Novel Parallel Algorithms for a Class of Deterministic Linear Optimization Problems

A Dissertation

Presented to

the Faculty of the Department of Industrial Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in Industrial Engineering

by

Likang Ma

August 2014

Novel Parallel Algorithms for a Class of Deterministic Linear Optimization Problems

Likang Ma

Approved:

Chairman of the Committee
Gino Lim, Associate Professor,
Industrial Engineering

Committee Members:

Qianmei Feng, Associate Professor,
Industrial Engineering

Jiming Peng, Assistant Professor,
Industrial Engineering

Edgar Gabriel, Associate Professor,
Computer Science

S. Lennart Johnsson, Professor,
Computer Science

Suresh K. Khator, Associate Dean,
Cullen College of Engineering

Gino Lim, Associate Professor and
Chairman, Industrial Engineering

Novel Parallel Algorithms for a Class of Deterministic Linear Optimization Problems

An Abstract

of a

Dissertation

Presented to

the Faculty of the Department of Industrial Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in Industrial Engineering

by

Likang Ma

August 2014

Abstract

The p-median problem and Intensity-Modulated Radiation Therapy (IMRT) treatment planning problems are very important practical applications in the area of optimization. Real-life instances of both problems are time-consuming to solve using traditional solution techniques. However, both problems can be involved in time-sensitive decision-making processes, in which rapid and accurate solutions are required. This study explores parallel computational algorithms and implementations for these two discrete optimization problems. Specifically, we address the use of Graphics Processing Unit (GPU) and Central Processing Unit (CPU) based algorithms that are specific to the needs of real-life applications.

The p-median problem is often used to model many real-world situations, which is NP-hard. Although the polynomial algorithm is available when the number of median is fixed, large scale p-median problems are still very difficult to solve. Previous studies in using a GPU to solve the p-median problem in parallel are limited. We propose the design and implementation of the parallel Vertex Substitution (pVS) algorithm for the p-median problem based on high-performance, many-core GPUs. pVS is based on the best profit search algorithm, an implementation of Vertex Substitution (VS), that is shown to produce reliable solutions for the p-median problem. Numerical experiments show pVS achieved speed gains ranging from 10x to 57x over the traditional CPU-based Vertex Substitution.

The Fluence Map Optimization (FMO) problem in IMRT can be modeled as a large-scale LP problem. Real-life FMO problem can be time-consuming to solve using traditional sequential LP solvers. We developed a GPU-based parallel linear programming solver (GPU LP solver) using the Bounded Variable Simplex algorithm with Steepest-edge pricing for large-scale sparse LP problems. This solver is designed

for general linear programming problems and can be used in branch-and-bound techniques for mixed integer programming problems. We propose a parallel explicit matrix update method to replace transformation-based matrix update in sequential simplex. A special sparse matrix format is designed so as to improve the speed of sparse column selection and parallel matrix operations. We tested our GPU-based LP solver in two FMO problems and obtained 2x speedup compared to CPLEX 12.1.

The Beam Angle Optimization (BAO) problem in IMRT is a Combinatorial Optimization Problem (COP), which is very difficult to obtain an optimal solution. Previous studies explored the theories and implementations of solution techniques for general COP problems. However, applications of such techniques to IMRT problems usually applies many approximations, which may impact the quality of the final solution. The parallelization of those techniques for BAO are also limited in the literature. We focused our research on CPU-based parallel algorithms in the applications of IMRT treatment planning using the Message Passing Interface (MPI). We developed an MPI-based Master-Worker framework for solving BAO problems using various types of algorithms including Genetic Algorithm and Simulated Annealing. The proposed framework separates integer variables from the MIP model and uses optimal LP solutions as evaluation functions. We developed a hybrid framework to communicate between algorithms in parallel. The results of numerical experiments demonstrate that this framework is 5x faster than traditional solution techniques and is able to obtain a clinic-standard treatment plan in a very short time.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Linear programming and its generalization, mathematical programming, can be viewed as part of a great revolutionary development that has given mankind the ability to state general goals and lay out a path of detailed decisions to be taken in order to "best" achieve these goals when faced with practical situations of great complexity. The tools for accomplishing this are the models that formulate real-world problems in detailed mathematical terms, the algorithms that solve the models, and the software that execute the algorithms on computers based on the mathematical theory.

– George B. Dantzig

1.1 Background

Since G. Dantzig first introduced the simplex method in 1947 [4], linear programming (LP) and mixed integer programming (MIP) have been widely studied in the field of optimization. The ability to solve real-life problems became one of the most important research objectives for many years. Hence, many researchers have focused on developing computationally efficient solution algorithms for solving real-world problems. In recent decades, algorithmic improvements have provided solid conceptual techniques for LP and MIP problems [5]. However, solving large-scale LP

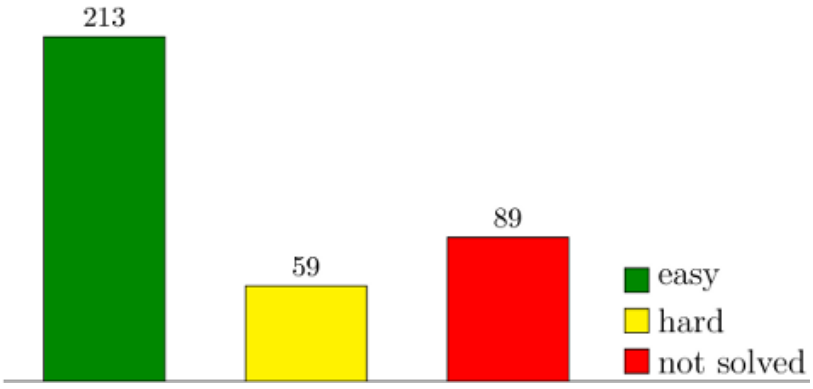


Figure 1.1: Solvability of the instances in MIPLIB2010 [1]

or MIP problems is still time-consuming. In the latest Mixed Integer Problem Library (MIPLIB 2010), a test library created to measure the performance of MIP solvers, more than 40% of the problems are regarded as difficult to solve, and around 24% of the problems cannot be solved by leading commercial or open source MIP optimizers in a reasonable time [1].

