2 We will then know that $G(n)$ equals the current K value, as the first Golomb ruler to be formed is also guaranteed to have the smallest length possible (i.e. be Optimal).

4.1.1. The Function $L(n)$

One way to encode a ruler is with a vector of integers x[1] to x[n]. In such an encoding, the distance where the i-th mark is put is equal to x[i]. Since the first mark is always used as the start of the ruler, x[1] is fixed equal to zero. Actually, $L(n)$ is a lower bound for x[n], the distance where the ruler's last mark is put.

With the requirement that the ruler encoded by x[] is Golomb, we can calculate such a lower bound for each mark x[i] based on the following observation: every subsection of a Golomb ruler is also a Golomb ruler. That is, if x[$1 \cdots n$] forms a Golomb ruler, subsections x[$1 \cdots 2$], x[$1 \cdots 3$], and so on, up to x[$1 \cdots n-1$] must also be Golomb rulers. But then, x[2], x[3], up to x[$n-1$] correspond to each ruler's length respectively. Remember that the function $G(n)$ defines the optimal (shortest) length for a Golomb ruler with n marks. We can thus say that if we are looking for a vector x[$1 \cdots n$] which forms a Golomb ruler, we are essentially looking for a vector where x[i] >= $G(i)$ for $1 \le i \le n-1$ (assuming we have knowledge of $G(1)$ up to $G(n-1)$).

One out of many possible ways to calculate $L(n)$ (a lower bound for $x[n]$), is by using the aforementioned lower bounds for x[1] up to x[$n-1$], and a couple of other observations, all summarized bellow:

(1) Since each mark is placed further from the previous one, we know that x[$i+1$] >= x[i]+1. As a result, x[n] >= $G(n-1)+1$.

(2) According to the work of dimitromanolakis concerning the relation between Golomb rulers and Sidon sets [10], another valid lower bound for the length of any Golomb ruler with n marks gives x[n] $\ge n^2 - 2n\sqrt{n} + \sqrt{n} - 2$.

(3) In addition to those last two lower bounds, a third valid lower bound resulting from what we have showed in chapter 2 is: x[n] $\ge G(n) \ge n(n-1)/2$. Taking into account that there can be no perfect Golomb ruler with $n \ge 5$ (e.g. $n(n-1)/2$ cross differences cannot cover the set of integers $1, \cdots, n(n-1)/2$), we can increment this lower bound by one, for rulers with at least 5 marks:

$$\mathrm{x}[n] \ge \begin{cases} \dfrac{n(n-1)}{2} & n \le 4 \\ \dfrac{n(n-1)}{2} + 1 & n \ge 5 \end{cases}$$

Note that none of those lower bounds is necessarily the highest. As a result (for rulers with at least 5 marks), derive that the lowest possible length, or in other words, the lowest distance the last mark can be set at, resutls by taking into account the combination of the three aforementioned lower bounds:

$$\mathrm{x}[n] \ge L(n) = \max\left(G(n-1)+1, n(n-1)/2+1, n^2 - 2n\sqrt{n} + \sqrt{n} - 2\right) \qquad \text{(Eq. 4.1)}$$

Summarizing, for all n marks of a Golomb ruler (where $n \ge 5$), we can bound the distance each can be set at from below according to the following expression:

$$\mathrm{x}[i] \ge \begin{cases} G(i) & 1 \le i \le n-1 \\ L(n) & i = n \end{cases}$$

4.2. Brute Force Search for GR-n, K

Now that we have showed how to calculate the minimum values for the distance each mark can be set to, let us proceed with the description of an inefficient, brute force search algorithm for GR-n, K.

Consider the Program 4.1 implementation of Algorithm 4.2, which searches for a Golomb ruler by systematically visiting each member of the set of n-mark rulers with a given length K.

On the left we can see the imperative version while on the right we can see the recursive version. The imperative version is of no practical value since the number of loops is hardcoded in accordance to parameter n. However, it's purpose is to clarify what the recursive version (depicted on the right) actually does.

To search for a Golomb ruler in the set of n-mark rulers with a certain length K, we just have to call the recursive program with arguments (n, K). It will then perform a search procedure equivalent to the imperative algorithm.

In Figure 4.1 we can see a tree layout of the recursive calls performed when calling BRUTEFORCE$(5, 11)$. That is, when searching for a Golomb ruler between rulers with 5 marks and with a length of 11. The lowest possible distances marks 2,3 and 4 can be set to are $G(2) = 1, G(3) = 4$ and $G(4) = 7$ respectively. In particular, we can see an animation [16] of the execution in rows of frames, depicting snapshots of the recursion tree whenever we check if $[x_1, \cdots, x_n]$ is a Golomb ruler. Each frame depicts two alternative views of the same situation. The upper view depicts at which node in the recursion tree we are at the moment. The lower view depicts which ruler we are checking at this point, to see if it is Golomb, with the *RULER* row showing where the marks of the ruler have been put and the *DIFFS* row showing, beneath each mark, it's distances to the marks that have been set before it.

4.3. Backtracking Search for GR-n, K

Enhancing the simple bruteforce algorithm, a more efficient approach would be to perform a backtracking search. That is, try to construct a member ruler of this space by placing one mark at a time, having

```
program BRUTEFORCE_IMPERATIVE(K)
1  -- First and last marks are fixed.
2  x[1] := 0
3  x[n] := K

4  -- Try all possible configurations for
5  -- remaining n − 2 marks.
6  for x[n − 1] from G[n − 1] to K − 1
7   for x[n − 2] from G[n − 2] to x[n − 1]-1
8    for x[n − 3] from G[n − 3] to x[n − 2]-1
9      ...
10       ...
11        for x[2] from G[2] to x[3]-1
12         if x[1, · · · , n] is Golomb
13          -- All marks placed successfully.
14          -- Output found Golomb ruler
```

```
program BRUTEFORCE(m, K)
1  -- Set m-th mark at distance K
2  x[m] := K

3  -- If managed to set first mark.
4  if m = 1
5   if x[1, · · · , n] is Golomb
6     -- All marks placed successfully.
7     -- Output found Golomb ruler

8  -- Try possible positions for next mark.
9  for K′ from G[m − 1] to x[m]-1
10  BRUTEFORCE(m − 1, K′)
```

Program 4.1. Brute force search implementation of Algorithm 4.2. Imperative version on the left, recursive on the right.

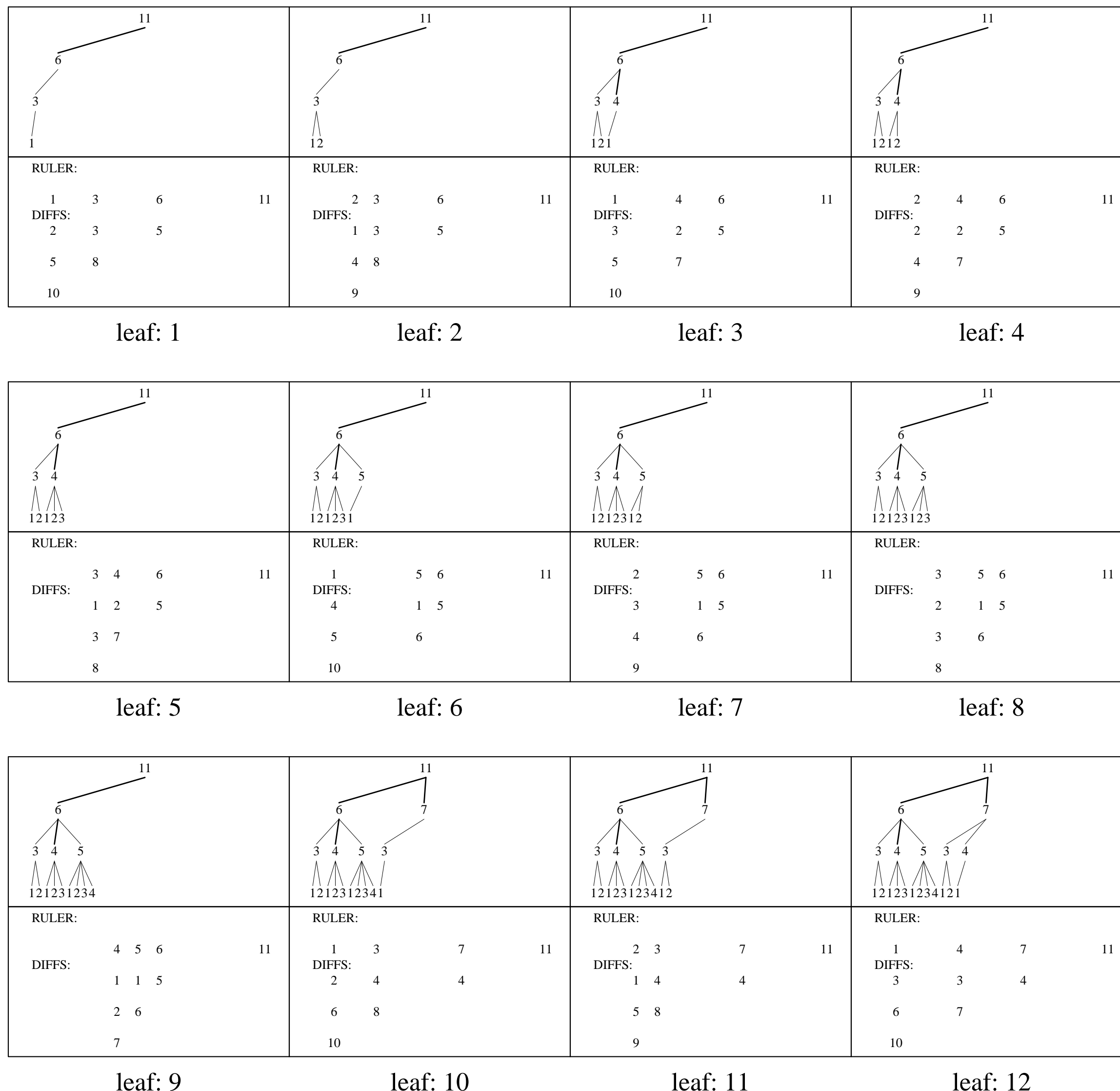

Figure 4.1. Step-wise animation of the first few steps of BRUTEFORCE where $n = 5$.

in mind we want it to be a Golomb ruler. Consider the backtracking search program depicted on Program 4.2, for finding a Golomb ruler in the space of n-mark rulers with a given length K.

Looking at the imperative program, it might appear that we now check whether some ruler is Golomb, many times more. In reality though, this check is actually performed many times less, because some trees in the search space get prunned and their branches never get checked Figure 4.2.

For example, at the third frame of the animation depicted in Figure 4.2, we can see that as soon as the second mark is put at distance 3, it introduces it's cross difference of 3 to the third mark which is put at distance 6. This cross difference equals the value of the second marks, which is the same with it's cross difference to the first mark which represents the start of the ruler. As a result, there is no meaning in using configuration $[3, 6, 11]$ furthermore. Subtree $(2, 3)$ gets prunned and after backtracking we move on to subtree $(2, 4)$ at the next frame, and so on.

```
program BACKTRACK_IMPERATIVE(K)
 1  -- Set first and last mark.
 2  x[1] := 0
 3  x[n] := K

 4  -- Try all possible configurations for remaining
 5  -- n − 2 marks, backtracking when necessary.
 6  for x[n − 1] from G[n − 1] to K − 1
 7    if x[n − 1, n] is Golomb
 8      for x[n − 2] from G[n − 2] to x[n − 1]-1
 9        if x[n − 2, n − 1, n] is Golomb
10          ...
11            ...
12              for x[2] from G[2] to x[3]-1
13                if x[1, · · · , n] is Golomb
14                  -- All marks placed successfully.
15                  -- Output found Golomb ruler.
```

```
program BACKTRACK(m, K)
 1  -- Set m-th mark at distance K.
 2  x[m] := K

 3  if x[m, m + 1, · · · , n] is Golomb
 4      if m = 2
 5        -- All marks placed successfully.
 6        -- Output found Golomb ruler.
 7  else
 8    -- Backtrack to previous mark.
 9    return

10  -- Setting the m-th mark at K is good.
11  -- Move on to the next mark.
12  for K′ from G[m − 1] to x[m]-1
13    BACKTRACK(m − 1, K′)
```

Program 4.2. Backtracking search implementation of Algorithm 4.2, Imperative version on the left, recursive on the right.

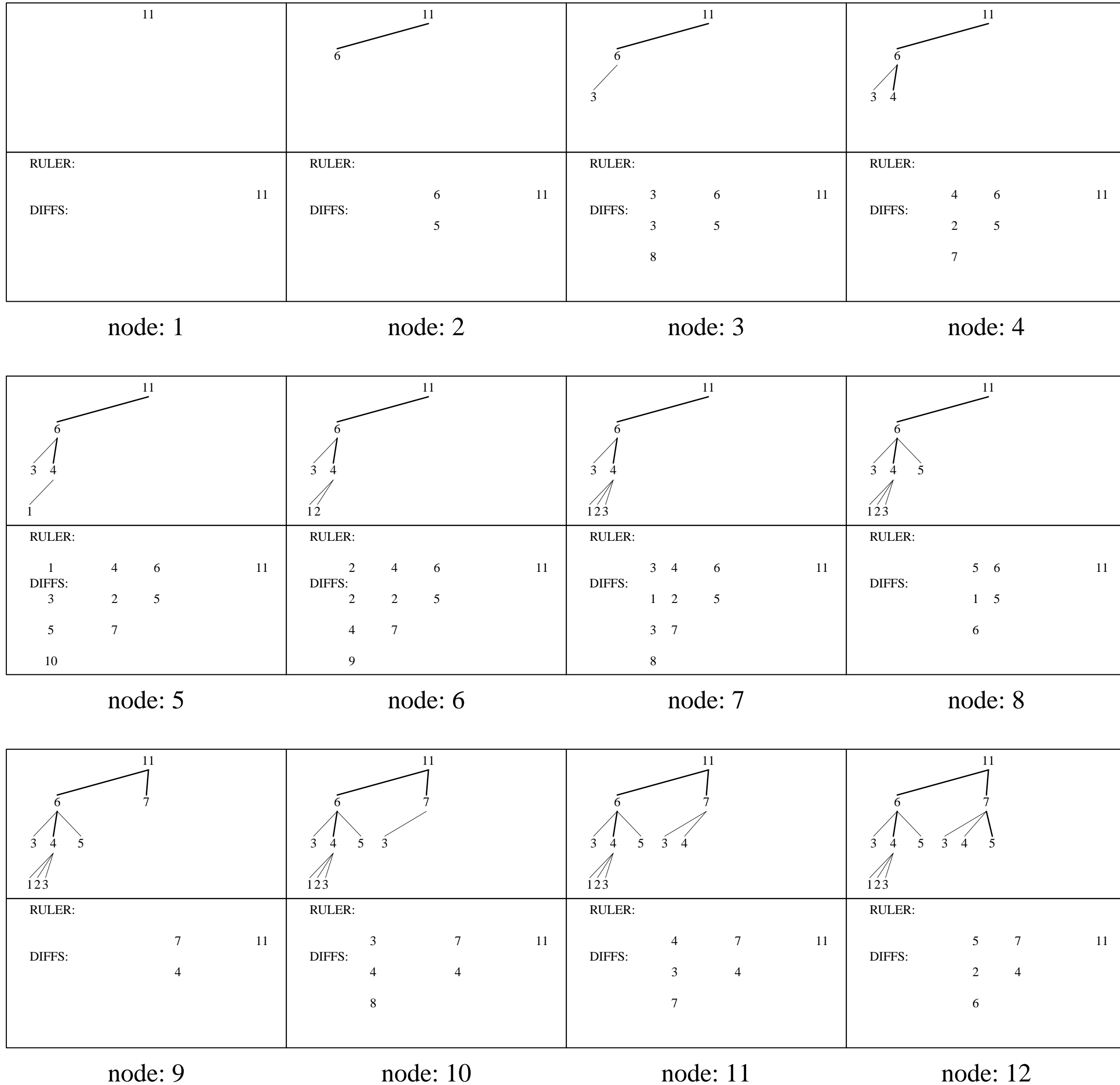

Figure 4.2. Step-wise animation of the first few steps of BACKTRACKING when $n = 5$ (see also Figure 4.1).

This very prunning of the search space is what makes backtracking more efficient than bruteforce search. Moreover, we no longer check complete n-mark rulers for being Golomb. When we arrive at a node in the search tree, we know that up to this point, the marks that have been set form a Golomb ruler. Thus, each time we only need to check if the mark we just set upon arriving at this node, introduces distances which already exist. A mark introduces $O(n)$ new distances, and each time there are $O(n^2)$ already existing distances. The complexity of comparing the distances introduced by a mark we just set, to the distances that already existed, is thus $O(n^3)$. On the other hand, in bruteforce search, upon having formed a complete ruler (along a tree branch), we perform this check for each one of it's n-marks, thus with a complexity of $O(n^4)$. That is, the complexity of performing this check upon arriving at each node of a branch, is the same with performing an overall check upon having formed a complete branch.

In Figure 4.3 and Table 4.1 we can see an experimental confirmation, of the anticipated superiority in performance, of backtracking over bruteforce searching. Measurements are based on our own C implementation of the two searching methods, for solving OGR-n instances over a range of increasing n values.

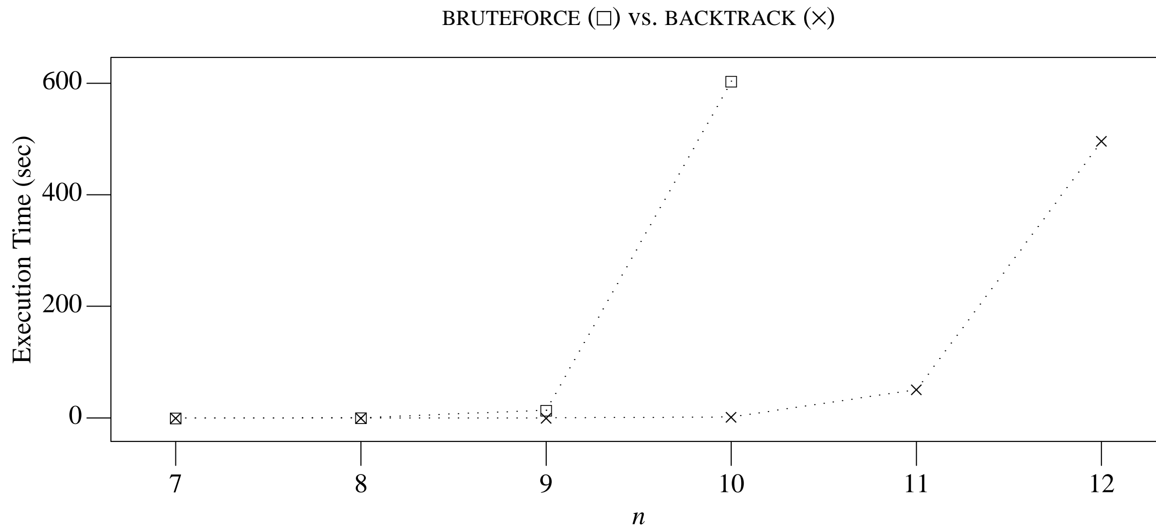

Figure 4.3. Execution times of brute force versus backtracking for finding a Golomb ruler over increasing n values.

	n			
	7	8	9	10
BRUTEFORCE	0.008	0.373	13.992	604.000
BACKTRACK	0.002	0.023	0.208	1.780

Table 4.1. Values of graph on Figure 4.3.