In recent years, parallel computing has provided a new paradigm for solving large-scale problems in various areas of science and engineering. Contrary to traditional solution techniques, parallel computing delegates the pressure of massive calculations to different processing units, which execute the computing instructions simultaneously.

1.2 The Problem Descriptions

Many practically important problems in optimization require that solutions be obtained in a short period of time in order to support decision-making processes, e.g., the deployment of emergency facilities in evacuation planning using the p-median problem and cancer treatment planning in Intensity-Modulated Radiation Therapy (IMRT). Real-life instances of such problems are usually very difficult to solve using traditional sequential algorithms due to problem size and complexity. However, rapid and accurate solutions are necessary to make better decisions. This study explores the

parallel computational algorithms designed for two discrete optimization problems, i.e., the p-median problem and the IMRT treatment planning problem.

1.2.1 p-Median Problem

The p-median problem concentrates on selecting the location of facilities on a network and allocating demand points to those facilities. The p-median problem is well-studied in the field of discrete location theory, which includes the p-median problem, p-center problem, uncapacitated facility location problem (UFLP), and the quadratic assignment problem (QAP) [6], to name a few.

The objective of the p-median problem is to select the location of facilities on a network, so that the sum of the weighted distances from all demand points to their nearest facility is minimized; this is also classified as the mini-sum location-allocation problem. The original problem can be dated back to the 17^{th} century, when Pierre de Fermat discussed the method to find the median point in a triangle on a two-dimensional plane, which minimizes the sum of the distances from each corner point of that triangle to the median point. This problem was extended in the early 20^{th} century by Alfred Weber [7] and acknowledged as the first location-allocation problem.

Weber's problem [7] was introduced into a network of graph theory from the Euclidean plane in the early 1960s by Hakimi [8, 9]. Hakimi later proved that the optimal median is always located at a vertex of the graph, albeit this point is allowed to lie along the graph's edge. This result provided a discrete representation of a continuous problem.

Let $(\mathbb{V}, \mathbb{E})$ be a connected, undirected network with vertex set $\mathbb{V} = \{v_1, \ldots, v_n\}$ and nonnegative vertex and edge weights w_i and l_{ij}, respectively. Further, let $\gamma(v_i, v_j)$ be the length (distance) of the shortest path between vertices v_i and v_j with respect to the edge weights w_i, and $d(v_i, v_j) = w_i\gamma(v_i, v_j)$ be the weighted length (distance) of the corresponding shortest path. Notice d is not a metric since w_i is not necessarily

w_j, $i \neq j$. This notation is extended so that for any set of vertices X, we have $\gamma(v_i, X) = \min\{\gamma(v_i, x) : x \in X\}$ and $d(v_i, X) = \min\{d(v_i, x) : x \in X\}$. With this notation, the p-median problem can be expressed as

$$\min \left\{ \sum_{v_i \in \mathbb{V}} d(v_i, X) : X \subseteq \mathbb{V},\ |X| = p \right\}. \tag{1.1}$$

The p-median problem is very important to model many real-world situations such as the location of public or industry facilities and warehouses [10, 11, 12]. However, numerous instances of location problems have proven to be too large for an exact solution to be found in a reasonable time [13]. The p-median problem in general has been proven to be NP-hard by Kariv and Hakimi [14]. Although a polynomial algorithm is available where p is fixed [15], large-scale real-world problems usually cannot be solved. As a result, heuristic algorithms that are capable of providing approximate solutions in a reasonable time are often applied on the p-median problem in order to reduce computation time.

1.2.2 Intensity-Modulated Radiation Therapy (IMRT) Treatment Planning Problem

Cancer is one of the most significant health problems worldwide and is the second leading cause of death in the United States, exceeded only by heart disease. A total of 1,665,540 new cancer cases and 585.720 deaths from cancer are projected to occur in the United States in 2014 [16]. There are many types of cancer treatments, which are determined by the cancer type and stage. The most common cancer treatments are surgery, chemotherapy, and radiation therapy.

IMRT is an advanced mode of high-precision radiation therapy which has been rapidly adopted by radiation oncologists to treat patients with specific types of cancer in United States [17]. IMRT has the potential to enhance the target tumor conformity

Figure 1.2: Linear accelerators in intensity-modulated radiation therapy [2].

and/or normal tissue sparing when compared with other treatment techniques [18]. IMRT utilizes linear accelerators (LINAC) to deliver precise radiation doses to a malignant tumor or specific areas within the tumor. The LINAC can rotate 360 degrees in a perpendicular plane when the patient is lying on the treatment couch, as shown in Figure 1.2. The LINAC stops at one angle to deliver multiple shapes of radiation beams and then moves on to the next angle. The multi-leaf collimator (MLC) is used to shape the beam to conform to the tumor's shape by moving its leaves back and forth to block a portion of each beam's radiation dose. As a result, many two-dimensional beam shapes can be constructed to perform intensity modulation. The MLC divides its aperture into rectangular grids so as to partition beams into sub-beams, called beamlets or pencil beams.

In contrast to conventional 3D-CRT, which relies on the forward planning, IMRT treatment plans are often created with inverse planning. In inverse planning, the physician defines the target volume on computed tomography (CT) or magnetic resonance imaging (MRI). After the desired dose to the tumor and the constraints for the surrounding normal structures provided by the treatment planner, a series of optimization techniques are applied to calculate non-uniform intensities that enhance the

conformity of the dose delivered to the tumor while sparing the surrounding critical and normal tissue. Typically, three problems are involved in IMRT treatment planning optimization: Fluence Map Optimization (FMO), Beam Angle Optimization (BAO), and Beam Segmentation (BS).

The BAO problem is to find a subset from a given candidate beam angle set to deliver treatment. Given a fixed beam angle set, the FMO problem is to optimize the beam intensity map for each beam angle. Once the beam intensity map is obtained, the BS problem converts the intensity map to MLC leaf sequences for dose delivery because the optimized beam intensity maps contain non-uniform dose for the beamlets. We focus on the FMO and BAO problem in our research, as a substantial speed gain can be obtained using a parallel computation framework.

The Fluence Map Optimization specifies the intensity of radiation doses of each beamlet within a beam. FMO solves for an optimal intensity map for each beam angle. Intensity of beamlet refers to the intensity of radiation delivered from one beam angle through that beamlet. There are three main target volumes in IMRT treatment planning: gross tumor volume (GTV), clinical target volume (CTV), and planning target volume (PTV) [19]. The GTV is the gross tumor area that can be imaged using an MRI or CT scan. The CTV applies a margin for microscopic disease spread which cannot be fully imaged. The PTV adds a margin around CTV so as to allow organ movements and clinical uncertainties in planning or treatment delivery. The organs at risk (OARs) are important organs close to the PTV, and all remaining portions are considered normal tissues.

There are usually two types of objective function in FMO: dose-based models and biological models. Dose-based models only consider radiation dose, while biological models also calculate the biological effects of dose distribution. Although many

research efforts are underway, biological models have not been widely used in practical optimization [20]. In this research, we focus on the dose-based model of IMRT. FMO can also be represented in both linear [21, 22, 23] and non-linear programming models [24, 25, 26]. It is found that the FMO problem is highly degenerate [27, 28]. Many existing nonlinear models cannot not guarantee optimality, and the result of the optimization will depend on the starting conditions [29, 30, 31, 32]. We use a linear programming model, which can guarantee the optimal solution and has full control over constraints.

Beam Angle Optimization determines treatment angles used by the LINAC to deliver the radiation dose. A well-designed FMO algorithm may result in a suboptimal treatment plan if the angles are not carefully selected. Therefore, BAO is a key component in the process of IMRT treatment planning, and it can greatly affect treatment quality [33, 34]. The BAO problem is a combinatorial optimization problem. The BAO solves for the optimal combination of a fixed number of treatment beam angles among all feasible orientations along the 360° circumference. Traditional exact solution techniques, such as branch and bound, typically do not produce good feasible solutions within a reasonable time due to the size and complexity of this problem.

1.3 Overview of Parallel Computing

Traditionally, software or algorithms are designed for serial computation which involves a discrete series of instructions to be executed on a single computer having a single Central Processing Unit (CPU). Those instructions are executed in certain order, and only one instruction is processed at a time. In parallel computing, a problem is broken into parts that can be solved simultaneously using multiple computing

resources. Those computing resources can be a single computer with multiple processors, a group of computers connected in a network, computing cores on a model Graphics Processing Unit (GPU), or a combination of the above approaches. The development of parallel computing is largely driven by the limitations of traditional serial computing, which faces physical and practical challenges in maintaining continuous growth in computing capacity [35]. Parallel computing provides methodologies and technologies to speed up the solution process of large problems which were, in practice, impossible to process using serial computing.

1.3.1 Message Passing Interface

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the MPI is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility.

MPI primarily addresses the message passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process. Originally, MPI was designed for distributed memory architectures. As architecture trends changed, shared memory computers were combined over networks, creating hybrid distributed memory and shared memory systems. MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also developed ways of handling different interconnections and protocols. Today, MPI runs on virtually any hardware platforms including Distributed Memory, Shared Memory, or Hybrid. Nevertheless,

the programming model clearly remains a distributed memory model regardless of the underlying physical architecture of the machine.

1.3.2 General-Purpose Computation on GPU

A Graphics Processing Unit is a microprocessor which was first designed to render and accelerate 2D and 3D image processing on computers. Although GPUs have been available for most desktop computers since the 1990s, they were once very difficult to program and were rarely used beyond graphics processing.

It was not until the early 2000s that the high-performance, many-core architecture of the GPU and its capability to handle very high computation and data throughput became available for programming and scientific computations. The applications of GPU parallel computing have been introduced into different areas of science and engineering [36, 37, 38, 39, 40]. Most of those applications achieved significant speedups when compared with traditional CPU-based algorithms. A GPU's many-core processors and high-bandwidth memory are extremely suitable for computation-intensive and data-parallel computations.

Figure 1.3 illustrates that desktop GPUs have exceeded CPUs in float point operation tests since 2003, and the gap between GPUs and CPUs has kept growing. Figure 1.4 shows the same trend in a memory bandwidth test, which is often the bottleneck in scientific computations.

Programming interfaces, industry-standard languages, and developing tools are very important for general-purpose computing on GPU. There are several GPU computing architectures currently available to the public, including NVIDIA CUDA, Open CL, and Direct Computing. Currently, NVIDIA CUDA is the mainstream framework for general-purpose computing on GPUs. Our research is based on the CUDA architecture.

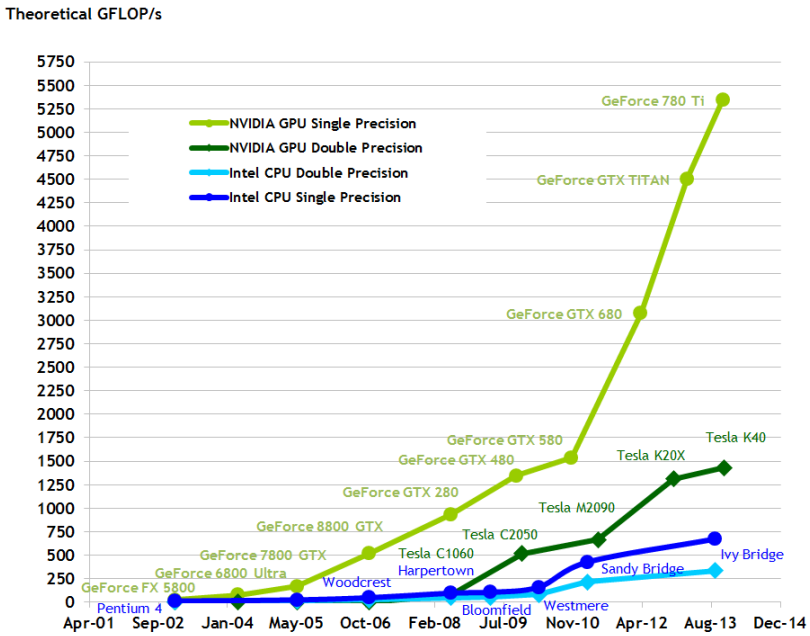


Figure 1.3: Float point operation capacity comparison between GPUs and CPUs [3].