4.3.1. Search Space Reduction

One basic property of Golomb rulers, is that each Golomb ruler has a corresponding mirror image. For constructing the mirror image of some Golomb ruler, suppose we place a mirror at the first mark on the left end. The projection of the ruler on the mirror corresponds to the ruler's mirror image which is a Golomb ruler too, since it's marks measure the same set of cross differences. In Figure 4.4 we can see an example of how to construct the mirror image of a given Golomb ruler, where we use one of the Optimal Golomb rulers found in the last search example with $n = 5$ marks. It can be proven, that except for the case of the $[0,1]$ Golomb ruler, the mirror image is always a different ruler than the original. Since each Golomb ruler has one mirror image, for a given number of marks n, there is always an even number of Optimal Golomb rulers that can be constructed.

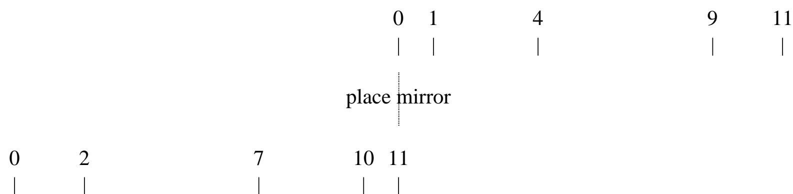

Figure 4.4. Each Golomb ruler has a mirror image. Schematic representation of the Golomb ruler $[0,1,4,9,11]$ and the construction of it's mirror image $[0,2,7,10,11]$. Both rulers measure the same set of cross differences.

However, since given a Golomb ruler, the construction of the mirror image is a trivial task, we might as well skip searching for the mirror images. Following, we present two techniques for skipping mirror images in our search, thus reducing the search space.

First Mark Preclusion

According to the mirror images we described earlier, if a Golomb ruler starts with $[0, 1 \cdots]$, that is, the second mark measures distance one from the beginning, then it's mirror image cannot have it's second mark at that same position. If in our search we never consider rulers which have their second mark set to one, we are guaranteed to skip either the original or the mirror image of Golomb rulers which indeed have (in their original or mirror image) their second mark at position one.

Midpoint Preclusion

This technique is also based on the concept of mirror images and we have actually used it to reduce the search space of our own algorithm. It is based on the concept of the `geometric center' and the `middle mark' of a ruler. The geometric center of a ruler with a certain length $x_n = K$ is at distance $K/2$ and the middle mark of a ruler with n marks is mark $n/2$.

Now consider a ruler with a certain length K and it's mirror image. If the middle mark of the original ruler is placed after it's geometric center, then it must be before the geometric center for the mirror image and vice versa. We can thus limit in our search, the middle mark to be either before or after the geometric center. This essentially cuts the search space in half.

However, when the desired number of marks n is even, the middle mark might as well be exactly at the geometric center, for both the original and the mirror image of Golomb rulers in the search space. In that case, both the original and the mirror image of a Golomb ruler might be found. As a result, midpoint preclusion is not anticipated to be as effective a search space reduction measure for n even, as it is for n odd.

Furthermore, note that midpoint preclusion cannot be used in conjuction with first mark preclusion. Each method is guaranteed to exclude from the search either the mirror or the original image of certain Golomb rulers in the search space. However, when used in conjuction, it might be the case the the mirror image is excluded by one of the methods, while at the same time the original image is excluded as well, by the other method.

Enhancing Program 4.2 so that it applies the Midpoint Preclusion reduction technique, we will call it BACKTRACKING^{MP} and in Figure 4.5 we can see the performance increase over BACKTRACKING.

4.3.2. Bit Vector Representation

So far we have been using integer vectors to represent the current state of the backtracking search, e.g. the constructed segment of the candidate Golomb ruler and the set of resulting cross differences. We will now consider an alternative, more efficient representation, utilizing bit vectors instead of integer vectors.

The concept of a bit vector essentially maps to the concept of computer registers of arbitrary length, that is, of an arbitrary number of bits. With the exception of FPGAs where we can build (and perform bitwise operations on) an actual register of arbitrary length, in computer architectures where the word size is constant (e.g. 32 bits for IA32), a bit vector can be implemented as an array of words. That is, in a system programming language such as C, a bit vector data type with the ability to represent K bits, can be represented as an array $m = K/32$ of integers. All bitwise operations such as *AND,OR,XOR* that can be performed between registers, can also be performed, pair-wise, across the integer elements which comprise the bit vector. The state of the search process at each point in time is defined by $\mathbf{X}$, a vector which holds the golomb ruler segment constructed so far and by $\mathbf{D}$ which holds the set of cross differences between the marks of that golomb ruler segment.

As an alternative to vector $\mathbf{X}$, we can use a bit vector *list* to maintain the golomb ruler segment constructed so far in the search. we start with all bits in *list* turned off. whenever we set the next mark i to a distance x_i, we turn bit number x_i of *list* on.

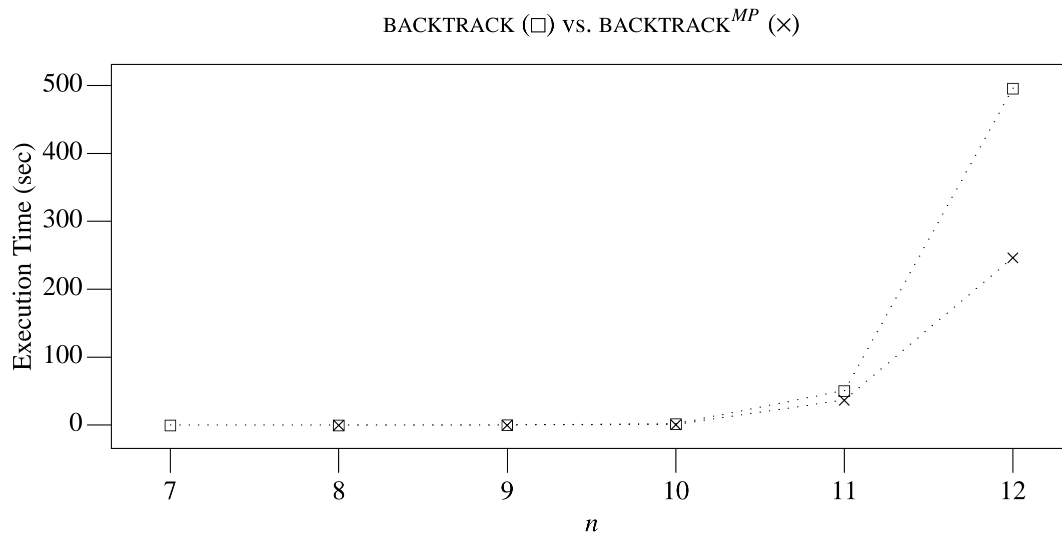

Figure 4.5. Applying Midpoint Preclusion roughly halves the search space (and correspondingly the required execution time).

As an alternative to **D**, we can use a bit vector *dist* to maintain the set of cross distances between the marks of the golomb ruler segment constructed so far in the search. Again, for each element d_i of **D**, we turn the d_i-th bit of *dist* on. Note that having set the last mark to some distance K, both the *list* and *dist* vectors will need to be exactly K bits long.

In each step of the backtracking search algorithm, we check if we can set some next mark i at distance x_i without introducing conflicting cross distances to the marks that have already been set. In particular, we compare the $n-i$ cross differences it introduces, to the cross differences that exist between the marks that have already been set. This enhanced version of our backtracking program is given in Program 4.3.

In Figure 4.6 we can see an animation of this procedure, this time representing the marks we have already set and their cross differences, using the *list* and *dist* vectors correspondingly. Upon setting mark i at distance x_i, shifting left the *list* bit vector by x_i, we reveal it's cross differences to the marks before it. If the result of an *AND* operation between the shifted *list* and the current *dist* is not zero, we have introduced cross distances that already existed in *dist*, thus we have to backtrack.

In Figure 4.7 and Table 4.2 we can see how performance of the backtracking search increases when utilizing the bit vector representation. Over the following sections, the term backtracking search will be refering to the version of backtracking search which utilizes the midpoint preclusion technique as well as the bit vector representation.

```
program BACKTRACK_{MP,BV}(m, K)
 1  -- Set m-th mark at distance K (set K-th bit).
 2  x[m] := K
 3  list[K] := 1

 4  -- Check cross distances of m-th mark to rest of marks against dist[]
 5  if (list[K + 1, ···] ∧ dist[] != 0) or (dist[K] != 0)
 6      -- m bits set in list[] form a Golomb ruler so far.
 7      if m = 2
 8        -- All marks placed successfully.
 9        -- Output found Golomb ruler.
10  else
11    -- Backtrack to previous mark (unset K-th bit).
12    list[K] := 0
13    return

14  -- Setting the m-th mark at K is good.
15  -- Record cross distances of m-th mark in dist[]
16  dist[] := dist[] ∨ list[K + 1, ···]
17  -- Move on to the next mark.
18  for K′ from G[m − 1] to x[m]-1
19    BACKTRACK_{MP,BV}(m − 1, K′)
```

Program 4.3. Enhanced and final version of our backtracking search implementation Algorithm 4.2, BACKTRACK$_{MP,BV}$. Search state at each instant is encoded using the pair of bit vectors *list* and *dist*, so that checking if the currently formed ruler segment is Golomb only takes a bitwise and ($\wedge$) operation between *list[]* and *dist[]*.

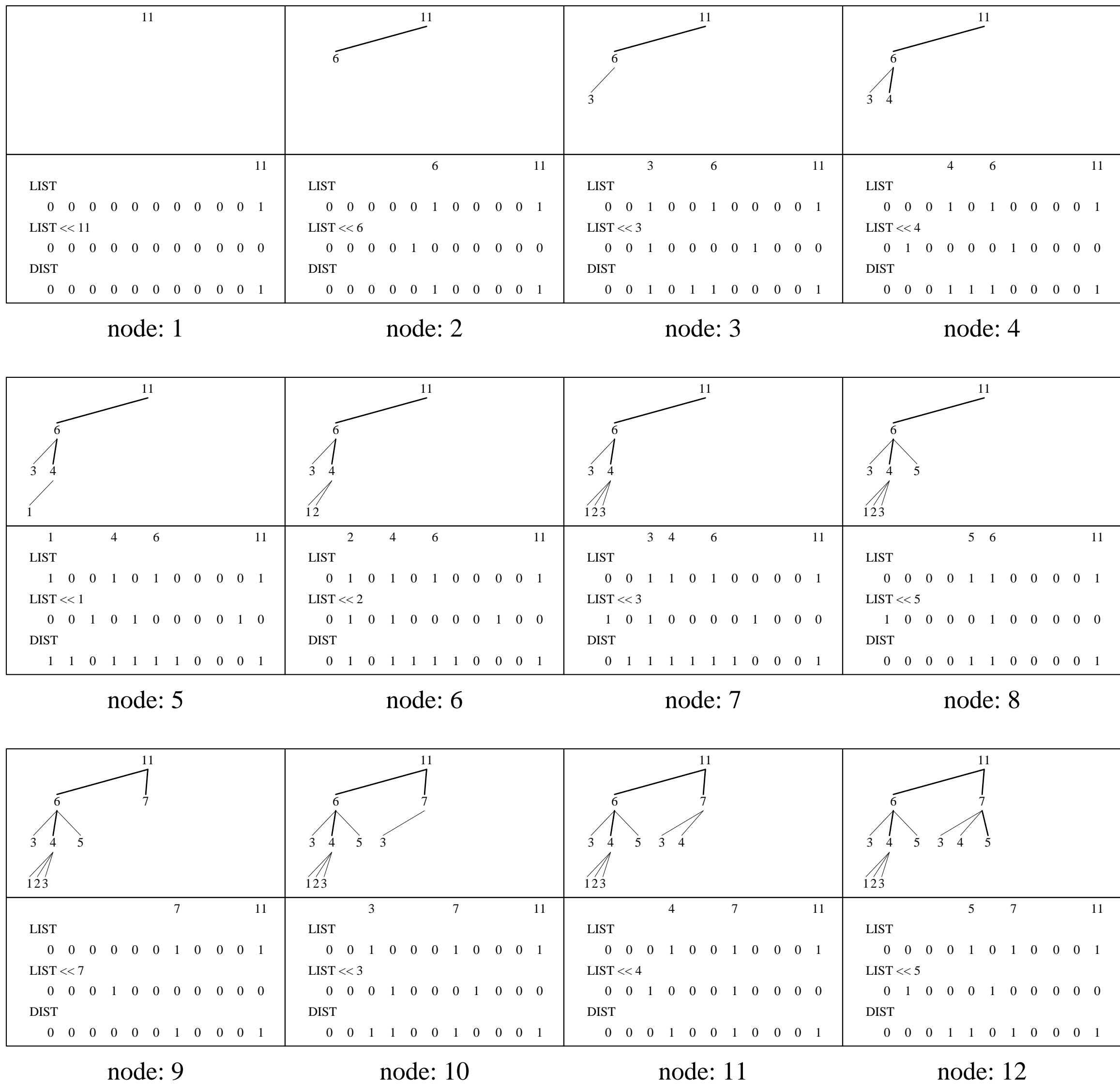

Figure 4.6. Step-wise animation of the backtrack search (for $n = 5$), using the *LIST* and *DIST* bit vectors.

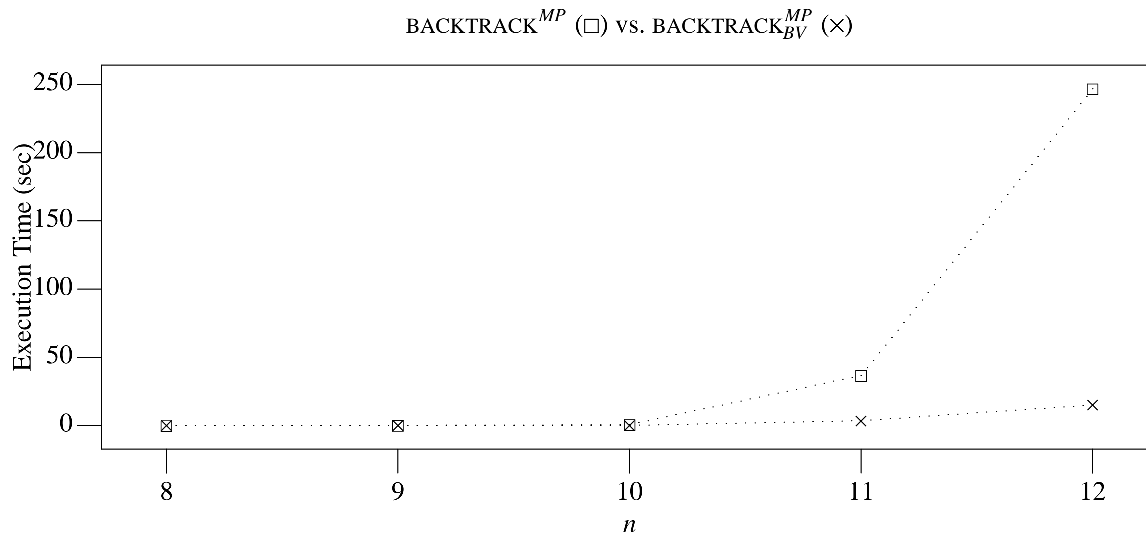

Figure 4.7. Utilizing the bit vector representation for performing the backtracking search, induces a considerable increase in performance.

	n				
	8	9	10	11	12
BACKTRACK^{MP}	0.007	0.142	0.694	36.800	246.815
BACKTRACK$^{MP}_{BV}$	0.004	0.021	0.141	3.560	15.130

Table 4.2. Values of graph on Figure 4.7.

CHAPTER 5

SEARCH SPACE PARTITIONING

This chapter covers the core aspect of our work: Partitioning the search space of each GR-n, K problem instance into an arbitrary number of pieces, that in turn contain an arbitrary number of candidate Golomb rulers. This allows for the parallel solution of GR-n, K (Problem 4.2) and by net-effect, of OGR-n (Problem 4.1), on an arbitrary number P of computational nodes.

We start with describing our search space partitioning method implemented by algorithm *MakePiece* in sections 1 and 2. We continue with describing a parallel version of Algorithm 4.1 and Algorithm 4.2 in section 3, accomplishing our main goal in this thesis which is to design a parallel algorithm for the OGR-n problem. The remaining chapters of this thesis cover the experimental evaluation of the parallel algorithm developed in this chapter.

5.1. Partitioning the GR-n, K Search Space

As we have already seen, the search space of a GR-n, K problem instance can be represented as a tree (Figure 4.1). Each branch corresponds to a ruler, candidate for being Golomb. Alternatively, we can view the search space as the sequence of (candidate) rulers resulting from a Depth-First-Search traversal of the tree (Figure 5.1). Partitioning the search space into pieces is equivalent to partitioning this sequence of rulers into pieces.

Consider we have a set of computational nodes and we feed them with search space pieces, so that the search is being performed in parallel. Those nodes might not be able to record snapshots of their progress every once in a while (e.g., CUDA threads).

The process of finding an Optimal Golomb ruler when n is bigger than say 22, can even take years to complete. This means that every once in a while we should be saving snapshots of the search progress. In case of failure, we would then be able to resume the search from the last snapshot. For this reason, the pieces being fed to the nodes should be small enough, so that progress can be recorded every time a node

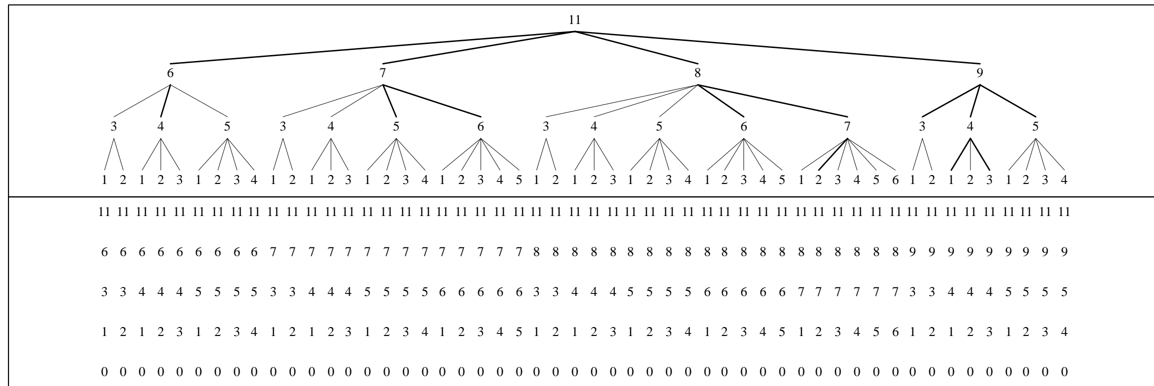

Figure 5.1. Representation of (a portion of) the search space of instance GR-5, 11 as a tree (upper) and as a sequence of rulers (lower).

finishes. The size of the pieces should be controllable, so that we could have an estimate of how much time is required for each node to finish.

That is, having P computational nodes, we should not feed them the whole search space in P large pieces. The search space should be distributed in many parts with P small pieces each.

Another reason why we should not partition the whole search space in P large pieces, is that if the number of available computational nodes P, increased, we would not be able to feed the new nodes. We should for example, feed each node that finishes before the others, with a new piece, insteading of leaving it idle (e.g., GRID nodes).

Let us now focus our attention back on partitioning the search space, e.g., the sequence of candidate (for being Golomb) rulers, with n marks and some given length K.

Consider an algorithm $MakePiece(start, size)\colon N^n \rightarrow N^n$, that takes as argument some ruler $start$ and an integer $size$. It returns the ruler end which is $size$ positions after $start$, in the sequence of rulers we defined before.

We know that the sequence of n-mark rulers with length K starts with ruler $[0, G(1), G(2), \cdots, G(n-1), K]$ and thus we can start the partitioning from this ruler. We produce consecutive pieces of equal size (as seen on Figure 5.2), where the start of the next piece is right after the end of the previous piece. Each piece we produce, we can for example append it to the part of pieces we are preparing for the computational nodes, or feed it to some node that just finished processing some other piece. We can detect if the search space cannot provide any more pieces, by checking if the end of the last piece (e.g., the last end ruler) equals the last ruler of the sequence, which is $[0, K-(n-1), K-(n-2), \cdots, K-1, K]$.

When the search space gets exhausted, we can for example go on to the next search space of n-mark rulers with length $K+1$.

5.2. The $MakePiece(start, size)$ Algorithm

As we have seen so far, we can use this algorithm to create consecutive pieces of the search space, of an arbitrary size.

Recapitulating, we saw that the search space of candidate Golomb rulers with n-marks and length K, is defined by a pair of values (n, K). We saw that this space can be represented as a tree where each branch forms a candidate Golomb ruler, or alternatively, as a sequence of candidate Golomb rulers, as a result of a Depth First Search traversal of the tree (Figure 4.1). For partitioning it into pieces, it is convenient to view the search space as a sequence of candidate Golomb rulers (Figure 5.1).

The algorithm $MakePiece(start, size)$, takes as argument a ruler start and an integer size and returns the ruler that is located size positions later in the search space. This way, we can define a piece of the search space that starts at ruler $start$, ends at the returned ruler (say end), and contains $size$ consecutive rulers in between.

Let us now observe the progression of the rulers from the left towards the right (e.g., towards the end of the search space), in the example in Figure 5.1. We can say that each ruler is a number of a numbering system where the lower marks are the least significant digits and the upper marks are the most significant digits. That is, between two rulers, starting from the upper towards the lower marks, the one that has the first bigger mark, is definitely further towards the end of the search space.

Consider the function $TreeSize(m, v)$[1] which returns the size (e.g., number of branches), of a subtree (m, v) of the search space.

Subtree (m, v) starts with ruler (e.g., it's leftmost branch is) $[0, G(2), \cdots, G(m-1), v]$ and it ends at (e.g., it's rightmost branch is) ruler $[0, v-(m-1), v-(m-2), \cdots, v-1, v]$. Essentially, the function

[1]This function's implementation is not given here. An implementation of this function is based on the theory behind the backtracking search algorithm's worst case complexity as given in Appendix A.

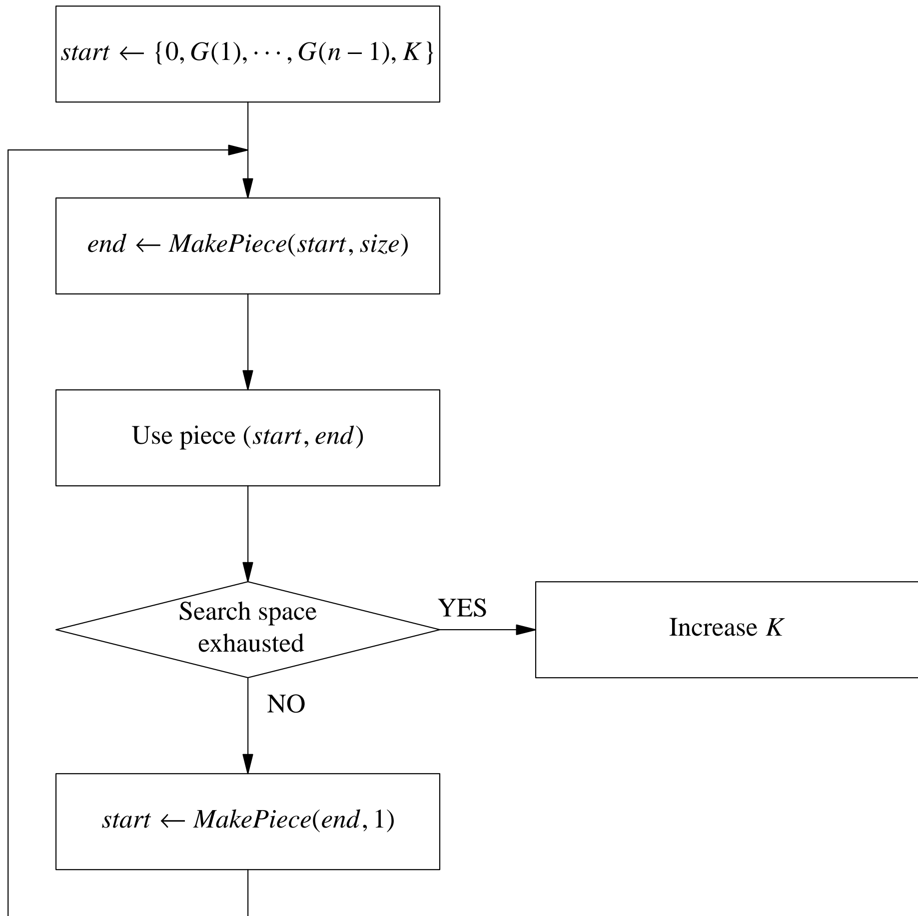

Figure 5.2. The partitioning of the search space in pieces of equal size, using the $MakePiece(start, size)$ algorithm.

$TreeSize(m, v)$ evaluates the nested summation of Eq. 12.1.

Starting from some ruler start$[1, \cdots, n]$, we want to see how to transform it into a ruler end$[1, \cdots, n]$, that is located $size$ positions later in the search space.

We start off changing the lower marks and move towards upper marks, keeping track of the distance coverered so far as we move towards the end of the search space. Below we will see how the algorithm $MakePiece(start, size)$ works, following one by one the steps of an example based on the search space depicted in Figure 5.1.

The first mark always equals zero and it never changes. Assume we begin with ruler $start = [0, 2, 4, 7, 11]$ and want to cover a distance of $size = 27$.

We start with increasing the second mark:

```
11    [11]
7     [7 ]
4     [4 ]
2 ->  [3 ]
0     [0 ]
```

So far we have covered a total distance of $TreeSize(2,3) = 1$.

Next, we cannot increase the second mark anymore. We have reached ruler $[0,3,4]$ which is the end of subtree $(3,4)$. Thus, we move on to increasing the third mark from 4 to 5 and enter subtree $(3,5)$, that extends from $[0,1,5]$ to $[0,4,5]$:

```
11      [11]     [11 11]
7       [7 ]     [7  7 ]
4       [4 ] -> [5  5 ]
2  ->   [3 ]     [1  4 ]
0       [0 ]     [0  0 ]
```

We have covered a distance of $TreeSize(2,3) + TreeSize(3,5) = 1 + 4 = 5$.

Then, we increase the third mark from 5 to 6 and enter subtree $(3,6)$:

```
11      [11]     [11 11]     [11 11]
7       [7 ]     [7  7 ]     [7  7 ]
4       [4 ] -> [5  5 ] -> [6  6 ]
2  ->   [3 ]     [1  4 ]     [1  5 ]
0       [0 ]     [0  0 ]     [0  0 ]
```

We have covered a distance of $TreeSize(2,3) + TreeSize(3,5) + TreeSize(3,6) = 1 + 4 + 5 = 10$.

Next, we cannot increase the third mark anymore. We have reached ruler $[0,5,6,7]$ which is the end of subtree $(4,7)$. Thus, we move on to increasing the dourth mark from 7 to 8, and enter subtree $(4,8)$, that extends from $[0,1,3,8]$ to $[0,6,7,8]$:

```
11      [11]     [11 11]     [11 11]     [11 11]
7       [7 ]     [7  7 ]     [7  7 ] -> [8  8 ]
4       [4 ] -> [5  5 ] -> [6  6 ]     [3  7 ]
2  ->   [3 ]     [1  4 ]     [1  5 ]     [1  6 ]
0       [0 ]     [0  0 ]     [0  0 ]     [0  0 ]
```

We have covered a distance of $TreeSize(2,3) + TreeSize(3,5) + TreeSize(3,6) + TreeSize(4,8) = 1 + 4 + 5 + 20 = 30$. This is more than the distance we wanted to cover, $size = 27$. That is, ruler $[0,6,7,8,11]$ is $30 - 27 = 3$ positions later than the ruler we are looking for. Now that we have surpassed the distance we wanted to cover, we can go one step back where we had covered a total distance of 10, and restart the whole process recursively, aiming to cover the distance $27 - 10 = 17$, starting from right after where we reached.

At the previous step, we had reached ruler $[0,5,6,7]$ that is the end of subtree $(4,7)$. We start right after that, at the start of subtree $(4,8)$, e.g, $start = [0,1,3,8]$, with increasing the second mark:

```
11     [11]
8      [8 ]
3      [3 ]
1  -> [2 ]
0      [0 ]
```

We have covered distance $TreeSize(2,2) = 1$.

Next, we cannot increase the second mark anymore, as we have reached ruler $[0,2,3]$ which is the end of subtree $(3,3)$. Thus, we move on to increasing the third mark from 3 to 4 and enter subtree $(3,4)$, that extends from $[0,1,4]$ to $[0,3,4]$:

```
11     [11]     [11 11]
8      [8 ]     [8  8 ]
3      [3 ] -> [4  4 ]
1  -> [2 ]     [1  3 ]
0      [0 ]     [0  0 ]
```

We have covered distance $TreeSize(2,2) + TreeSize(3,4) = 1 + 3 = 4$. Next, we increase the third mark

from 4 to 5 and enter subtree $(3, 5)$:

```
11    [11]     [11 11]     [11 11]
8     [8 ]     [8  8 ]     [8  8 ]
3     [3 ] ->  [4  4 ] ->  [5  5 ]
1  -> [2 ]     [1  3 ]     [1  4 ]
0     [0 ]     [0  0 ]     [0  0 ]
```

We have covered distance $TreeSize(2, 2) + TreeSize(3, 4) + TreeSize(3, 5) = 1 + 3 + 4 = 8$. At this point, we repeat that our target is to cover distance $size = 17$. The remaining steps will not be described in detail, however they will be reported below:

We enter subtree $(3, 6)$:

```
11    [11]     [11 11]     [11 11]     [11 11]
8     [8 ]     [8  8 ]     [8  8 ]     [8  8 ]
3     [3 ] ->  [4  4 ] ->  [5  5 ] ->  [6  6 ]
1  -> [2 ]     [1  3 ]     [1  4 ]     [1  5 ]
0     [0 ]     [0  0 ]     [0  0 ]     [0  0 ]
```

We have covered a total distance of

$$TreeSize(2, 2) + TreeSize(3, 4) + TreeSize(3, 5) + TreeSize(3, 6) = 1 + 3 + 4 + 5 = 13$$

We enter subtree $(3, 7)$:

```
11    [11]     [11 11]     [11 11]     [11 11]     [11 11]
8     [8 ]     [8  8 ]     [8  8 ]     [8  8 ]     [8  8 ]
3     [3 ] ->  [4  4 ] ->  [5  5 ] ->  [6  6 ] ->  [7  7 ]
1  -> [2 ]     [1  3 ]     [1  4 ]     [1  5 ]     [1  6 ]
0     [0 ]     [0  0 ]     [0  0 ]     [0  0 ]     [0  0 ]
```

We have covered a total distance of

$$\begin{aligned}&TreeSize(2, 2) + TreeSize(3, 4) + TreeSize(3, 5) + TreeSize(3, 6) + TreeSize(3, 7)\\&= 1 + 3 + 4 + 5 + 6 = 19\end{aligned}$$

Again, we have surpassed by 19-17=2 the distance we wanted to cover. Thus, we go back to the previous step again, starting right after ruler $[0, 5, 6, 8, 11]$ that is the end of subtree $(3, 6)$. We restart recursively from the start of subtree $(3, 7)$, e.g., $start = [0, 1, 7, 8, 11]$, wanting to cover a distance of $19 - 17 = 2$. We increase the second mark:

```
11    [11]
8     [8 ]
7     [7 ]
1  -> [2 ]
0     [0 ]
```

We have covered a distance of $TreeSize(2, 2) = 1$. We increase the second mark again:

```
11    [11]    [11]
8     [8 ]    [8 ]
7     [7 ]    [7 ]
1  -> [2 ]->  [3 ]
0     [0 ]    [0 ]
```

We have covered a total distance of $TreeSize(2, 2) + TreeSize(2, 3) = 1 + 1 = 2$, which equals the distance we wanted to cover.

Finaly, we find out that for $start = [0, 2, 4, 7, 11]$, after $size = 27$ places, there is ruler $[0, 3, 7, 8, 11]$, as can be confirmed by visual inspection of Figure 5.1.

The *MakePiece* algorithm is actually fairly simple and is depicted in Algorithm 5.1.

```
algorithm MakePiece(start[1, ... , n], size)
 1  end[1, ···, n] := start[1, ···, n]

 2  for i from 2 to n − 1
 3   -- For increasing values of mark i, enter subtree (i, end[i]).
 4   for end[i] from end[i]+1 to end[i + 1]-1
 5     prev := cur
 6     cur  := cur + TreeSize(i, end[i])
 7     -- Have we surpassed the desired distance?
 8     if cur > size
 9      start[1, ···, n] := [0, G[1], ... , G[i − 1], end[i], start[i + 1], ... ,start[n]]
10          return MakePiece(start[1, ···, n], size − prev)

11     If cur = size
12      return end[1, ···, n]

13  -- Size exceeds the end of the search space, return the ending ruler.
14  return [0, end[n]-(n − 1), end[n]-(n − 2), ... , end(n)]
```

Algorithm 5.1. This algorithm takes as input some ruler $\text{start}[1, \cdots, n]$ and a scalar *size* and returns the ruler $\text{end}[1, \cdots, n]$ which is *size* positions later in the search space.

5.3. A Parallel OGR-n Algorithm

We have in so far described an algorithm *MakePiece* for creating the next piece of the search space of any GR-n, K instance. The boundaries of each piece are defined by a tuple of rulers ($\text{start}[1, \cdots, n]$, $\text{end}[1, \cdots, n]$).

Assuming we have P computational nodes available, consider the following parallel version of Algorithm 4.1 for OGR-n:

Algorithm 5.2 (Parallel OGR-n algorithm)

Given positive integer n:

1 Calculate $L(n)$, some lower bound **for** the length of a Golomb ruler with n marks.
2 Select search space piece size *size*.
3 **For** each integer $K \geq L(n)$:
4   **While** GR-n, K search space has more pieces:
5   Create next P pieces of size *size* using *MakePiece*.
6   **For** all created pieces, do in parallel:
7   Solve problem instance PIECE-GR-n, K on some computational node.
8   **If** at least one ruler has been found,
9   **return** any found ruler, quit search.

Problem 5.1 (PIECE-GR-n, K) Given positive integers n, K, and piece boundaries defined by pair of rulers $\text{start}[1, \cdots, n], \text{end}[1, \cdots, n]$, find and return a Golomb ruler with n marks and length K within piece boundaries, or failure if that is not possible.

Algorithm 5.3 (PIECE-GR-n, K algorithm)

Given positive integers n, K and rulers $\text{start}[1, \cdots, n], \text{end}[1, \cdots, n]$:

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

In Algorithm 5.2, we consume each GR-n, K instance in multiple rounds of P pieces each. All created pieces have the same size, although this is not necessarily what has to be done in general, as the *Make-Piece* algorithm allows for calibrating each piece's size according to circumstances.

In Algorithm 5.3, we do exactly what we did in Algorithm 4.2, except the search is confined between the given piece boundaries. This algorithm can be implemented by the program given in Program 4.3, slightly modified so that the list$[1, \cdots, K]$ vector is kept between the given piece boundaries.

CHAPTER 6

PARALLEL AND DISTRIBUTED COMPUTING

The common characteristic between parallel and distributed computing is that they can both be defined as a form of computing where multiple calculations are carried out simultaneously. This form of computing results from the principle that big problems, can often be divided into smaller ones, which can then be solved in parallel across multiple computational nodes. The computational nodes can range from a single device with multiple processing cores to a network of independent computers.

Even though there are many similarities between the goal of parallel and distributed computing there are also subtle differences between them. These two different terms are often — incorrectly — used synonymously.

The term parallel computing typically refers to the existence of multiple computational nodes within one machine, with all nodes being dedicated to the overall system collectively at each time. The term distributed computing refers to a group of separate machines, each one contributing computational cycles to the overall system, over a network, over time.

The following quote provides a clear distrinction between parallel and distributed computing:

> ``Parallel computing splits an application up into tasks that are executed at the same time, whereas distributed computing splits an application up into tasks that are executed at different locations, using different resources.''[17]

6.1. Problems for Parallel and Distributed Computing

Parallel and distributed comptuing cannot be used to speed up the solution of all kinds of computational problems. Furthermore, problems that are indeed suitable for parallel and distributed computing benefit in varying degrees depending on their type.

Understanding the nature of data dependencies of a problem is fundamental for solving it in parallel. There are problems that consist of a long chain of dependent calculations that have to be performed in a certain order, for example, the problem of calculating the fibbonacci sequence. The solution of such "purely sequencial" problems, when they appear, cannot benefit from parallel and distributed computing.

Most problems consist of serial segments and from segments that can be partitioned into subproblems that can be solved in any order and then get combined to form a single final result.

However, it is usually necessary that computational nodes cooperate by communicating in order to solve the overall problem correctly. Depending on the nature of the problem, it might be necessary that we have frequent communication between the nodes (fine grained parallelism), on the other hand, required communication might be relatively infrequent (coarse grained parallelism).

Note that for the case of distributed computing, communication between computational nodes is much more expensive than for the case of parallel computing, in terms of how much it slows down overall computation. In distributed computing, communication is performed through a slow medium such as a network, while in parallel computing, communication is usually performed through a high speed bus, since by definition, computational nodes reside within the same device.

For this very reason, problems that demand fine grained parallelism are suitable for parallel computing, while problems that demand coarse grained parallelism can benefit from both parallel and distributed computing.

However, another way of distinguishing between types of problems, is between problems that admit a data parallel solution and problems that admit a task parallel solution. Problems that admit data parallelism are tose where some processing is applied over a wide range of datums. The processing of each single datum is done on some corresponding computational node. Problems that admit task parallelism are those where the logical procedure is what actually gets partitioned between computational nodes and there is no required distribution of datums across nodes.

Note that a data parallel problem does not necessarily require communication between computational nodes. For example, consider we just want to apply a xor operation between the pixels of a pair of huge images. This however, does not render the problem suitable for distributed computing, because it is required to transmit a lot of data to the nodes across a slow medium.

On the other hand, a problem that does not require such a distribution of data and also no communication between the nodes, while suitable for distributed computing, is usually not suitable for parallel computing. Usually, highly data parallel problems require the application of some simple, lightweight processing over a wide range of datums. Data parallel problems on the other hand, do not require any distribution of data to computational nodes, but as a tradeoff, require the application of a complex and heavy (in terms of computational resources and execution path variation) process, across computational nodes. The capability of a computational node for the case of parallel computing, is limited in comparison to the sophisticated architecture of a modern processor. Thus, computational nodes in parallel computing devices cannot handle such heavy procedures as well as a distributed computing nodes can. This observation, as we will see in the process, is highly relative to this very thesis.

No interprocess communication problems can be broken further down into two sub categories: parametric and data parallel problems. A parametric problem is an embarrassingly parallel problem that needs to be calculated multiple times with different parameters. For example a physical simulation may require the same calculations applied to the same data with different starting parameters as a problem set. After the multiple calculations have occurred, post processing may be required in order to chose the best fitting result or produce an aggregation of results. A data parallel problem is one where the data are evenly divided between processing nodes before the same algorithm is applied. The difference to plain embarrassingly parallel is that the computation amount may vary depending on which data are allocated. After computation has finished post processing is usually required to combine the result set.

According to what we have defined so far, the way we parallelize the problem of constructing OGRs by partitioning the search space, converts it to a pure embarassingly parallel problem. Communication between processing nodes is not required and other than the start and the end of each piece, there is no need to feed the computational nodes with any substantial amount of data. The processing of each search space piece can be performed in isolation on each node. Furthermore, it is of the parametric type, since all nodes execute the same algorithm, each with different parameters, e.g., each on a different search space piece.