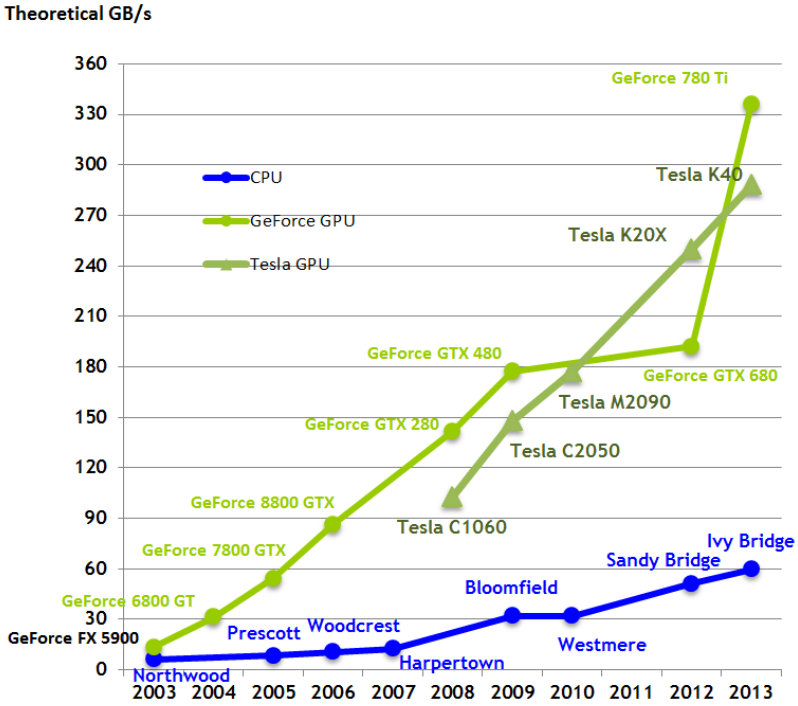


Figure 1.4: Memory bandwidth comparison between GPUs and CPUs [3].

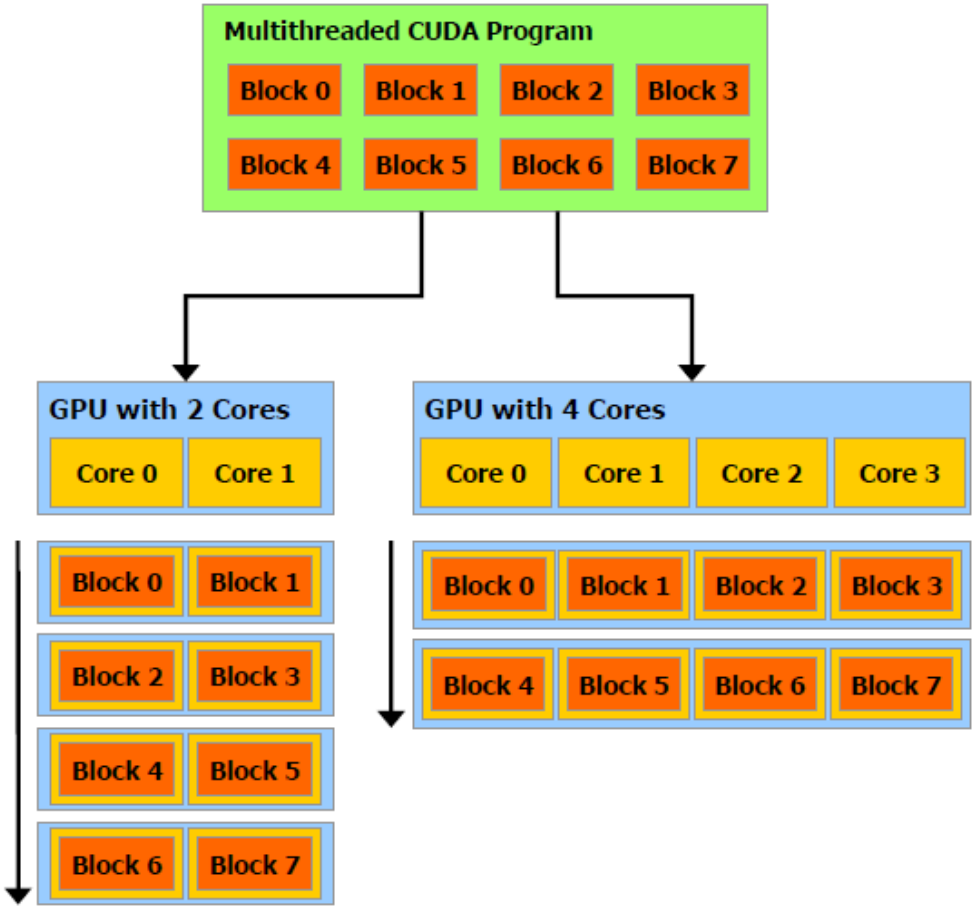


Figure 1.5: CUDA scalability and abstractions [3].

NVIDIA CUDA [3] was first introduced by NVIDIA in November 2006 as a general-purpose parallel computing architecture. This architecture provides parallel programing model and instruction set, which enables the programmers to directly access NVIDIA's GPUs. CUDA's environment allows developers to use high-level programming languages, such as CUDA C and CUDA Fortran. Three key abstractions, namely a hierarchy of thread groups, memory, and barrier synchronizations, are introduced by CUDA to provide scalability in applications. Figure 1.5 shows how CUDA automatically distributes same application to different pieces of hardware.

CUDA *threads* are the smallest hardware abstraction that CUDA provides to control the GPUs multiprocessors. A *block* is a group of threads which are expected

to run on the same processor core. *Blocks* are organized into *grids*. The number of *blocks* in a *grid* is usually determined by the total number of threads or the size of the data. CUDA *kernel* is a group of logic statements which is similar to a function in the C programming language. However, A CUDA *kernel* can be executed multiple times by multiple different CUDA threads simultaneously.

There are three levels of memory structure in CUDA architecture: *local*, *shared*, and *global* memory. CUDA threads can access data from different memory spaces. Each *thread* has private *local* memory which can be accessed by the corresponding *thread* only. Within each *block*, *shared* memory is accessible by all *threads* in that *block* and the data in the *shared* memory has the same lifetime as the *block*. *Global* memory is accessible by all *threads* during the program's lifetime. *Shared* memory and *local* memory have ultra-fast bandwidth and the lowest latency, but the size is very limited.

1.4 Contributions

In this research, we present the design and implementation of parallel algorithms for two real-world problems: the p-median problem and the IMRT treatment planning problems. Both of these large-scale real-life instances posed a severe challenge to obtaining solutions. Some test cases took days, even weeks, to obtain a solution using traditional solution techniques. We developed parallel computational algorithms using both CPUs' and GPUs' parallel computing platforms.

For the p-median problem, a GPU-based parallel Vertex Substitution (pVS) algorithm is proposed and the results are compared with a similar CPU-based sequential algorithm. We achieved speed gains ranging from 10x to 57x over the traditional VS in all test network instances. pVS also reduces worst-case complexity from sequential VS's $O(n^3)$ to $O(p \cdot (n-p))$ on each thread by parallelizing computational tasks in GPU implementation.

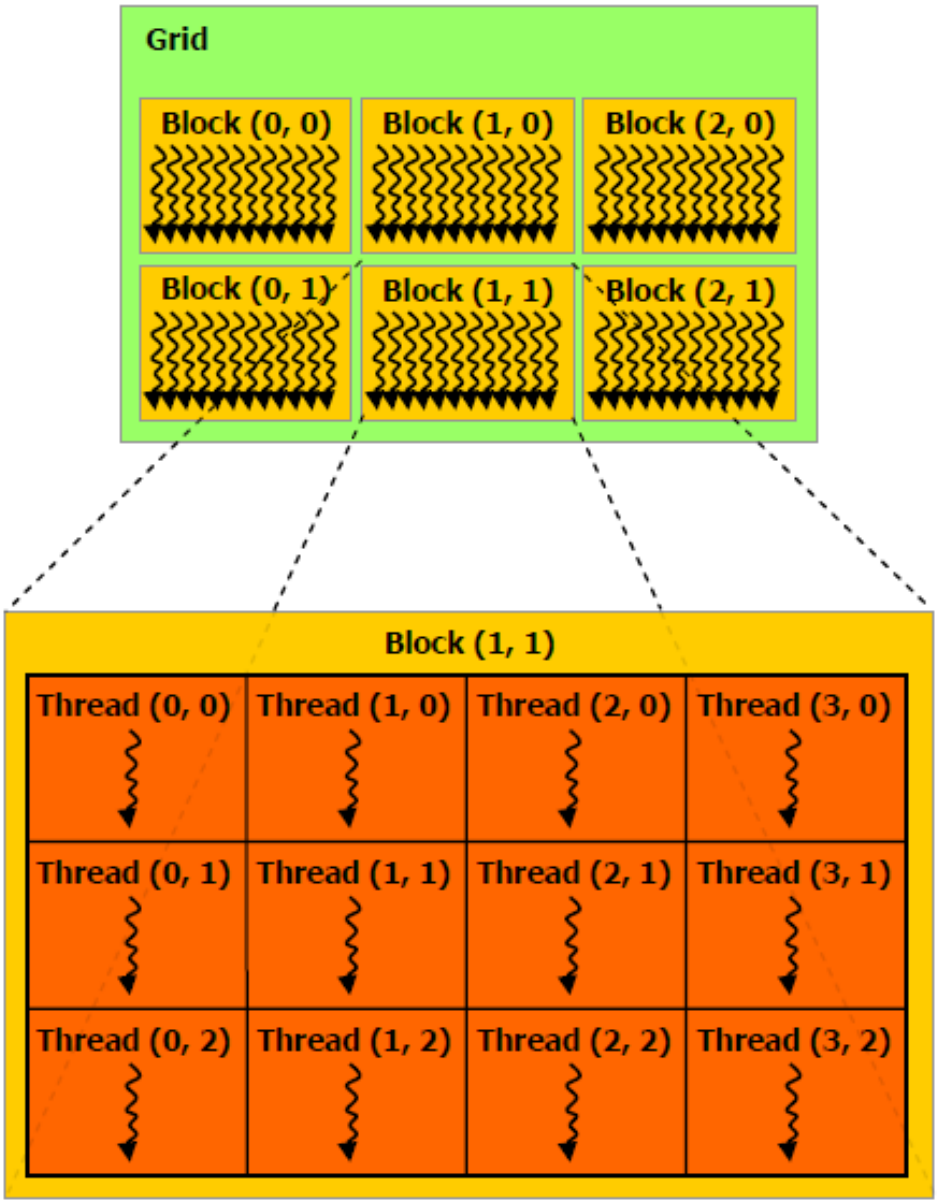


Figure 1.6: CUDA threads, blocks, and grids [3].