6.2. Types of Parallel and Distributed Computing

Flynn's taxonomy [18] is a classification of computer architectures that is also regularly used to classify algorithms as well Figure 6.1.

Single Instruction, Single Data refers to a architecture that consists of a single processing unit operating on a single stream of data without any parallelism whatsoever. This is analogous to an early personal computer or a classic Von Neumann architecture.

Single Instruction, Multiple Data is an architecture where a single stream of instructions are applied to multiple streams of data simultaneously. Examples would include a processor optimised for array based operations or a graphical processor. The data parallelism model fits SIMD architectures well.

Figure 6.1. Flynn's taxonomy of computer architectures.

Multiple Instruction, Single Data is an architecture where multiple streams of instructions are applied to a single stream of data. Of the taxonomy, this is the rarest seen applied, as only the most fault tolerant designs require separately developed systems operating to provide a consensus result. Examples are found in aerospace applications such as flight control systems.

Multiple Instruction, Multiple Data is an architecture of multiple operating processors working on multiple streams of data. Multi threaded programming is often MIMD and distributed systems, where asynchronous operations by multiple processors on separate data, are a clear example. The task parallelism model fits MIMD capable systems well, such as computing clusters.

6.3. Measuring Performance Gain

In parallel and distributed computing there are two elementary measures called speedup and efficiency that allow the practitioner to compare how the system is working against theoretical ideals. The measures of speedup and efficiency can quickly allow us to determine the scalability of the system or how well it will continue to perform as more nodes and tasks are added.

Speedup is the ration of the overall time needed to solve the problem serially (on one node), over the time needed to solve it in parallel, on P nodes, after having partitioned it.

$$Speedup = \frac{\text{serial execution time}}{\text{parallel execution time}}$$

$$Efficiency = \frac{Speedup}{P}$$

In an ideal case, after having partitioned and distributed the problem across P computational nodes, we would anticipate to have a speedup of P and an efficiency of 1. However, for various reasons, most of the time this is not the case.

This ideal measure is known as linear speedup and is often included on speedup graphs as a baseline for which the actual speedup measurements per number of processing elements is compared. Where speedup is on the y axis and and number of processors on the x axis, linear speedup will draw a straight line at 45 degrees between the axes Figure 6.2: However, for several reasons, most of the time, parallelization

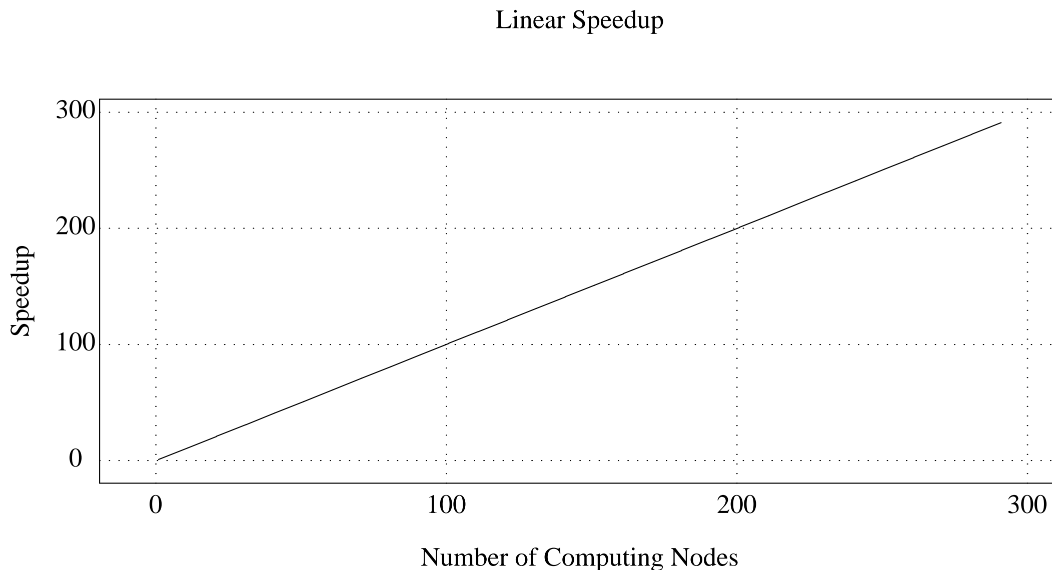

Figure 6.2. Graph presenting the ideal case where the gained speedup increases linearly with the number of utilized computational nodes.

does not yield an ideal, linear speedup. Some of those reasons that limit the actual speedup, we have described above, such as for example, communication between computational nodes through a slow medium, incapability of computational nodes in parallel devices to perform a complicated task efficiently, etc. Besides these issues, the most important cause of actual speedup limitation is that only a certain portion of a program's total time is spent at the part we parallelize. Thus, the serial program part that is left, depending on what portion of the total time is spent at it, limits the maximum achievable speedup value.

Amdahl's Law can be used to predict the maximum possible speedup achievable when a problem is parallelized using in comparison to using only a single serial processor, given that the problem size remains the same when parallelized. It states that if F is the proportion of a program that can be made parallel, then the serial portion can be defined as $(1-F)$. If we then define the total time for the serial computation as 1 for any time unit, we can compute the speedup by dividing the old computation time with the new computation time that consists of the serial portion plus the parallel portion divided by the number of parallel computational nodes, denoted as P. This gives us the following equation:

$$Speedup = \frac{\text{serial execution time}}{\text{parallel execution time}} \leq \frac{1}{(1-F)+F/P}$$

Then we can see that as the number of processors tends to infinity the maximum speedup tends to (1/1-F) and the serial portion becomes dominant. In Figure 6.3 we can see how the maximum anticipated speedup is limited in comparison to the ideal, according to amdahl's law.

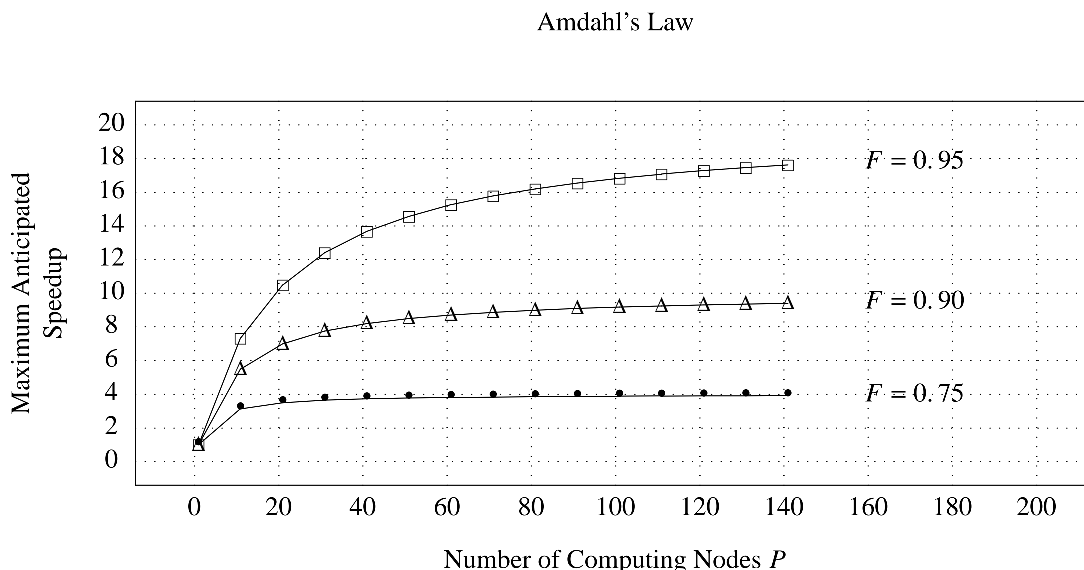

Figure 6.3. According to Amdahl's law, the anticipated speedup of some parallelization of a program, is bounded from above according to the percentage of the time spent in executing the serial portion of the program $(1-F)$.

CHAPTER 7

NVIDIA CUDA

Over the last few years, there is a trend in using GPU devices for solving difficult problems that admit solutions with high parallelism. The NVIDIA CUDA platform [19] [20] is, so far, the most prevalent approach of this kind.

The idea of using computer graphics hardware for general-purpose computation (GPGPU) has been around now for over two decades. However, GPGPU did not really take off until ATI and NVIDIA introduced programmable shading in their commodity GPUs in 2002. This enabled programmers to write short programs that were executed for each vertex and pixel that passed through the rendering pipeline. Researchers were quick to realize that this was not only useful for graphics programming but that this could also be used for general purpose calculations. The processing power of the GPU has been increasing at a much faster pace than of the CPU. This caused a swift increase in research that utilized GPGPU computation. Graphics processing can be parallelized as each vertex or pixel can most often be processed independently of other vertexes or pixels in each step of the graphics pipeline.

As GPU development has mainly been driven by computer games and the quest for better and faster graphics it has caused the GPU to become specialized for intensive, highly parallel SIMD computation and is therefore designed such that more transistors are devoted to data processing rather than data caching and flow control unlike the CPU. Today GPUs have multiple cores driven by a high memory bandwidth, offer massive processing resources, and are especially well-suited to address problems that can be expressed as data-parallel computations.

The utilization of the processing power of the GPU, however, did not come for free. It required researchers to pose their problems as graphics rendering tasks and go through the graphics API, which is very restrictive when it comes to programming. The APIs where used in such a way that textures were used for input and output and fragment shaders (program stubs which run for each pixel projected to the screen) were used for processing.

This meant a high learning curve for programmers not already familiar with the graphics APIs, and the environment was very limiting when it came to debugging. This also greatly narrowed the range of potential problems that could be solved by using the GPU. GPU manufacturers, however, noticed these efforts and have now introduced both software and hardware to greatly simplify the use of GPUs for general purpose computation.

In late 2007 NVIDIA introduced the Compute Unified Device Architecture (CUDA), a parallel-programming model and software environment designed to enable developers to overcome the challenge of developing application software that scales transparently over parallel devices of different capabilities.

The CUDA parallel programming model is independent of it's potential (software and hardware) implementations. A central concept to this model, is the Parallel Thread Execution (PTX) Instruction Set Architecture, which defines a virtual parallel processing machine.

With CUDA, NVIDIA also introduced a new line of graphics processing units that implemented this parallel programming model and with its application these devices were no longer standard GPUs but became massively parallel stream processors which were now programmable with (a slightly augmented version) of standard C. One or more installed CUDA-enabled Devices, are intended to be used in conjuction with the CPU, as coprocessors. The idea is that the CPU is efficient in solving serial problems while the GPU is efficient in solving parallel problems. The CUDA platform, provides the ability to develop applications that can choose to run their serial parts on the CPU and their parallel parts on the GPU. These applications are written in the CUDA C language, essentially a version of standard C, slightly augmented

with functionality relative to using the GPU as a coprocessor device.

7.1. Architecture

Nvidia's implementation of the CUDA parallel computation model, consists of software and hardware components. Hardware components are CUDA-enabled GPUs, starting from the G80 series and ending at the (scientific computation) specialized Tesla series.

The software components are the CUDA driver API, the CUDA runtime API and the CUDA Software Development Kit. The CUDA driver API is implemented as a user-level library and it allows for low level control of installed CUDA devices. This API is not normally used, except for special occassions. What is normally used for the (intended) usage of installed CUDA-enabled GPUs as multicore co-processors to the CPU, is the CUDA runtime API, which allows for starting parallel tasks and transfering data to and from the CUDA device.

The CUDA SDK provides the necessary tools for the development of CUDA applications, such as a specialized compiler, linker and assembler, a specialized debuggger and a specialized profiler.

Besides the usage of the runtime API, a CUDA application must be written in the specialized programming language CUDA C. This language is nothing more than a slightly augmented version of standard C, with functionality relative to accessing and using installed CUDA-enabled Devices.

The CUDA C compiler (nvcc), produces assembly code for the PTX instruction set. Then, the CUDA assembler (ptxas) converts this intermediate assembly code into a `cubin' specialized object format, which after getting linked to the necessary installed cuda libraries, is converted to an executable, native to the operating system.

7.2. Programming Model

According to the cuda parallel programming model, a developer spots some portion of a program which admits a high level of parallelism and chooses to run it on an installed CUDA device.

This portion of the program gets implemented as a regular C function with some special characteristics, that according to CUDA terminology is called a "kernel".

This kernel function gets executed by each thread that gets created on the CUDA device, but with different arguments for each thread. Actually, the arguments are the same across all threads and refer to data that have been stored in device memory. However, as we will see in detail later, the CUDA C language defines some special variables which have a different value for each thread, such that each thread can calculate it's unique numberical id. Each thread can use it's unique numerical id in order to choose which of the data in device memory it is going to use. Only a certain kernel function can be run by the threads of a CUDA device at a time.

The most abstract and simplified way to view a CUDA device, is as a set of multiprocessors that have access to a large device memory, each containing a certain number of processing cores. Multiprocessors cannot communicate with eachother, while processing cores within each multiprocessor can communicate with eachother, through a small but fast shared memory that each multiprocessor provides to it's processing cores. Furthermore, processing cores within each multiprocessor can be synchronized by concurrent programming primitives such as atomic operations and barriers.

In those terms, CUDA provides the capability to develop parallel programs in two levels of parallelism. At the first level, between independend multiprocessors, we have coarse grained parallelism, where there is no communication between them. At the second level, between processing cores within each multiprocessor, we have fine grained parallelism with cooperation and sharing of data.

According to CUDA terminology, this model of parallel computation is called STMD (Single Task Multiple Data). It is similar to SIMD, but does not require all the code to follow the same execution path; instead it enables programmers to write code which specifies the execution and branching behavior of a single thread. If, during execution some of the threads which are being executed in parallel diverge, the hardware automatically serializes the branch and executes each branch path independently. This hardware

architecture enables programmers to write thread-level parallel code for independent threads, as well as data-parallel code for coordinated threads.

A CUDA program is comprised of CPU code which allocates memory on the GPU, transfers necessary data for the CUDA threads (from host to device memory) and then launches a kernel. The allocation and transfer of data on the GPU is performed through functions defined by the runtime API (cudaMalloc, cudaMemcpy). The kernel launch is performed asynchronously, with the use of a special CUDA C statement. As long as the kernel is being executed on the GPU, every runtime API function which operates on the GPU will block waiting. After the kernel execution finishes on the GPU, results are transfered (again with cudaMemcpy) from device to host memory Figure 7.1

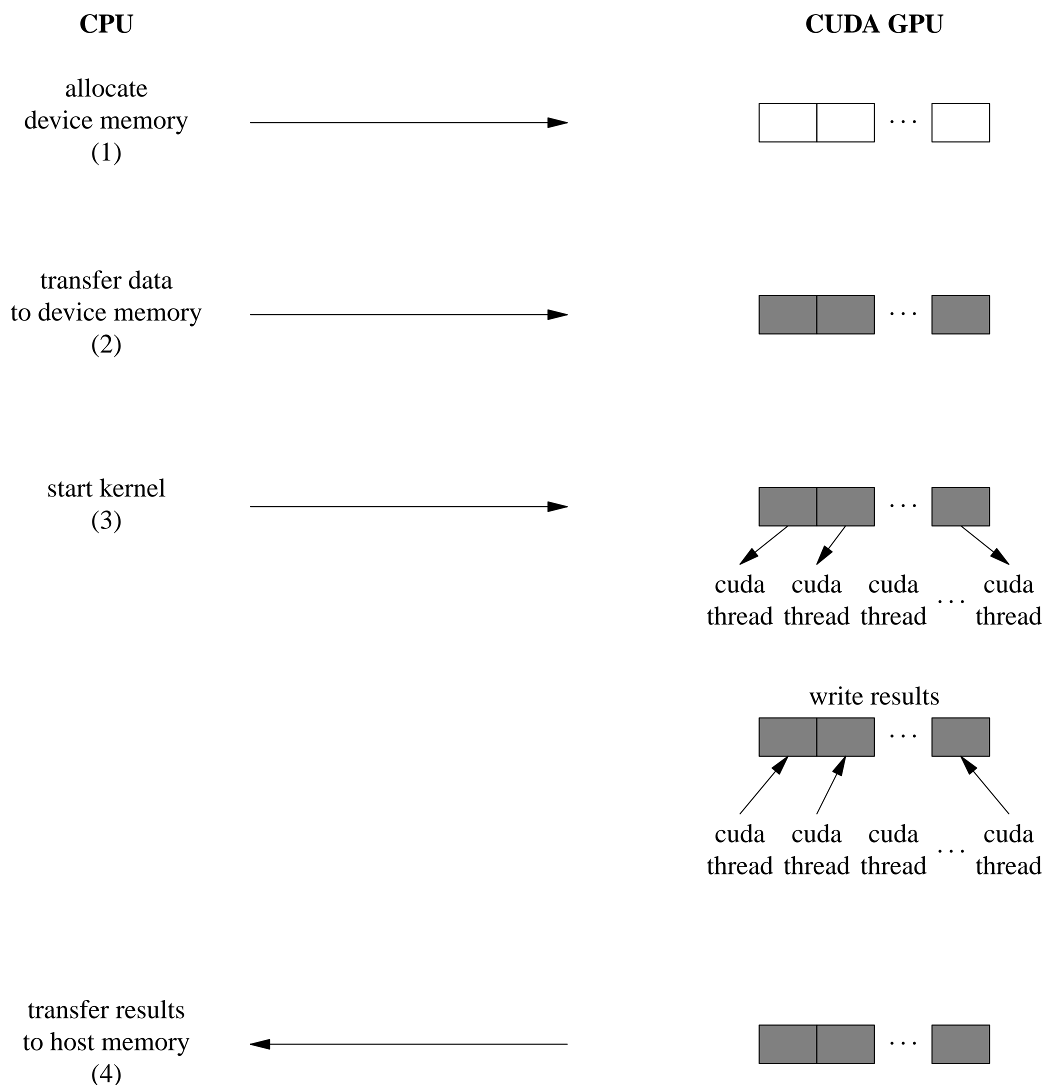

Figure 7.1. Workflow of a typical interaction between the host CPU and an installed CUDA Device, acting as a multicore co-processor.

7.2.1. Allocating and Transfering Data

Device memory allocation can be performed with various functions defined by the runtime API. However, the simplest and most straightforward way is to use the cudaMalloc function, and is sufficient for most circumstances. The prototype of the cudaMalloc() function can be seen below:

```
cudaError_t cudaMalloc        (void **     devPtr, size_t   size)
```

The first argument is the address of a pointer, which gets "filled" with a memory address in the device address space, and points to the allocated space. The first argument is the number of bytes we want to be allocated. As with every runtime API function, the returned value either equals cudaSuccess or cudaError, according to whether the call has been completed successfully or unsuccessfully respectively

The transfer of data to and from device memory, is also performed with various functions defined by the runtime API. The simplest and most straightforward way is through cudaMemcpy, which is also sufficient for most usage scenaria. The prototype of cudaMemcpy is shown below:

```
cudaError_t cudaMemcpy(void* dst, const void *src, size_t count,
                       enum cudaMemcpyKind kind)
```

The first argument is the value of a pointer that has been "filled" by an allocation call as described above. The second argument is the value of a pointer that points to the data in host memory we want to be transfered to device memory. The third argument is the size of the data to be transfered, in number of bytes. The fourth argument can either equal cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost and defines whether data is being transfered from device to host memory or vice versa respectively. In the first case, the semantics of the first two arguments are as described above, while in the second case, the first argument corresponds to host memory and the second to device memory.

The transfer of data from and to the device is being performed through the high speed PCI-X bus. However, it can be a possible bottleneck in performance, especially in cases where we have the transfer of a lot of data and/or the need for repetitive transfer of data between repetitive kernel calls.