For the FMO problem in IMRT treatment planning, we developed a GPU-based parallel linear programming solver (GPU LP solver). This bounded variable simplex solver is designed for solving general linear programming problems. We tested this solver on two FMO problems. The solution quality obtained by GPU LP solver were as good as the results of CPU-based sequential algorithms and the solution times for the GPU-based parallel LP solver were twice as fast as thsse of CPU-based algorithms.

As an effort to solve the BAO problem, we developed a MPI-based Master-Worker Framework with hybrid algorithms including parallel Genetic Algorithm and parallel Simulated Annealing. This framework achieved at least 5x-12x speedups compared to the traditional solution approaches. The framework is able to reach clinic standard treatment planning within a short time, while the MIP solution failed to converge in more than a week.

1.4.1 List of Publications

1.4.1.1 Journal

- G.J. Lim and L. Ma, GPU-based Parallel Vertex Substitution Algorithm for the p-median Problem, *Computers and Industrial Engineering*, June 2012.

- L. Ma and G.J. Lim, A GPU-based Parallel Linear Programming Solver and Its Application to the Fluence Map Optimization Problem in Radiation Therapy Planning, submitted to *Annals of Operations Research*.

1.4.1.2 Conference Proceedings

- L. Ma and G.J. Lim, GPU-based Parallel Algorithm for IMRT Beam Angle Optimization, *In Proceedings of the 2011 Industrial Engineering Research Conference*.

1.4.1.3 Presentations

- L. Ma and G.J. Lim, GPU-based Parallel Algorithm for IMRT Beam Angle Optimization, *2011 INFORMS Annual Conference*.
- L. Ma and G.J. Lim, GPU-based Bounded Variable Simplex Method for Solving Fluence Map Optimization Problem, *2012 INFORMS Annual Conference*.
- L. Ma and G.J. Lim, MPI-based Parallel Genetic Algorithms for Radiation Treatment Planning, *2013 INFORMS Annual Conference*.

1.5 Dissertation Overview

This dissertation is organized into six chapters as follows. In Chapter 2, we briefly review the literature of the p-median problem and parallel solution techniques. Existing parallel algorithms for IMRT treatment planning problems are introduced. In Chapter 3, we describe the Vertex Substitution on CPU followed by the design and implementation of GPU-based parallel algorithms. Numerical results are discussed in the last part of Chapter 3. In Chapter 4, we explain the details of the GPU-based linear programming solver and its application in the FMO problem. In Chapter 5, we present a MPI-based Master-Worker framework with parallel Genetic Algorithm, parallel Simulated Annealing, and the hybrid approaches for solving Beam Angle Optimization problems. In Chapter 6, we conclude the dissertation with a summary of our contributions and the direction of future research.

Chapter 2

Literature Review

2.1 p-median Problem

The p-median problem is well-studied and widely applied in operation research, machine learning, and graph theory. Numerous solution methods for the p-median problem have been discussed in literature. These methods can be categorized into several types [41]. The major types are: Heuristic, such as Vertex Substitution and variable neighborhood search, LP relaxation, IP formulations reductions, approximation algorithms, and enumeration.

Since solving large scale p-median problems is time-consuming, heuristic methods are always popular topics in this area. In 1963, Kuehn [42] introduced a greedy method which selects potential locations that maximize the saving gain from replacing those initially chosen locations. Maranzana [43] presented a method which divides the network and calculates the center of gravity for each partition. Neebe and Rao [44] proposed one sub-gradient method to solve the dual of the relaxed LP model for p-median problem. More recently, Hansen and Mladenovic [45] applied the variable neighborhood search (VNS) method, which is a meta-heuristic that systematically changes the neighborhood within a local search algorithm. Genetic Algorithms (GA) has become popular in recent years, e.g., Erkut and Drezner's infeasible initialization method [46].

One important and most common solution technique for the p-median problem is the vertex substitution (VS) method, which was first developed by Teitz and Bart [47]. The basic idea of vertex substitution is to find one vertex which is not in the solution set and replace it with one vertex in the solution set until further switching cannot improve the objective. Many different versions of the vertex substitution method have appeared after Teitz and Bart's work [47]. Those variants use different rules in choosing vertex switch pairs. *Lim, et al* [48] showed that VS can usually return a stable and robust solution for the p-median problem. However, it typically takes longer to converge than other heuristic algorithms, such as Discrete Lloyd's Algorithm [49].

Recently, CPU-based parallel implementations have been reported in literature. López [50] introduced a parallel variable neighborhood search algorithm. Crainic [51] improved López's method by adding a cooperative feature among parallel computing jobs. López [52] also proposed a parallel scatter search algorithm. However, those implementations usually achieve speedups of less than 10x. Additionally, it is still time-consuming to solve problems with more than 1000 vertices and p larger than 100. GPU-based parallel algorithms for the p-median problem are rarely reported in the literature.

This dissertation presents a parallel Vertex Substitution (pVS) algorithm and its implementation on a GPU with the NVIDIA CUDA framework. This GPU-based pVS is able to solve the p-median problem for very large networks with more than 5000 nodes. The numerical experiments show the proposed pVS obtained a peak 57x speedup compared to CPU-based VS.

2.2 GPU-based Parallel Solution Techniques for FMO Problem

Solving large-scale optimization problems in IMRT treatment planning is time-consuming with CPU-based sequential algorithms. Recently, GPU-based algorithms became popular in IMRT research because the potential computation speed can overcome the computational challenge in treatment planning. GPU-based dose calculation algorithms have been developed to meet the massive data-calculation demand in treatment planning. [40, 53, 54]. For the FMO problem, Men [55] developed a GPU-based IMRT planning framework. Their work implemented a gradient projection method with Armijo line search rule and concentrated on fluence map re-optimization during the treatment plan based on a non-linear optimization model. However, there are few parallel methods for solving FMO problems with the linear programming model.