Assuming the available CUDA device supports it, we can use "pinned" host memory instead, which becomes common between the host and the device (by excluding it from being paged), and thus we no longer need to transfer back and forth. For more information please refer to the official documentation[1].

7.2.2. Starting a Kernel

CUDA organizes the threads that execute a kernel function in a two-dimensional grid of blocks, where each block is a three-dimensional cube of threads. This means that the unique (numerical) id of each thread is, in essence, five-dimensional. As we said before, each thread can have knowledge of it's multidimensional id, by taking the value of some special variables. The value of those special variables is different in each thread context, i.e., each thread sees a different value.

As we said, blocks are arranged on a two-dimensional grid. Thus, the id of each thread block is two-dimensional and the special variable that contains the block id to which some thread belongs, is blockIdx, which contains the fields blockIdx.x and blockIdx.y.

Each block defines a three-dimensional thread space and the variable that contains the three-dimensional id of each thread within the block it belongs to, is threadIdx which contains the fields threadIdx.x, threadIdx.y and threadIdx.z.

Each thread uses it's multidimensional id in order to select the data that correspond to it. That is, data transfered on device memory, if need be, can be viewed by the threads as a five-dimensional array (of some data type).

The specialized statement for starting a kernel function on the device, takes as arguments the size of the two grid dimensions, the size of the three block dimensions and the call of the kernel function:

---

[1] http://www.nvidia.com/cuda

```
function_name <<<gridsize, blocksize>>> (arg1, arg2, ... , argn)
```

Arguments gridsize and blocksize are of the data type dim3, and a variable a of this type contains fields a.x, a.y and a.z. Arguments arg1 up to argn correspond to the common arguments that the kernel function will take across all threads that execute it, corresponding to either ponters on data in the device memory, or to scalar values, as in any usual C function.

Since the kernel call is performed asynchronously, there is no return value. As a result, after each thread has finished it's calculation, it returns it's results by copying them on device memory, from which they can be later copied back to host memory as we demonstrated above.

7.2.3. Kernel Functions

Functions that are meant to be executed from cuda threads as kernel functions should be declared the following way:

```
__global__ void function_name(arg1, arg2, ... , argn)
{
      ...
}
```

Beyond returning void and preceding their declaration with thei special directive __global__, these functions cannot recursively call themselves. They can, however call other functions which are declared with the special directive __device__ and return values like any regular C function. Of course, those functions cannot recursively call themselves either.

7.3. Warps and Divergence

As we said before, each multiprocessor is assigned a certain set of thread blocks. At any moment in time, some of it's assigned thread blocks is acive'', meaning that the threads it contains are being executed on the multiprocessor. Even though each block defines a three dimensional space of threads, the enumeration of threads can be viewed as one-dimensional:

```
int plane  = (threadIdx.z*(blockDim.x*blockDim.y));
int row    = (threadIdx.y*blockDim.x);
int col    = threadIdx.x;

int id     = plane + row + col;
```

Imagine the three dimensional structure of a thread block as a rubrik cube. Then, threadIdx.z identifies the plane (with dimensions blockDim.x $\times$ blockDim.y), across the z-axis, on which the thread resides. Then, within that plane, threadIdx.y identifies the row (of length blockDim.x) on which the thread resides. Finally, threadIdx.x identifies the exact spot where the thread resides on the row.

Each multiprocessor, as of today (independently of compute capability), contains eight computational cores (e.g., ALUs). There is only one instruction fetch unit per multiprocessor and each fetched instruction gets repeated four times, each time with different operands. That is, each fetched instruction gets executed 4 x 8 = 32 times, each with different operands.

As long as a certain block is active on a multiprocessor, we have time-sharing between groups of 32 consecutive (according to the serial enumeration we demonstrated) threads. A group of 32 consecutive threads that is being executed at any instant on a multiprocessor, is called a warp. When an instruction gets fetched, the 32 different operands that are needed for it's replayed execution, correspond to the warp that is being executed at the time. However, it is not guaranteed that all threads of a warp will be executing the same instruction at any instant. For example, we might have a divergence of execution paths between threads, as a result of an if condition.

In case different threads follow different execution paths, each subset of threads in a warp that follow the same execution path gets executed one after another. That is, in case there are x such subsets within a warp, the execution time of this warp will be x times larger. Actually, each time a subset of threads gets

executed, their common instruction gets indeed executed 32 times, but memory writting is disabled for processing cores that correspond to threads which dont belong in this subset. After all execution paths have been executed, all threads converge back to a common execution path.

7.4. Shared Memory and Register File

Each multiprocessor provides access to a small (16KB currently) and fast memory, to the threads of it's assigned blocks. In particular, the 16 kilobytes of a multiprocessor's shared memory, are divided evenly among it's blocks. The threads of each block can either use it as a means for communication, or as a scratchpad for computations, since the device memory we have been refering to so far, also named global memory in cuda terms, is a lot slower.

To use shared memory in our program, we pass a third argument to the specialized statement for starting a kernel, that defines how many bytes of shared memory need to be allocated per multiprocessor:

```
function_name <<<gridsize, blocksize, smemsize>>> (arg1, ... , argn)
```

The number of shared memory bytes needed per multiprocessor equals the number of shared memory bytes used in the kernel function times the number of threads per multiprocessor (blocks per multiprocessor times threads per block).

Now, within the kernel function, using the special keyword __shared__, we can declare variables that reside in shared memory and are common among the threads of a block, as for example:

```
__shared__ int smem_array1[128];
__shared__ char smem_array2[256];
```

Additionally, each multiprocessor contains a register file memory, with a certain number of 32bit registers that varies depending on the compute capability of the device. As with shared memory, register file space is evenly shared among the blocks of a multiprocessor.

7.5. Grid size, Block size and Occupancy

The total number of threads to execute a kernel on a CUDA device, is a function of the grid size (number of blocks) and the block size (number of threads per block).

Different ratios between those two parameters can yield the same total number of threads, but for each occasion, depending on device capabilities and the kernel function's characteristics, there is a golden ratio that gives the best possible performance.

For optimal performance we need as many threads as possible. In particualr, since the scheduling of threads is performed in terms of warps, we need as many warps as possible per multiprocessor. The reason for this is that each time some warp initiates a request for accessing the global memory, the shared memory or even the register file, there is a certain latency time for the completion of this request. Until this request is completed, another warp, assuming there is one available in the currently active block, can be scheduled on the multiprocessor instead of leaving it idle. This practice is analogous to pipelining. The next warp to be scheduled on the multiprocessor, must not have any data dependendencies with those waiting, thus it is desired to have as many warps available as possible at any time in each block.

There is a restriction on the maximum numbr of warps that can be assigned on a multiprocessor. As of today, this upper bound is 24 warps (or 768 threads) per multiprocessor. The number of warps we managed to assign over the maximum number of warps that can be assigned on a multiprocessor, defines how well we hide latency and in CUDA terms is named GPU occupancy:

$$occupancy = \frac{\#\ warps\ per\ MP}{24} = \frac{\#threads\ per\ block/32}{24}$$

Note however that there is another hardware imposed restriction, which is a maximum of 512 threads per block. This means that it would not be ossible to achieve 100% occupancy by using only one block per multiprocessor, even if we could indeed fit 512 threads per block.

Depending on the number of blocks we have a corresponding amount of register file and shared memory space available per block. The maximum number of threads in a block is restricted by the requirements of the kernel function, either in number of registers, or in amount of shared memory.

Thus, we must choose the values of the block size and the grid size in a way that maximuzes occupancy, given the restrictions imposed by the requirements of the kernel function in resources. If the requirements of the kernel function are overwhelming, it is not possible to achieve high enough occupancy.

CHAPTER 8

PARALLEL SEARCH WITH CUDA

This section describes how we utilized the Nvidia CUDA technology for performing a parallel n-marks OGR search, according to the partitioning of the search space we have demonstrated.

Nvidia CUDA is the most prevalent among technologies allowing the usage of multicore GPUs for solving highly data parallel problems, without having to use the (problem irrelevant) graphics API. For not cluttering this section by describing how Nvidia CUDA works, when necessary, we refer to relevant sections provided at the appendix.

For solving the problem of finding an OGR in parallel with CUDA, having P cuda threads available, we create the first P pieces of the search space, distribute them to the cuda threads and let them work. While waiting for the results, we create the next batch of P pieces and so on. Note that because all pieces cover the same number of search space branches (e.g. have equal sizes), we anticipate that all threads will finish their search at about the same time. This process is depicted as a flowchart in Figure 8.1 and as an algorithm in Algorithm 8.1.

As explaied, our partitioning of the search space yields an *Embarassingly Parallel* problem, meaning that each thread works in isolation and does not require any communication with the rest. The kernel executed by each thread is the C implementation of the final form of the search algorithm Algorithm 5.3. The data that need to be transfered to the global memory of the device, are the arguments of the search algorithm for each thread. Thus, we transfer the initialization values vector `G` and the start and ending ruler for each piece, as depicted in Figure 8.2.

In order to achieve high occupancy (SEE) we must maximize the number of threads per multiprocessor. This number is limited either from the number of registers, or the amount of shared memory that a multiprocessor can provide to it's threads, given how many registers and/or how much shared memory is required for the execution of the kernel function per thread.

The CUDA device we are going to use is the Nvidia GeForce 8800 GTX and it's limitations on resources are depicted on the following table:

Nvidia GeForce 8800 GTX	
Compute Capability	1.0
Global Memory	804585472 Bytes (767 MBytes)
Shared Memory per Multiprocessor	16384 Bytes (16 KBytes)
32 bit Registers per Multiprocessor	8192
Maximum Warps per Multiprocessor	24 (768 Threads)
Maximum Threads per Block	512
Multiprocessors	16
Clock Rate	1.35GHz

Table 8.1. Resource limitations of a GeForce 8800 GTX, which we will be using for experimentation.

As reported by the compiler, each thread requires 20 registers for executing our kernel. Due to the restricted number of available registers (ignoring register allocation granularity), no more than 8192/20 = 409 threads can reside on each multiprocessor. Making sure this number is a multiple of warp size (32), this means we cannot have more than 384 threads per multiprocessor. With a maximum of 768 threads per multiprocessor, the maximum occupancy we can achieve is exactly 384/768 = 50%.

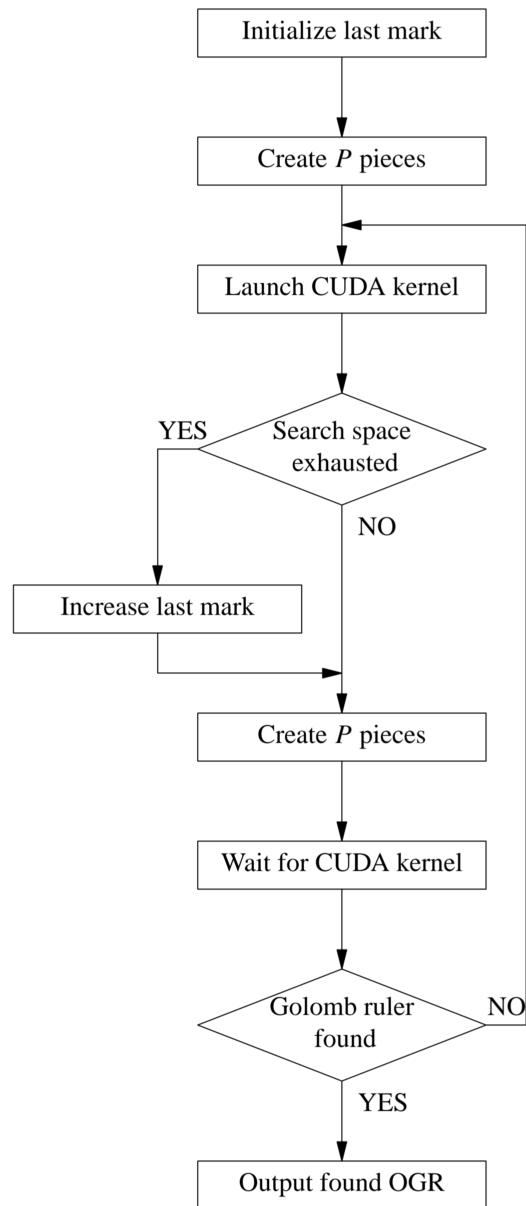

Figure 8.1. Flowchart description of how we utilize CUDA for performing the OGR search in parallel.

algorithm CUDA-OGR-SEARCH(n, G[])
1 Initialize K according **to** Eq. 4.1.
2 **do**
3 Create P pieces
4 Launch CUDA kernel; each thread runs Algorithm 5.3 on it's piece
5 **if** search space of rulers with length K exhausted
6 Increase K
7 **while** Golomb ruler of length K not found
8 -- *Found Golomb ruler is guaranteed to be optimal.*
9 **Output** found OGR

Algorithm 8.1. Algorithm to be executed on the CPU, for using the GPU as a coprosessor to find an OGR with n marks, utilizing Algorithm 5.3.

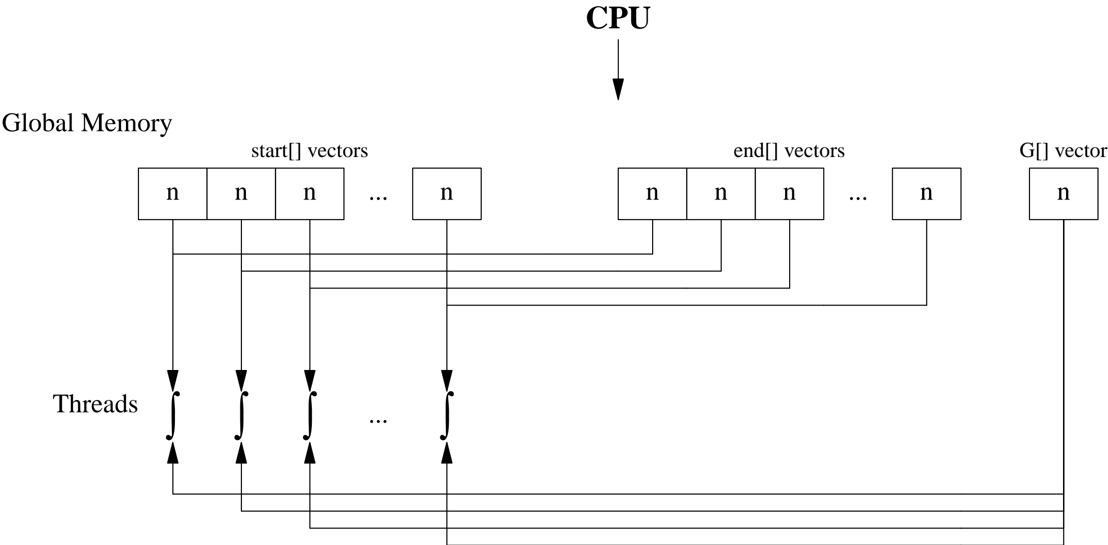

Figure 8.2. Transfer of search space pieces from host to device memory.

We will first present the performance results of the naive approach, where all variables needed by the algorithm reside in global memory. Our first attempt does not require the use of any amount of shared memory, thus the values of the block and grid size are only decided by restrictions relative to registers. Next, we will apply best practices such as utilization of shared memory and elimination of bank conflicts, in order to observe any performance enhancements they induce.

We can achieve this maximum number of threads per multiprocessor, either with many small blocks (larger grid size, smaller block size), or with fewer larger blocks (smaller grid size, larger block size). What would be more efficient depends on the kernel's characteristics. If, for example, __syncthreads is used a lot, in which case the threads of a block wait until memory write requests complete, then it would be desirable to have many alternative blocks that could get scheduled in it's place. In the general case however, it is important that blocks contain many warps, so that latencies (in accessing global memory, shared memory or even registers), gets hidden efficiently (SEE).

In our case, the treads within a block do not have to get synchronized, so can just define one big block per multiprocessor (grid size = 16), with 384 threads each (block size = 384) and yield an occupancy of 50%. The resulting performance of our first (naive) approach in implementing the $\text{backtrack}_{bv}^{mp}$ algorithm on CUDA, labeled GPU, can be seen in Table 8.2 and in Figure 8.3.

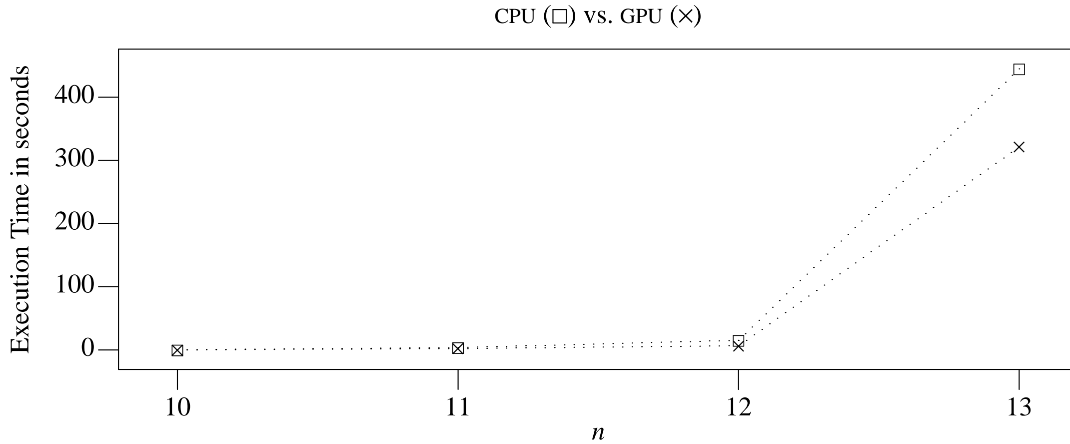

Figure 8.3. Performance results of the GPU approach, where all the algorithm's variables reside in global memory.

	n = 10	n = 11	n = 12	n = 13
Configuration (*gridsize* × *blocksize*)	32 × 192	32 × 192	32 × 192	32 × 192
Occupancy	50%	50%	50%	50%
GPU time (sec)	0.011	2.07	6.45	321.22
CPU time (sec)	0.141	3.560	15.130	444.722
Speedup (vs. CPU)	12. 81 ×	1. 71 ×	2. 34 ×	1. 38 ×
Piece Size (per thread)	555e3	555e5	145e7	2178e8

Table 8.2. Performance results of the GPU approach (graph on Figure 8.3).

Vectors `start` and `end` are only used for initializing vectors `list` and `dist`. These vectors, together with the `G` vector, are used continuously during the algorithm's execution. In our first, naive approach, these vectors reside within the slow, off-chip global memory of the CUDA device.

Attempting to increase performance, in our seccond approach, each thread will maintain vectors `G`, `list` and `dist`, within the portion of shared memory that belongs to it. All that needs to be done, is to include the amount of shared memory needed by each thread block at the execution configuration, and for the first thread of each block, to copy the `G` vector from global to shared memory upon starting. That is, `G` will only exist once within the search memory portion of each block, and will be used by all the threads of the block Figure 8.4.

Now, the execution of the kernel by each thread, also has requirements in shared memory. This means that the maximum number of threads per multiprocessor might be further limited. This, in turn, moderates our expectations for increase in performance.

The amount of shared memory required by each thread is a function of n, the number of marks. In particular, as n increases, the length K (of candidate Golomb rulers we try) is set within ranges of larger values. But the sizes of `list` and `dist` equals $K/32$ and is thus indirectly dependent on n. Also, the size of the vector `G` is directly dependent on n, but it is not important because it only resides in shared memory once for each block.

Even though for each n the requirements in shared memory for vectors `list` and `dist` increases as K increases, for defining the grid and block sizes, we will assume K is constant, equal to the largest value it

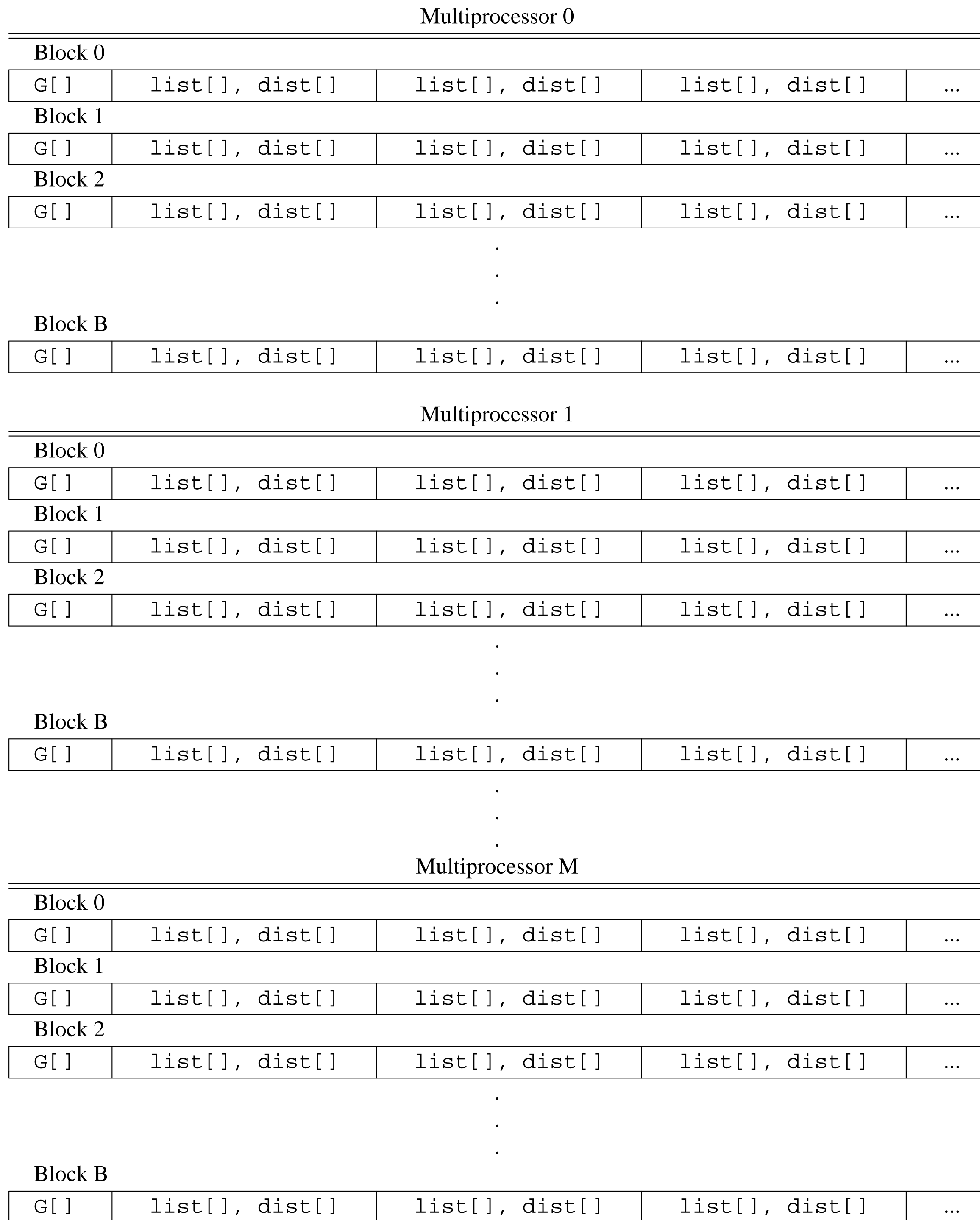

Figure 8.4. Layout of algorithm's variables in shared memory for our second approach.

is going to be set to, which is the actual length of the OGR to be found. In Table 8.3 we can see the requirement in shared memory per thread as n increases, and the limitation it imposes on the maximum number of threads per multiprocessor.

n	max K $= G(n)$	list[], dist[] length $= K/32$	required shared memory per thread	max threads per multiprocessor
10	55	2	8	1024
11	72	3	12	682
12	85	3	12	682
13	106	4	16	512
14	127	4	16	512
15	151	5	20	409
16	177	6	24	341
17	199	7	28	292
18	216	7	28	292
19	246	8	32	256
20	283	9	36	227
21	333	11	44	186
22	356	12	48	170
23	372	12	48	170
24	425	14	56	146
25	480	15	60	136
26	492	16	64	128

Table 8.3. How increasing n also increases shared memory requirements per thread, which in turn decreases the maximum number of threads per multiprocessor.

As can be seen, up to $n = 15$ the limitation imposed due to register requirements, at 409 threads per multiprocessor, dominates the limitation imposed due to shared memory requirements. However for larger n values, the limitation imposed due to shared memory requirements dominates. In particular, as n increases over 15, we anticipate that as occupancy lowers, performance will degrade accordingly.

Performance results for our second approach are depicted in Table 8.4 and Figure 8.5. We observe that performance has increased in comparison to our first approach. However, we can see that unlike our first approach, we now have warp serializations as a result of the fact that we utilize shared memory without taking into consideration bank conflicts (SEE). In our first approach, we did not have any warp serializations since we did not utilize shared memory and thus did not have any bank conflicts.

In our third approach, we will further enhance performance by avoiding bank conflicts. Each multiprocessor's shared memory is structured in 16 banks. In each group of 16 consecutive threads, we must make sure that all shared memory accesses of each thread are confined within a different shared memory bank from the rest Figure 8.6.

Upon starting up, each thread defines which bank corresponds to it. For example, assuming that variable `sm` corresponds to the shared memory of a block and that each thread views the bank as an array of integers:

```
int *bank = &sm[threadIdx.x % 16]
```

The elements of this bank can be traversed vertically, by defining a stride factor of 16:

```
#define STRIDE(i) (i)*16

bank[STRIDE(0)]
bank[STRIDE(1)]
bank[STRIDE(2)]
```

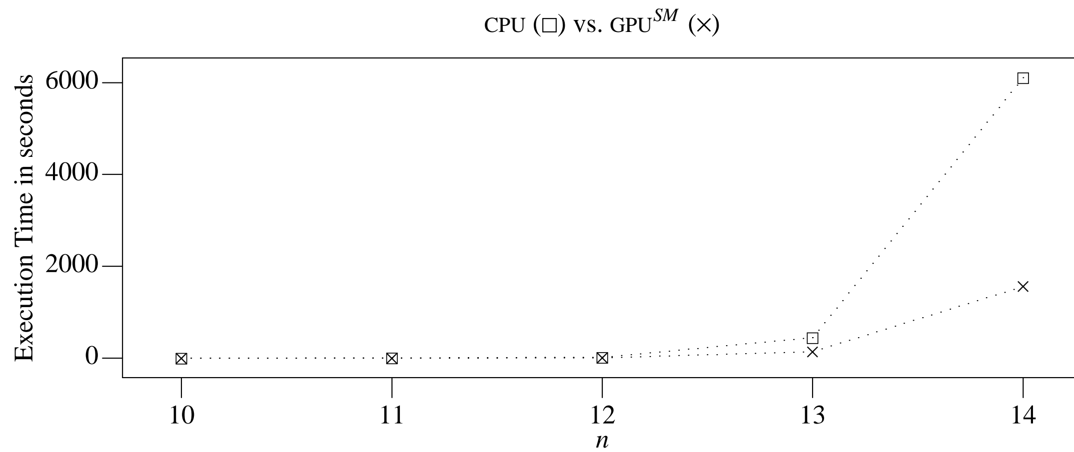

Figure 8.5. Execution time versus n for our second approach, GPU^{SM}.

	n				
	10	11	12	13	14
Configuration (*gridsize* × *blocksize*)	32 × 192	32 × 192	32 × 192	32 × 192	32 × 192
Occupancy	50%	50%	50%	50%	50%
GPU^{SM} time (sec)	0.005	1.007	3.95	139	2018
CPU time (sec)	0.141	3.560	15.130	445	6107
Speedup (vs. GPU)	28. 2 ×	3. 53 ×	3. 83 ×	3. 20 ×	3. 02 ×
Piece Size	555e3	555e5	145e7	2178e8	191199e8

Table 8.4. Performance results of our second approach, where all the algorithm's variables reside in shared memory.

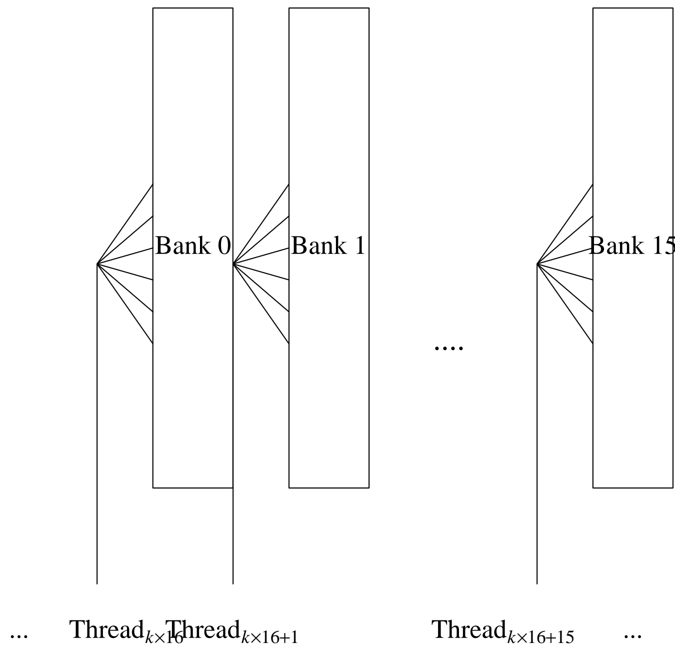

Figure 8.6. In order to avoid shared memory bank conflicts, for each consecutive group of 16 threads, each thread accesses a different bank.

```
...
```

So far, it is clear that each bank is shared by a subset of the threads within a block. Consider now that these threads share their bank with each one occupying m integers. The start of the m integers portion that corresponds to each thread can be defined the following way:

```
int *mine = &bank[(threadIdx.x/16) * m]
```

since the members of each next group of 16 threads will occupy their own m integers portion within each bank.

Thus, each thread can finaly access it's m integers within it's bank with a stride of 16 elements:

```
mine[STRIDE(0)]
mine[STRIDE(1)]
...
mine[STRIDE(m)]
```

In our case, each thread's portion within it's bank, is where it keeps vectors `list` and `dist`, $m = K/32$ integers each:

```
int *mine = &bank[STRIDE((threadIdx.x/16) * (2 * m))]
int *list = &mine[0]
int *dist = &mine[m]
```

Hereafter, we only have to make sure that accesses to vectors list and dist use a stride factor of 16, that is, replace each `list[i]` statement with `list[STRIDE(i)]`.

Performance results of our third and final approach are depicted in Figure 8.7 and Table 8.5. where we can see that without bank conflicts, we no longer have any warp serializations either. Note that the practice of avoiding bank conflicts diminshes warp serializations, however it also introduces extra calculations dute to the utilization of the stride factor.

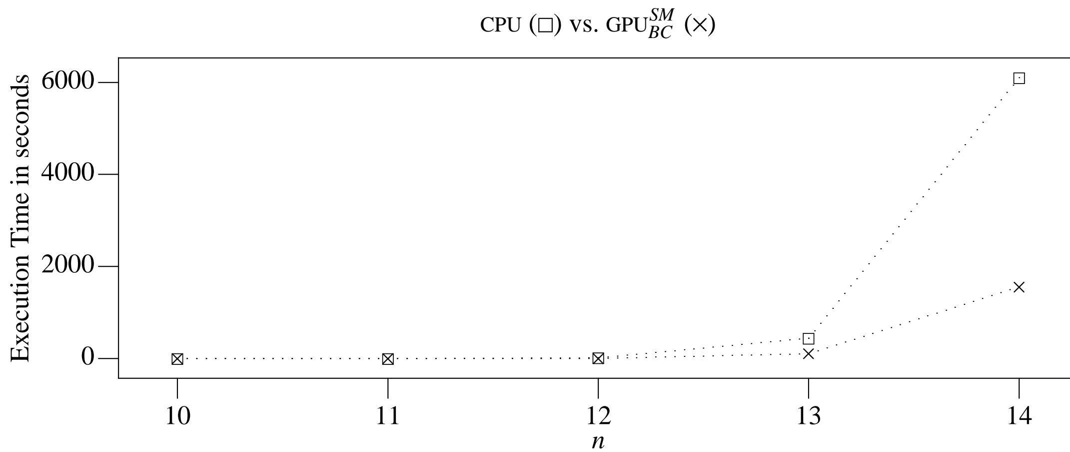

Figure 8.7. Execution time versus n for our third, final approach, GPU$_{BC}^{SM}$.

	n				
	10	11	12	13	14
Configuration (*gridsize* × *blocksize*)	32 × 192	32 × 192	32 × 192	32 × 192	32 × 192
Occupancy	50%	50%	50%	50%	50%
GPU$_{BC}^{SM}$ time (sec)	0.0047	1.001	3.033	107	1558
CPU time (sec)	0.141	3.560	15.130	444.722	6107
Speedup (vs. GPU^{SM})	30.00 ×	3.53 ×	4.98 ×	4.15 ×	3.92 ×
Piece Size	555e3	555e5	145e7	2178e8	191199e8

Table 8.5. Performance results of our third approach, where all the algorithm's variables reside in shared memory and bank conflicts have been eliminated.

CHAPTER 9

PARALLEL SEARCH WITH THE T.U.C. GRID

A Grid computing environment is comprised of heterogenous and distributed computational nodes such as desktop computers or dedicated servers and appears at the end user as a single, powerfull computational node. The term Grid was first introduced in the mid nineties and was refering to a proposed distribution of computational resources for engineers and scientists. In essence, a Grid is an aggregation of geographically dispersed heterogenous computational resources that composes and offers access to a unified and powerful virtual computer.

A Grid environment is usually formed by the collaboration between administrative domains of different institutions (such as for example universities) each one corresponding to a "Virtual Organization" (VO). The structure of a Grid environment can be given in terms of the following tiers:

Applications & Users
: Applications that require and can benefit from the distributed resources of the Grid, and their users.

Middleware
: Software that allows applications to utilize distributed resources within the Grid by presenting them as a virtual powerful computer, offering functionality such as submission, monitoring and management of jobs or authentication and authorization of users.

Resources
: Distributed computational and storage resources to be used by applications through the middleware.

Network
: Interconnection medium between distributed resources, either locally within an administrative domain or across administrative domains.

In this section we will use the T.U.C. Grid computer for solving our problem, e.g. the search for an OGR with n marks, in parallel, according to the partitoning of the search space we have described. The T.U.C. Grid is comprised of 44 similar HP Proliant BL465c server blade computers with the following computational power characteristics:

- Two AMD Opteron processors, Model 2218(2.6GHz, 2MB, 95W)
- 4GB RAM, extendible up to 32GB interconnected via Gigabit ethernet.

Only 41 of those processors are dedicated computational nodes, since 4 of them support the operation of the Grid. This homogeneous system of computational nodes comprises a Grid cluster with the purpose of eventually integrating with the international Grid computing environment HellasGrid-EuroGrid.

For executing distributed applications, a user must first gain access to the user interface host of the Grid. This host has installed and configured necessary middleware software that permits the submition and management of tasks on the computational nodes. Various kinds of middleware software is installed in the user interface node of the T.U.C. Grid, including Torque, OpenMP and Maui.

In our case, all we have to do is start as many processes as there are computational nodes, which for the most part do not need to communicate. All processes execute the same program.

One way this can be done is by using the Torque Resource Manager. For executing the same process across *P* computational nodes, we append a new job in the jobs queue, using the `qsub` command:

```
qsub -q tuc -l nodes=P ogr-search.pbs
```

Argument `-q tuc` defines the name of the jobs queue which is `tuc' in our case. Argument `-l nodes=P` defines the number of computational nodes that we need to allocate for our enqueued job. The

progam `ogr-search-P.pbs` is a shell script that contains the commands that will utilize the allocated computational nodes. In our case there is only one such command included in the script:

```
pbsdsh /storage/tuclocal/kmountakis/ogr-search
```

The program `pbsdsh`, spawns a process for executing `ogr-search` on each allocated computational node, which is our actual C implementation of Algorithm 5.3.

Note that since each HP Proliant host has 2 processors with 2 processing cores each, the middleware uses it as a source of 4 processing nodes. Essentially, with 41 HP Proliant hosts, we have 41x4 = 164 available computational nodes.

The purpose of the program that each process executes is to repeatedly create and search it's own piece of the search space. In order for each process to know which piece it should create next, it essentially needs to know what is the "fringe" of the search. That is, which is the ending ruler of the "farthest" (from the start of the search space) piece that has been created and taken over by a process.

The most convenient way to achieve the desired instrumentation of the processes, is to use the common network filesystem available between the nodes of the T.U.C. Grid. The common point of reference between all processes will be a "status file" which at each instant contains the following information:

(1) If an OGR has been found and which one it is.

(2) The starting and ending ruler of the last piece taken over by a process.

Upon starting up, each process locks this file by use of the flock() system call and opens it for exclusive usage. It then reads the current fringe (the ending ruler) and after creating the next consecutive piece (of a certain size), it sets the fringe to the ending ruler of the piece it just created, unlocks and closes the status file. When a process finds an OGR of a certain length, it records it in the status file, but only if another process has not already recorded that it has found an OGR with a smaller length, or if an OGR with a larger length has been recorded. In each case, if the status file contains a recorded OGR, each process stops the search instead of creating the next piece and continuing. A flowchart describing the operation of each process is depicted in Figure 9.1.

Beyond conducting the parallel search, the status file helps us measure the duration of the search. The duration of the search can be defined as the time interval between starting it and the last modification time of the status file, after all processes have finished.

In case the search has to stop for some reason, as we have explained, because the whole process might even last for years, it would be desirable to be able to restart it, roughly from the point where it stopped. At each instance, the set of starting rulers of the P pieces being processed by the P processes, define the checkpoint of the search progress.

Let us assume that each process, upon starting up, opens and locks for it's exclusive use, some file named `checkpoint-`i, where i is an intger from 1 up to the number of processes P, which will be different for each process. Each process tries file names from `checkpoint-1` up to `checkpoint-`P and the first filename for which there is no already created file, defines the name integer i it is going to use. During it's operation, each process maintains within it's `checkpoint-`i file the starting ruler of the last piece it took over.

In case of restart, each process tries filenames in the sequence described earlier, aiming to open and lock the first unlocked `checkpoint-`i file. Then, instead of the fringe recorded in the status file, it uses the starting ruler recorded in it's `checkpoint-`i file and the piece size used during the previous (interrupted) search process, in order to define the first piece it will take over. After finishing with this first piece - reminiscent of the previous search operation, processes start using the status and `checkpoint-`i files regularly as described first.