Recent literature review [56] shows that there is no CPU-based parallel simplex implementation reported that outperforms sophisticated sequential implementation. Early works involving the CPU-based parallel simplex method were investigated by Zenios [57]. More recently, Eckstein *et, al* [58] developed a parallel standard simplex with dense matrix representation. Shu [59] explored the performance of the revised simplex method with sparse matrix representation. Both Eckstein and Shu's work can be categorized as data parallelism. Task parallelism was introduced into simplex by Hall and Mckinnon [60], and Bixby and Martin [61]. However, as concluded by Hall's survey [56], most existing parallel simplex methods and implementations are CPU-based and have very limited advantages compared to their CPU-based sequential counterpart.

As for GPU-based parallel simplex implementations, Spampinato presented a CUDA-based simplex method in 2009 [62] which achieved 2x speedup in comparison with BLAS-based CPU implementation. Bieling [63] implemented the revised

simplex method which obtains 18x speedup compared to the GLPK simplex solver. Lalami [64] reported 12x speedup on a problem with 4000 variables and constraints when compared to a similar CPU-based C++ implementation. Meyer [65] proposed a revised simplex with steepest-edge pricing on a multiple-GPUs cluster and observed 100x speedup compared to the COIN-OR CLP solver.

Though GPU-based simplex implementations have been frequently reported in recent years, most of these implementations have limitations in problem size. A notably misleading message on some of these methods is that their performance comparisons are made against self-implemented or open source solvers rather than leading commercial solvers such as CPLEX and GUROBI. These commercial solvers solved the majority of the test problems much faster than the leading open source LP solvers [1].

This dissertation introduces a GPU-based parallel linear programming solver (GPU LP solver) using the Bounded Variable Simplex algorithm with the steepest-edge pricing method. The implementation of this GPU LP solver focuses on the ability to handle real-life sparse problems, e.g., the FMO problem in IMRT treatment planning. We propose a special sparse matrix format and parallel explicit matrix update method to improve the speed of sparse column selection and parallel matrix operations in Simplex.

2.3 Solution Techniques for Combinatorial Optimization and Beam Angle Optimization

Many practically important problems can be categorized as Combinatorial Optimization problems (COP), including the Travel Salesman problem, Quadratic Assignment problem, and Minimum Spinning Tree problem. Beam Angle Optimization is also a type of COP problem.

A typical COP is described as follow. Given a finite set S of vectors in $\mathbf{R}^n$ and objective function $f(x)$, find

$$\begin{aligned} \min \quad & f(x|x \in S) \\ s.t. \quad & A \cdot x \ \leq \ c \end{aligned} \tag{2.1}$$

For large-scale problems, S is often huge and exponential in n. Though polynomial algorithms are available for some specific problems, general COPs are NP. For large problems, exact solution methods or semi-enumerate algorithms are usually computationally intractable for COP.

Heuristic algorithms have been a rapidly growing research area in combinatorial optimization. Simulated annealing (SA) was first introduced by Kirkpatrick [66] to solve COPs. With the explicit strategy to escape from local optima, SA has been successfully applied to several COPs [67, 68]. However, stand-alone SA can be unstable in solution quality as well as time-consuming. Variable Neighborhood Search techniques were presented by Hansen and Mladenović [69]. VNS is a general algorithm that can be implemented to solve various COPs [70], and it usually guarantees a local optimal solution. Other heuristic algorithms such as Tabu search and the Ant Colony algorithm have also proved to be valuable tools in solving COPs. More recently, successful GA implementations have been explored to solve mixed integer programming and combinatorial optimization problems [71]. Parallel GA implementations [72, 73] and parallel SA [74]have been explored separately in the literature.

Sequential GA [75, 76] and SA [33, 77] have been applied to solve the BAO problem. Hybrid approaches [76, 78] are also reported in the literature. Most of the mentioned approaches applied a sequential combination of two algorithms, where one algorithm solves for beam angles and the other algorithm approximates the beam intensities. These approaches involve many approximations in solving subproblems as well as the relaxation of integer constraints, which may lead to inefficient searching

directions.

This dissertation proposes a Global Parallel Genetic Algorithm, a parallel Simulated Annealing Algorithm, and a hybrid approach for solving the Beam Angle Optimization problem on the MPI-based Master-Worker framework. We designed this framework to handle a type of Combinatorial Optimization problems, where integer variables can be separated from the MIP model and linear programming subproblems can be generated using any feasible combination of integer variables. This hybrid framework focused on real-time communication between parallel algorithms so as to improve the overall performance. By using the optimal LP solution as the evaluation test, the framework ensures that the searching direction is accurately contributed by each evaluation. This framework also has the potential to integrate extra heuristic algorithms in its operation.

Chapter 3

GPU-based Parallel Vertex Substitution Algorithm for the p-Median Problem

3.1 Integer Programming Model of the p-Median Problem

Typically, the p-median problem can be formulated as an integer programming (IP) problem as follows. Let ξ_{ij} be the decision variable such that:

$$\xi_{ij} = \begin{cases} 1 & \text{if vertex } v_i \text{ is allocated to vertex } v_j \\ 0 & \text{otherwise,} \end{cases}$$

$$\begin{aligned} \min \quad & \sum_{ij} d(v_i, v_j)\xi_{ij} \\ \text{subject to} \quad & \sum_{j} \xi_{ij} = 1, \text{ for } i = 1, \ldots, n, \\ & \sum_{j} \xi_{jj} = p, \\ & \xi_{jj} \geq \xi_{ij}, \text{ for } i, j = 1, \ldots, n, \ i \neq j, \\ & \xi_{ij} \in \{0, 1\}. \end{aligned} \tag{3.1}$$

The objective function is to minimize the sum of distances, and constraints ensure that each vertex is allocated to one and only one element in the p subset and guarantee the number of median to be p. The p-median problem is NP-hard for general p on general

graphs and networks. If p is fixed, the p-median problem is solvable in polynomial time [41].