In a distributed computing system such as the T.U.C. Grid, we expect to gain a linear speedup of the parallel OGR search. For the problem of finding an OGR with n=13 marks, for an increasing number of computational nodes, we can see that our expectations get confirmed, according to the results of Figure 9.2 and Table 9.1. Finally, in (Figure 9.3) and (Table 9.2) we can see the overall needed execution time for

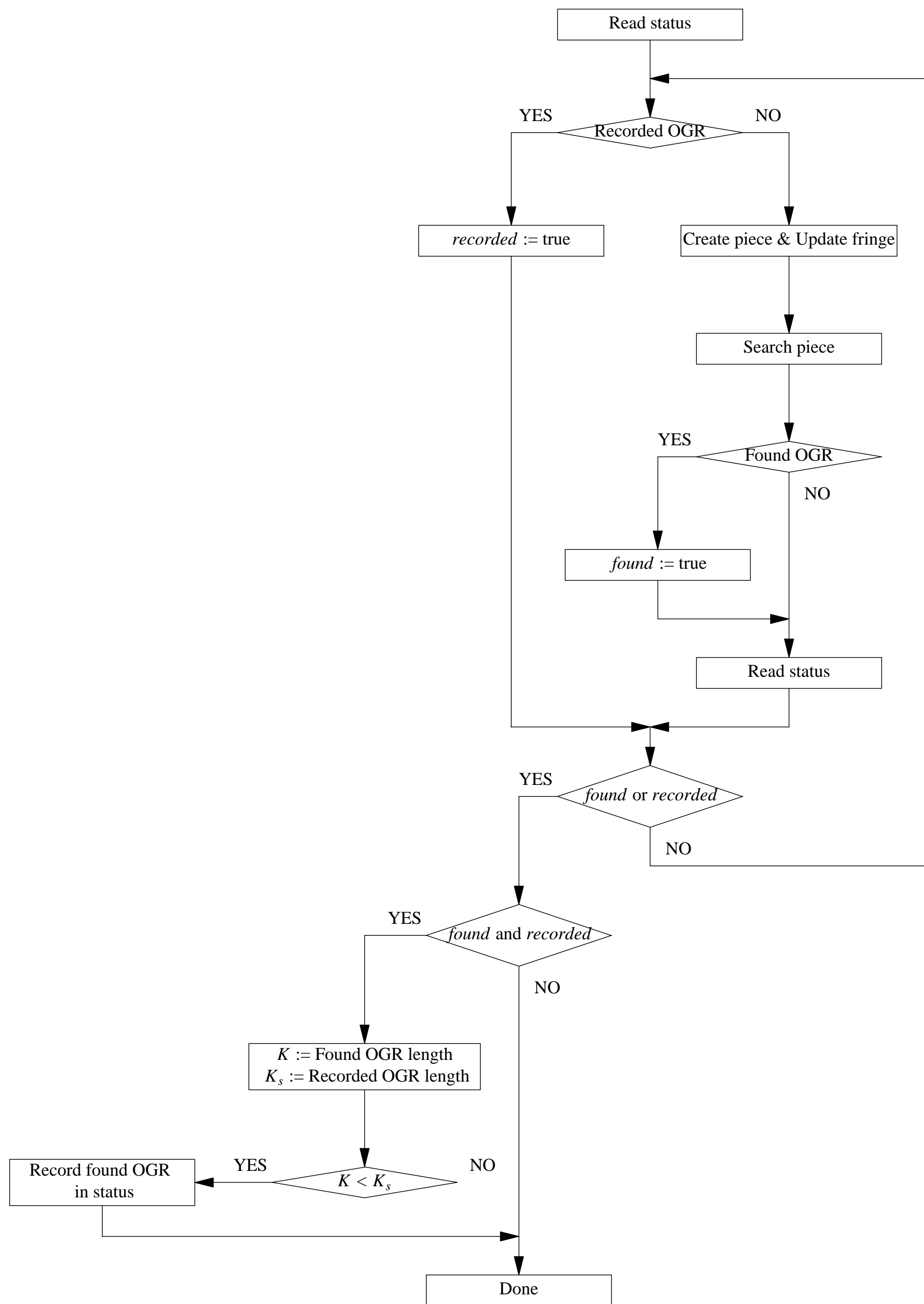

Figure 9.1. Flowchart description of how each computational Grid node operates during the parallel search.

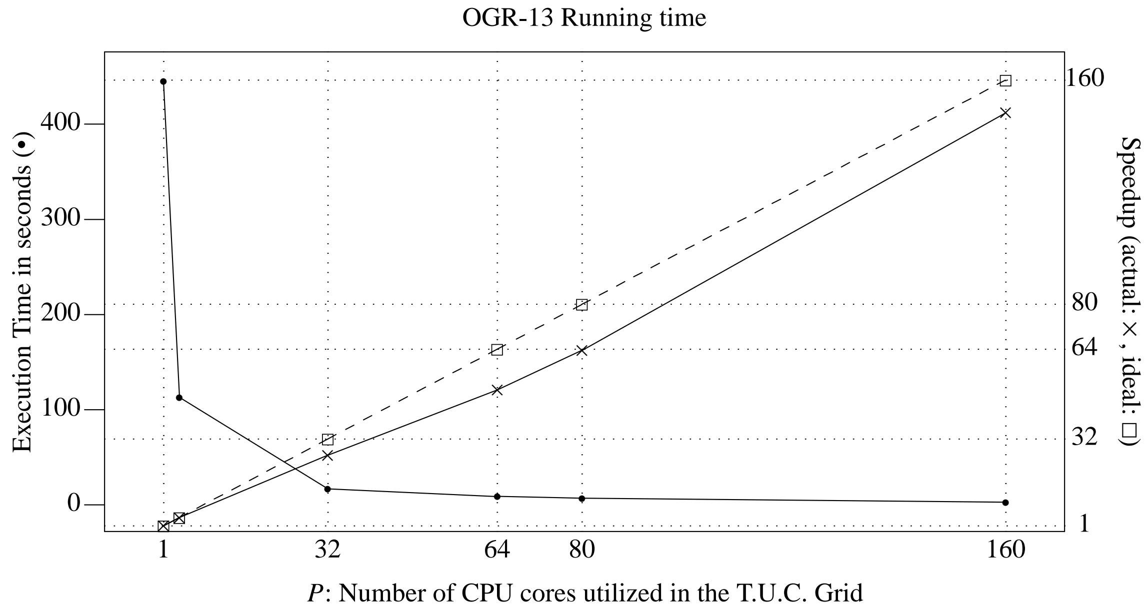

Figure 9.2. Execution time and actual versus linear speedup as the number of Grid nodes increases, for finding an OGR with $n = 13$ marks, using the BACKTRACK$_{MP,BV}$ algorithm.

P	Execution Time (sec)	Speedup		Efficiency
		actual	linear	
1	445	1	1	1
4	113	3.93	4	0.98
32	17	26.18	32	0.81
64	9	49.40	64	0.77
80	7	63.57	80	0.79
160	3	148.33	160	0.92

Table 9.1. Values of graph on Figure 9.2.

finding an OGR over increasing n values.

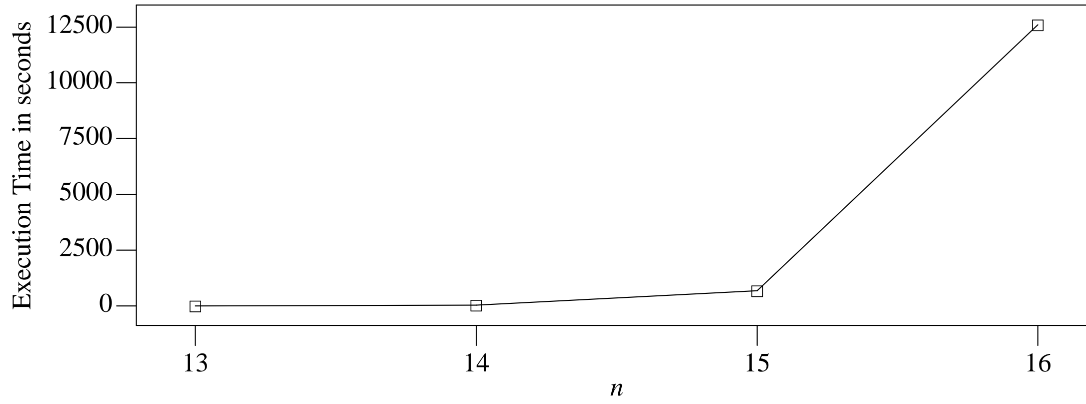

Figure 9.3. Execution time for finding an OGR with increasing n values, utilizing all $40 \times 4 = 160$ processing cores of the T.U.C. Grid (see Table 9.2).

n	Execution Time in seconds
13	3
14	39
15	675
16	12600

Table 9.2. Execution time for finding an OGR with increasing n values, utilizing all $40 \times 4 = 160$ processing cores of the T.U.C. Grid (see Figure 9.3).

CHAPTER 10

FUTURE WORK

In this chapter we will give a brief overview of some methods that would potentially enhance our work in searching for Optimal Golomb rulers.

10.1. Improving the CUDA Implementation

An alternative CUDA implementation could possibly yield a larger speedup with respect to the implementation of our parallelixation on the CPU.

We need an implementation of our search algorithm with fewer requirements in number of registers per thread. Our current implementation requires 20 registers per thread. On a GTX 8800 Nvidia CUDA device, this limits occupancy to at most 50%, which might possibly not be enough to hide the latency of memory and register file accesses (for details, refer to chapter 7).

It seems possible that such a "lightweight" CUDA implementation could be developed based on the imperative version of our backtracking search algorithm, given in chapter 4.

For each OGR-n instance (for each different n) this imperative version of the algorithm consists of n nested for loops. Thus, for each different instance of the problem, we would have to produce the corresponding C source code. But this would be trivial to accomplish by using for example a UNIX shell or an awk script.

What makes the imperative version suitable for the case, is it's simple structure, which allows us to implement the algorithm with only using a few variables. Subsequently, we suggest to investigate the possibility for implementing the imperative version of backtracking search, where it's simple structure requires the use of fewer variables, which in turn translates to fewer registers, allowing to maximize occupancy of the CUDA device.

10.2. Increasing $L(n)$ with Linear Programming

The function $L(n)$, which we defined in chapter 4, returns the smallest candidate Golomb ruler length that makes sense to try in our search process. Thus, the closer $L(n)$ is to the actual length of the Optimal Golomb ruler to be eventually found, the faster the search process will finish.

The way that we currently calculate $L(n)$ is based on certain observations regarding the properties of the Golomb ruler to be eventually found. Alternatively, for calculating $L(n)$, we can use the work of Meyer et. al in [7]. Their method for calculating $L(n)$ is based on a technique for solving maximization/minimization problems, called *Linear Programming*[6]. In particular, Meyer et. al describe how to express the problem of minimizing the length of a Golomb ruler with n marks, using various Linear Program formulations. Solving those Linear Programs with a method such as *Simplex* or *Interior Point* only takes a few seconds. With their method, the resulting value for $L(n)$ is always impressively close to the actual Optimal Golomb ruler(s) length with n marks.

We thus suggest that replacing our method for calculating $L(n)$ with an implementation of the Linear Programming method of Meyer et. al, will yield an equally impressive decrease in the time required to solve any OGR-n instance with our method.

10.3. A Dynamic Programming Algorithm for GR-n, K

A side-effect of our research in developing the backtracking search algorithm for finding Optimal Golomb rulers, is the design of a dynamic programming [1] algorithm for solving the GR-n, K problem.

The advantage of dynamic programming over backtracking search, is that we store the result of searching for Golomb rulers within a subtree (m, K) of the search space and recall that result when we need it again, instead of searching within this subtree again.

Searching for Golomb rulers within a subtree (m, K) is the basic building block of the GR-n, K problem. Taking a closer look along a certain level m of the search space tree (Figure 5.1), we find out that each subtree (m, K) appears multiple times. This means that we only need to store the result of processing (m, K) the first time and recall the stored information every other time we meet (m, K).

More information regarding this approach, including experimental measurements of a memoization implementation, is soon to be presented in a separate paper by yours trully.

CHAPTER 11

CONCLUSION

11.1. Overview of Our Work

Over the course of this thesis we have successfully managed to develop and evaluate a method for solving OGR-*n* by utilizing multiple computational nodes in parallel. Computational nodes can be any kind of processing cores, such as CPU cores distributed over a set of hosts, or GPU cores residing within an Nvidia CUDA device.

Our parallelization method consists of a pair of algorithms that allow for solving OGR-*n* for any *n* value, by utilizing multiple computational nodes in parallel. In chapter 5, we developed an algorithm for creating a (search space) piece of arbitrary size (Algorithm 5.1). In chapters 4 and 5, we developed an algorithm for searching for a Golomb ruler within the boundaries of some given search space piece (Algorithm 5.3). Combining these two algorithms allows for solving OGR-*n* in parallel by assigning search space pieces to computational nodes, in a producer-consumer way (Algorithm 5.2). Creating pieces is meant to be performed on the CPU by a producer process. Consuming pieces is meant to be performed by each individual computational node. Note that it is perfectly possible that the producer process utilizes any number of different types of computational nodes at the same time, such as for example CPU cores and GPU cores.

Other related projects such as project OGR of distributed.net and project GE of the Technical University of Crete also deploy their own methods of OGR-*n* parallelization. Their parallelization methods however, produces pieces of unpredictable size. The advantage of our OGR-*n* parallelization method over other currently deployed methods, is that it allows for arbitrarily choosing the amount of work assigned on each computational node, as the size of each piece (and thus the corresponding amount of work required) can be arbitrarily chosen.

Furthermore, we evaluated our work experimentaly on platforms that represent two extremes of the parallel and distributed computation practice. In particular, we evaluated our work on a GTX 8000 Nvidia CUDA device which is intended for massively parallel computations and on the Grid distributed computation system hosted by the Technical University of Crete, using our implementation of both the search space partitioning and the search space piece searching algorithm in the C programming language. Required running times for solving various instances of OGR-*n* on both these platforms are summarized on Table 11.1.

In this table we juxtapose running times (in seconds), for sequentially solving the OGR-*n* problem on a single CPU core, versus solving it in parallel on the Grid system and the GTX 8800 CUDA device. The CPU column shows running times across various OGR-*n* instances, with only one instance of the search algorithm running on one CPU core, processing one piece at a time. The GPU_{BC}^{SM} column shows running times across various OGR-*n* instances, with 384 CUDA threads (each assigned a search space piece) running our search algorithm on 128 GPU cores, organized in 16 streaming multiprocessors. The Grid column shows running times across various OGR-*n* instances, with 160 CPU threads (each assigned a search space piece) running our search algorithm on 160 CPU cores.

Our algorithm for finding Golomb rulers within the boundaries of a given search space piece is simple and easy to understand (Algorithm 5.3). due to it's simplicity, this algorithm can be easily implemented for a variety of platforms. Furthermore, our analysis of it's worst case complexity (Appendix A) can be used to develop accurate methods for estimating the running time required to solve any OGR-*n* instance.

n	CPU	GPU_{BC}^{SM}	CPU $\times$ 160 (Grid)	GPU SpeedUp	Grid SpeedUp
10	0.141	0.0047	-	$30\times$	-
11	3.560	1.007	-	$3.53\times$	-
12	15.130	3.033	-	$4.98\times$	-
13	445.000	107.000	3.000	$4.15\times$	$148\times$
14	6107.000	1558.000	39.000	$3.91\times$	$156\times$
15	-	-	675.000	-	-
16	-	-	12600.000	-	-

Table 11.1. Overview of resulting running times (in seconds) for solving the OGR-n where $10 \leq n \leq 16$ on a GTX 8800 Nvidia CUDA device (128 GPU cores at 1.35GHz) as described in chapter 7 and on the T.U.C. Grid system (160 CPU cores at 2.6GHz) as described in chapter 9.

11.2. Grid and CUDA results

Based on the characteristics of our parallelization method and based on certain assumptions about the capabilities of the CUDA device, it is possible to estimate a theoretical maximum expected efficiency for each. On any parallel/distributed computation platform, our parallelization method requires no communication among computational nodes. Thus, with (at least one) instance of our search algorithm running on each computational node, we can expect a maximum speedup (with respect to only utilizing one node of the same type) equal to the number of nodes, or in other words we can expect a maximum efficiency of 100%.

On the Grid system, running one instance of our search algorithm on each of the 160 available CPU cores, we can expect a maximum speedup value of 160, with respect to searching one piece at a time on a single CPU core. That is, for the Grid system, we can expect a maximum efficiency of $160/160 = 100\%$.

The actual efficiency achieved on the Grid system almost reaches our maximum expectations. In practice, solving the OGR-13 instance on the Grid system using all 160 CPU cores, happens roughly 156 times faster (Table 11.1) than solving it on a single CPU core, which translates to an efficiency of $156/160 = 97.5\%$, fairly close to the maximum expected value.

Note that we execute the exact same C program on both the GPU and the CPU cores. In CUDA applications that yield large speedup values, the implementation of the problem solution for CUDA is usually formulated in a different way than it is formulated on the CPU. The kernel program is usually written in a way that matches the SIMD programming model of the CUDA platform, where fine-grained parallelism takes place and CUDA threads work in close collaboration. In our CUDA application however, where we simply spawn multiple instances of our Golomb ruler search algorithm on each GPU core (where no communication takes place between GPU cores), we essentially treat each GPU core as an autonomous processing core.

This way, comparing the time needed to solve a certain OGR-n instance on the CUDA device and on a single CPU core, essentially maps to a direct comparison between the capability of an individual GPU core and an individual CPU core to execute the C program that implements our search algorithm.

Let us now assume that a GPU core can execute our C implementation of the search algorithm c times faster or slower than a CPU core. On the CUDA device, running multiple instances of our search algorithm on each of the 128 available GPU cores, we can expect a maximum speedup value of $c \times 128$, again, with respect to searching one piece at a time on a single CPU core. Thus, for the CUDA device, we can expect a maximum efficiency of $(c \times 128)/128 = c$. In practice, solving some OGR-n instances on the CUDA device, utilizing 128 GPU cores, happens, on average, 4.2 times faster (Table 11.1) than solving them on a single CPU core, which translates to an efficiency of $4.2/128 = 3.3\%$.

If each GPU core was equally capable to a CPU core with respect to running our algorithm, then with 128 utilized GPU cores we would expect speedup value of 128× with respect to only utilizing a single CPU core. Our interpretation of the actual speedup, is that each individual GPU core is $128/4.2 = 30$ times

slower than a CPU core in executing our implementation of the Golomb ruler search algorithm. In other words, that each individual GPU core can only reach 3.3% of the performance of an individual CPU core in executing our search algorithm.

In our effort to explain why this might happen, we will begin by excluding the following set of possible explanations:

Global Memory Accesses
: In our implementation of the search algorithm as a CUDA kernel program, once we download the necessary per thread data from the device's slow, off-chip global memory into the fast, per multiprocessor shared memory, we barely even use global memory again, except if a solution to our problem is found. In this case we only write one integer back to global memory. In particular for solving an OGR-n instance, we only access a region of n integers in global memory in read-only texture more, in which case (according to documentation) global memory accessess are being cached and it is thus not considered probable that they constitute a performance bottleneck.

Bank Conflicts
: As we have described in chapter 7, all bank conflicts are indeed being resolved by careful strided access to the algorithm's local variables that reside in shared memory. As a result, it is not possible that bank conflicts consitute a bottleneck in performance either.

Warp Divergence
: Due to the fact that our implementation of the Golomb ruler search algorithm as a CUDA kernel program contains a multitude of cases where a branch might be taken (i.e. if statements and loops), it would be possible that performance suffers due to warp divergence. However, as reported by the CUDA Visual Profiler[1] only 3% of taken branches are divergent within warps of 32 threads. Based on this, we conclude that it is unlikely that warp divergence constitutes a bottleneck in performance.

Even though we have followed all best practices described in the original CUDA documentation, so that in essence, GPU cores do not interfere with eachother, there are still many possible reasons why a GPU core would be much slower than a CPU core in executing a certain program:

Program Size
: Small programs can be handled with the same efficiency by either a GPU or a CPU core. For a small program that compiles into a short sequence of multiply-add instructions, with no branches, there really is no need for branch prediction and no need for data dependencies resolution. Larger programs however, can be handled much more efficiently by a CPU core than by a GPU core and our kernel program implementing our Golomb ruler search algorithm, is a large and complex program. For a large program which compiles into a long sequence of instructions, including loops, CPU-specific features for optimal Instruction-Level Parallelism (ILP) make a real difference in delivered performance. The more sophisticated architecture of a CPU core provides features that lack from a GPU core such as deep pipelining, out-of-order execution for resolving data dependencies and sophisticated branch prediction. The CPU architecture has been maturing for decades so that large and complex programs can be executed with maximum efficiency. The architecture of each individual GPU core on the other hand is very simple. Each GPU core is meant for performing simple operations that contribute to the collective effort of all GPU cores residing within a CUDA device for the completion of a task. Hundreds of GPU cores packed within a CUDA device cost almost as much as a quad core CPU.

Compiler Maturity
: Nvidia's *nvcc* compiler translates (CUDA-extended) C programs to PTX assembly programs. The PTX architecture has not been around for even nearly as many years as the x86 architecture.

---

[1] http://www.nvidia.com/cuda

Clock Speed

The clock rate of the CPU cores (2.6GHz) is almost double the clock speed of the GPU cores (1.35GHz).

Low Occupancy

Kernel's large size and complexity results in overwhelming requirements in shared memory and/or registers per CUDA thread, not enabling us to launch enough CUDA threads per Multiprocessor. As shown in chapter 7, we are only able to launch 384 out of a maximum of 768 threads per Multiprocessor, which results in only 50% occupancy. According to documentation, maintaining high occupancy helps in hiding the latency when accessing registers, shared memory and global memory. It might be the fact that inefficient latency hiding induces a serious performance penalty.

In conclusion, our search algorithm cannot be executed as efficiently by each individual GPU core as it can be executed by each individual CPU core because our C program that implements our Golomb ruler search algorithm is too large and complex to be handled by a GPU core and because it requires overwhelming register and shared memory resources per CUDA thread.

APPENDIX A

BACKTRACKING SEARCH COMPLEXITY

This chapter covers the derivation of the worst case complexity of the aforementioned backtracking search algorithm given in Chapter 4. We will be expressing worst case complexity in terms of how the number of needed recursive calls increases as a function of n. The worst case complexity of creating one branch of the search tree is $O(n)$ which is linear. In the worst case scenario, each and every possible branch in the search tree would have to be visited by the backtracking algorithm. For this to happen, each and every branch would have to yield a golomb ruler, but every time the last mark to be put at the bottom (i.e. x_1) would introduce conflicting cross distances. In that worst case scenario, the dominant contributing factor to the algorithm's overall complexity, would be how the total number of branches that need to be searched grows as a function of n. For this reason, we will be actually be expressing overall worst case complexity in terms of how the maximum number of branches that need to be searched grows as a function of n.

As we have already seen, we try a range of increasing values for x_n, up to the point where a golomb configuration for the remaining $n-1$ marks can be found, at which point the whole searching process stops. For each value K that we set x_n to, let us denote by $[\bullet][K]$ the corresponding subtree of possible configurations for the remaining $n-1$ marks.

The maximum value for x_n is bounded from above, since as we have showed in the "Golomb Rulers" chapter, one can trivialy construct a Golomb ruler for any n. Let $T(n)$ denote the length of such a trivialy constructed ruler, with n marks. It then follows that $x_n \leq T(n)$. Furthermore, in chapter "Backtracking Search" we have showed that the distances marks can be set to are bounded from bellow according to the following equations:

$$x_i \geq \begin{cases} G(i) & i \leq n-1 \\ \max\left(G(n-1)+1, n(n-1)/2+1, n^2-2n\sqrt{n}+\sqrt{n}-2\right) & i = n \end{cases}$$

In other words, in a worst case scenario, we will have to search at most all search space subtrees $[\bullet][G(n)], [\bullet][G(n)+1], \cdots, [\bullet][T(n)]$ until we find one that contains at least one branch which is a Golomb Ruler. We know which subtrees we have to search in the worst case. If we could somehow calculate the size (i.e. the number of branches) of each such subtree, we would also be able to calculate the maximum needed number of recursion tree branches.

Let us now focus our attention on search space subtrees. In each such subtree, the value of elements x_1 and x_n are fixed to 0 and K respectively. The value of elements $x_2, \cdots, x_{n-1}$ is variable, but bounded from above, since x_n is kept constant. In general, for each element i $x_i \leq x_i - 1 \leq x_{i+1} - 2 \leq \cdots \leq x_n - (n-i)$. For a subtree where $x_n = K$, we have that

$$
\begin{aligned}
&x_1 = 0 \\
&G(2) \le x_2 \le K - (n-2) \\
&G(3) \le x_3 \le K - (n-3) \\
&\ldots \\
&G(k) \le x_3 \le K - (n-k) \\
&\ldots \\
&G(n-1) \le x_{n-1} \le K - 1 \\
&x_n = K
\end{aligned}
$$

Returning to the example we saw last where $n = 5$, for the first subtree [11][•] (i.e. where $x_n = 11$), the values of the remaining $n-1$ elements are bounded as

$$
\begin{aligned}
&x_1 = 0 \\
&1 \le x_2 \le 8 \\
&3 \le x_3 \le 9 \\
&7 \le x_4 \le 10 \\
&x_5 = 11
\end{aligned}
$$

In other words, assuming we perform a depth-first search parsing of the corresponding search space subtree, the first branch to be produced would be $[0, 1, 3, 7, 11]$ and the last branch to be produced would be $[0, 8, 9, 10, 11]$.

We now know that in a recursion subtree, the elements of vectors (which correspond to it's branches), are bounded from below and from above.

In order to derive a mathematical expression for calculating the number of branches of a given subtree, it would help if we first considered the following trivial program for doing so:

```
program TreeSizeNaive_n(i)
1  count := 0
2  x[1] := 0
3  x[n] := G[n] + i
4  foreach x[n − 1] from  G[n − 1] to x[n] − 1
5    foreach x[n − 2] from G[n − 2]  to x[n] − 2
6      foreach x[n − 3] from G[n − 3] to x[n] − 3
7        ...
8          foreach x[2] from G[2] to x[n] − (n − 2)
9              count := count +1
10 return count
```

There will be as many branches, as many times the innermost statement will be executed. But now, it is easy to map this number to a nested summation like the following:

$$
count = \sum_{x_{n-1}=G(n-1)}^{x_n-1} \sum_{x_{n-2}=G(n-2)}^{x_{n-1}-1} \sum_{x_{n-3}=G(n-3)}^{x_{n-2}-1} \cdots \sum_{x_2=G(1)}^{x_2-1} \sum_{x_1=0}^{0} 1 \qquad \text{(Eq. 12.1)}
$$

Returning to the example we saw last where $n = 5$, for the first subtree [11][•] (i.e. $x_5 = 11$), it's total number of branches can be calculated by the following nested summation:

$$
count = \sum_{x_4=7}^{10} \sum_{x_3=3}^{x_4-1} \sum_{x_2=1}^{x_3-1} \sum_{x_1=0}^{0} 1 = 104
$$

unfold this summation to an algebraic expression. evaluating with nested loops (as shown above) is too slow and impractical. evaluating the size of a search tree is key to partitioning the search space. can be used to express the worst case complexity in as a function of n. can be used to express performance in number of (search tree) branches per second. size of each $[G(n)+i][\bullet]$ subtree divided by total wall clock time needed.

use the concepts of discrete integrals and discrete factorials. the factorial of a discrete function $f(k)$ is denoted as δf and can be defined as:

$$\delta f(k) = f(k+1) - f(k) \qquad \text{(Eq. 12.2)}$$

we will be utilizing the discrete factorial of the ``k choose l'' discrete function, in terms of k, which is defined as:

$$\delta C(k,l) = \delta \binom{k}{l} = \binom{k}{l-1}$$

the integral of a discrete function $f(k)$, reverses the effect of the δ factorial operator and is defined as:

$$\sum_{k=a}^{b-1} \delta f(k) = f(b) - f(a) \qquad \text{(Eq. 12.3)}$$

the innermost sum of the nested summation can be written in terms of the $C(k,l)$ function as:

$$\sum_{x_2=G(2)}^{x_3-1} 1 = \sum_{x_2=G(2)}^{x_3-1} \binom{x_2}{0}$$

since it is a well known property of $C(k,l)$ that $C(k,0) = 1$ for any k. from the definition of the discrete factorial operator δ in Eq. 12.2 we can also write this as:

$$\sum_{x_2=G(2)}^{x_3-1} \binom{x_2}{0} = \sum_{x_2=G(2)}^{x_3-1} \delta \binom{x_2}{1}$$

since we know that $\delta C(k,1) = C(k,0)$. from the definition of the discrete integral operator $\sum$ in Eq. 12.3 we can finally re-write the nested summation as:

$$\sum_{x_2=G(2)}^{x_3-1} 1 = \sum_{x_2=G(2)}^{x_3-1} \binom{x_2}{0} = \sum_{x_2=G(2)}^{x_3-1} \delta \binom{x_2}{1} = \binom{x_3}{1} - \binom{G(2)}{1}$$

wrap our last result with the second to last summation in the nested summation of Eq. 12.1 and get[1]:

$$\sum_{x_3=G(3)}^{x_4-1} \left[\binom{x_3}{1} - \binom{G(2)}{1} \right] = \sum_{x_3=G(3)}^{x_4-1} \binom{x_3}{1} - \sum_{x_3=G(3)}^{x_4-1} \binom{G(2)}{1} = \sum_{x_3=G(3)}^{x_4-1} \binom{x_3}{1} - \binom{G(2)}{1} \sum_{x_3=G(3)}^{x_4-1} \binom{x_3}{0}$$

applying the same principles as before, we can turn summations to the following algebraic expressions:

$$\sum_{x_3=G(3)}^{x_4-1} \binom{x_3}{1} - \binom{G(2)}{1} \sum_{x_3=G(3)}^{x_4-1} \binom{x_3}{0} = \left[\binom{x_4}{2} - \binom{G(3)}{2} \right] - \binom{G(2)}{1} \left[\binom{x_4}{1} - \binom{G(3)}{1} \right]$$

now we can rearrange the position of the components of the previous expression, to separate the ones which are functions of x_4 from the constant factors:

---

[1]Note that since $\binom{G(2)}{1}$ is a constant, $\sum_{x_3=G(3)}^{x_4-1} \binom{G(2)}{1} = \binom{G(2)}{1} \sum_{x_3=G(3)}^{x_4-1} 1 = \binom{G(2)}{1} \sum_{x_3=G(3)}^{x_4-1} \binom{x_3}{0}$

$$\sum_{x_3=G(3)}^{x_4-1}\sum_{x_2=G(2)}^{x_3-1} 1 = \binom{x_4}{2} - \binom{G(2)}{1}\binom{x_4}{1} + \left[\binom{G(2)}{1}\binom{G(3)}{1} - \binom{G(3)}{2}\right]\binom{x_4}{0}$$

$$= \binom{x_4}{2} - \alpha_1\binom{x_4}{1} + \alpha_2\binom{x_4}{0}$$

where

$$\alpha_1 = \binom{G(2)}{1}, \alpha_2 = \left[\binom{G(2)}{1}\binom{G(3)}{1} - \binom{G(3)}{2}\right]$$

each nested summation of the form:

$$\sum_{x_{n-1}=G(n-1)}^{x_n-1}\sum_{x_{n-2}=G(n-2)}^{x_{n-1}-1}\sum_{x_{n-3}=G(n-3)}^{x_{n-2}-1}\cdots\sum_{x_2=G(1)}^{x_2-1}\sum_{x_1=0}^{0} 1$$

when unfolded from the innermost towards the outtermost summation as demonstrated above, it is anticipated to develop into an algebraic expression of the form:

$$\binom{x_n}{n-2} - \alpha_1\binom{x_n}{n-3} - \alpha_2\binom{x_n}{n-4} - \alpha_3\binom{x_n}{n-5}\cdots - \alpha_{n-2}\binom{x_n}{0}$$

As we showed before, this expresses the size (in number of branches) of a search tree where x_n has been set to a certain value. We can use this result to derive the worst case complexity searching through all branches of one subtree $[x_n][\bullet]$:

$$O\left(\binom{x_n}{n-2} - \alpha_1\binom{x_n}{n-3} - \alpha_2\binom{x_n}{n-4} - \alpha_3\binom{x_n}{n-5}\cdots - \alpha_{n-2}\binom{x_n}{0}\right) \in O\left(\binom{x_n}{n-2}\right)$$

As we showed before, in the worst case we would have to visit each and every branch of all subtrees $[G(n)][\bullet], [G(n)+1][\bullet], \cdots, [T(n)][\bullet]$, where $T(n)$ denotes the maximum possible length of an OGR with n marks (see Chapter 1).

Finally, we are able to express the overall worst case complexity of searching each and every branch of each subtree, as the following function of n:

$$O\left(\binom{G(n)}{n-2} + \binom{G(n)+1}{n-2} + \cdots\binom{T(n)+1}{n-2}\right)$$

To be able to convert a nested summation of as in Eq. 12.1, we have to be able to calculate the coefficients $\alpha_1, \cdots, a_{n-2}$.

let us now perform a step-wise investigation of the nested summation unfolding procedure we demonstrated.

(1) we start off with an expression of the form:

$$\binom{x_2}{0}$$

(2) we intergrate over $\sum_{x_2=G(2)}^{x_3-1}$ to get:

$$\binom{x_3}{1} - \binom{G(2)}{1}\binom{x_3}{0}$$

(3) we intergrate over $\sum_{x_3=G(3)}^{x_4-1}$ to get:

$$\binom{x_4}{2} - \binom{G(2)}{1}\binom{x_4}{1} - \left[\binom{G(3)}{2} + \binom{G(2)}{1}\binom{G(3)}{1}\right]\binom{x_4}{0}$$

and so on.

At this point we are able to derive a pattern on how the next expression is produced by integrating the current one. Each term of the form

$$\alpha \binom{x_k}{\lambda}$$

when integrated over $\sum_{x_k=G(k)}^{x_{k+1}-1}$, expands to:

$$\alpha \binom{x_{k+1}}{\lambda+1} - \alpha \binom{G(k)}{\lambda+1}$$

producing a term which is a function of x_{k+1} and a constant term.

Based on this observation we can derive the general rule displayed in Figure 12.1 for producing the next expression by integrating the current one.

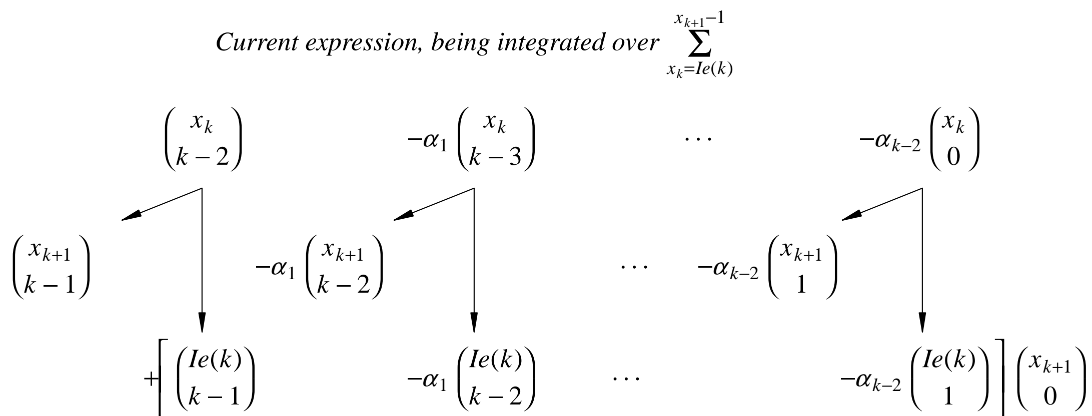

Figure 12.1. Schematic representation of a general rule for producing the next expression, by integrating the current one over $\sum_{x_k=G(k)}^{x_{k+1}-1}$.

BIBLIOGRAPHY

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Cliff Stein, *Introduction to Algorithms,* McGraw-Hill (1990).
2. Gurari Eitan, “Backtracking algorithms "CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms",” Ohio-State University lectures (1999).
3. Stephen W. Soliday, Abdollah Homaifar, and Gary L. Lebby, “Genetic Algorithm Approach To The Search For Golomb Rulers,” *6th International Conference on Genetic Algorithms (ICGA’95)*, pp. 528-535 Morgan Kaufmann, (1995).
4. Naouel Ayari, The Van Luong, and Abderrazak Jemai, “A hybrid genetic algorithm for the Golomb ruler problem,” *Computer Systems and Applications, ACS/IEEE International Conference on*, pp. 1-4 IEEE Computer Society, (2010).
5. Vazirani Vijay V and W. C. Babcock, “Intermodulation Interference in Radio Systems,” *Bell System Technical Journal*, pp. 63-73 Springer, (1953).
6. George B. Dantzig and Mukund N. Thapa, *Linear programming 1: Introduction,* Springer-Verlag (1997).
7. Meyer, Christophe and Jaumard, Brigitte, “Equivalence of some LP-based lower bounds for the Golomb ruler problem,” *Discrete Appl. Math.* **154**(1) pp. 120-144 Elsevier Science Publishers B. V., (January 2006).
8. Martin Gardner, “Mathematical Games,” pp. 108-112 in *Scientific American*, (March 1972).
9. Justin Colannino, “Circular and Modular Golomb Rulers,” Available online: http://cgm.cs.mcgill.ca/~athens/cs507/Projects/2003/JustinColannino/ (unknown date).
10. Apostolos Dimitromanolakis, *Analysis Of The Golomb Ruler And The Sidon Set Problems And Determination Of Large Near-Optimal Golomb Rulers,* Technical University of Crete (June 2002).
11. Grimaldi Ralph, *Discrete and Combinatorial Mathematics: An Applied Introduction,* (no listed publisher) (1998).
12. W. T. Rankin, “Optimal Golomb Rulers: An Exhaustive Parallel Search Implementation,” MSc Thesis, Duke University (1993).
13. E. Sotiriades, A. Dolals, and P. Athanas, “Hardware - Software Codesign and Parallel Implementation of a Golomb Ruler Derivation Engine,” *Proceedings, Field Programmable Custom Computing Machines*, pp. 227-235 (2000).
14. Pacheco, Peter S., *Parallel Programming with MPI,* Morgan Kaufmann (1997).
15. P. Malakonakis, “Study, design and FPGA implementation of an architecture for the parallel computation of Optimal Golomb Rulers (GE3),” Bachelor’s Thesis, Technical University of Crete (2009).
16. Bentley, Jon L. and Kernighan, Brian W., “A System for Algorithm Animation Tutorial and User Manual,” CSTR-132, Bell Laboratories (1987).
17. Claudia Leopold, *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches,* Wiley-Interscience (November 17, 2000).

18. Flynn, M, “Some Computer Organizations and Their Effectiveness,” *IEEE Trans. Comput*, p. 948 (1972).

19. Jason Sanders and Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming,* Addison-Wesley (2010).

20. Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis, “Gnort: High Performance Network Intrusion Detection Using Graphics Processors,” *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, (2008).

TABLE OF CONTENTS

Chapter 1 — Introduction 1
Chapter 2 — Golomb Rulers 3
Chapter 3 — Related Work 7
Chapter 4 — Backtracking Search 9
Chapter 5 — Search Space Partitioning 21
Chapter 6 — Parallel and Distributed Computing 28
Chapter 7 — Nvidia CUDA 32
Chapter 8 — Parallel Search with CUDA 39
Chapter 9 — Parallel Search With the T.U.C. Grid 48
Chapter 10 — Future Work 53
Chapter 11 — Conclusion 55
Appendix A — Backtracking Search Complexity 59
References 64

ACKNOWLEDGMENTS

I would like to thank my supervisor, Ioannis Papaefstathiou, for his patience and his understanding. I would also like to thank my family for their trust in me and their support.