3.2 Vertex Substitution

We now briefly describe the Vertex Substitution algorithm to motivate this study. A pseudocode is presented in Algorithm 1, and details can be found in [48]. Let $\mathcal{S}$ be the candidate solution set and $N = V \backslash \mathcal{S}$ be the remaining vertices. We consider every v_i in N as a candidate median and insert v_i into $\mathcal{S}$ to construct a candidate solution set with $p + 1$ medians. The gain of inserting each v_i is calculated. Then, based on each candidate solution set, the loss of removing each v_r in previous solution set $\mathcal{S}$ is calculated, with $\mathcal{S}_0$ being the initial solution. As a result, a new candidate solution set with p medians is obtained. The profit $\pi(v_r, v_i)$ is defined as the gain minus loss for each pair of (v_r, v_i). A new solution set is obtained by swapping vertex v_r with another vertex v_i based on the profit evaluation approach. There are two common implementations for evaluating the profit: first-profit approach [79] and best-profit approach [80]. The first-profit approach selects (v_r, v_i) once the first positive profit is found. The best profit approach evaluates all possible profits and selects (v_r, v_i) which has the most positive profit.

Algorithm 1 Pseudocode for Resende and Werneck's implementation of vertex substitution

```
1: procedure VertexSubstitution(S_0)
2:     q ← 1
3:     S_q ← S_0
4:     repeat
5:         π*_q ← max{π(v_r, v_i) : v_r ∈ S_q, v_i ∉ S_q}
6:         if π*_q > 0 then
7:             S_{q+1} ← (S_q ∪ {v*_i})\{v*_r}, where (v*_r, v*_i) is a solution to Line 5
8:             q ← q + 1
9:     until π*_q ≤ 0
```

3.3 Design of Parallel Vertex Substitution

Our parallel Vertex Substitution method is based on Lim [48], which is a modified version of Hansen's implementation [80]. The purpose of the parallel Vertex Substitution is to fully utilize the power of a GPU's many-core architecture and to reduce the solution time required by VS without compromising solution quality. Thus, only the best-profit approach is considered in our pVS implementation. The best-profit approach requires evaluating all possible swap pairs that profit evaluations among different swap pairs are independent and can be one of the major source of task parallelism. When implementing VS with the best-profit approach, the algorithm evaluates the total distance between each vertex to its nearest median among all candidate swap pairs (v_r, v_i) and its worst case-complexity is $O(n^3)$. In order to reduce the solution time, pVS utilizes an important feature of VS, which is the independence of the candidate solution evaluation. Within each iteration, new candidate solutions are obtained by swapping two vertices (v_r in the solution set with another vertex v_i) for all possible combinations. The objective value of a new candidate solution can be evaluated separately due to the independence. The main idea of pVS is to map the evaluation job of each candidate solution to each virtual thread on the CUDA GPU. All evaluation jobs are processed on individual threads.

A pseudocode of our parallel Vertex Substitution method is presented in Algorithm 2. Parallel Vertex Substitution begins with an initial solution $\mathcal{S}_0$, which is randomly generated. A swap pair consists of a vertex $v_i \in N$ and a vertex $v_r \in \mathcal{S}$, and this pair will be assigned to one thread, $thread(v_r, v_i)$. Each $thread(v_r, v_i)$ then establishes a new solution set $(\mathcal{S}_q \cup \{v_i\})\backslash\{v_r\})$ and the objective value of each $(\mathcal{S}_q \cup \{v_i\})\backslash\{v_r\})$ is evaluated on this thread. Profit $\pi(v_r, v_i)$ is obtained by using the objective value of $\mathcal{S}_q$ minus the objective value of each $(\mathcal{S}_q \cup \{v_i\})\backslash\{v_r\})$. The best solution is found based on the most positive profit by using the parallel reduction method [81]. Parallel

Algorithm 2 Pseudocode for parallel vertex substitution

1: **procedure** ParallelVertexSubstitution($\mathcal{S}_0$)
2: $q \leftarrow 1$
3: $\mathcal{S}_q \leftarrow \mathcal{S}_0$
4: **repeat**
5: Activate $thread(v_r, v_i)$
6: $thread(v_r, v_i)$ evaluates objective value of $(\mathcal{S}_q \cup \{v_i^*\})\backslash\{v_r^*\}$
7: $\pi_q^* \leftarrow \max\{\pi(v_r, v_i) : v_r \in \mathcal{S}_q, v_i \notin \mathcal{S}_q\}$ by parallel reduction
8: **if** $\pi_q^* > 0$ **then**
9: $\mathcal{S}_{q+1} \leftarrow (\mathcal{S}_q \cup \{v_i^*\})\backslash\{v_r^*\}$, where (v_r^*, v_i^*) is a solution to Line 7
10: $q \leftarrow q + 1$
11: **until** $\pi_q^* \leq 0$

reduction is an iterative method which utilizes parallel threads to perform comparison on elements in one array to find a minimum or maximum value. For example, in maximizing reduction, each thread compares two elements and selects the larger one. Thus, in each iteration, the number of candidate maximizers will be reduced to half of its number in the previous iteration until it narrows down to the last single element.

Theorem 1. *The complexity of pVS is* $O(p * (n - p))$.

Proof. Proof of the complexity of pVS consists of three parts. The first step is to group the non-solution set N into p clusters. This requires p comparisons for each vertex, and it can be done in $O(p*(n-p))$. The next step is to evaluate the objective value of the solution candidate on each thread, which is obtained by the summation of all distances among those $n-p$ vertices in set N with $O(n-p)$ operations. Third, pVS performs parallel reduction to find the best solution among $p*(n-p)$ candidate solution sets which has a $O(\log p*(n-p))$ complexity [81]. Therefore, its worst-case complexity is $O(p * (n - p))$. □

3.4 GPU Implementation of pVS

The implementation of pVS is achieved by a GPU-CPU cooperation procedure as illustrated in Figure 3.1. The procedure begins with CPU operations which prepare the distance matrix $D = \{d(v_i, v_j) \text{ for all } v_i, v_j \in \mathbb{V}\}$ and the initial solution information $\mathcal{S}_0$. GPU then executes the main pVS procedures. The results are copied back to the CPU for checking the termination condition.

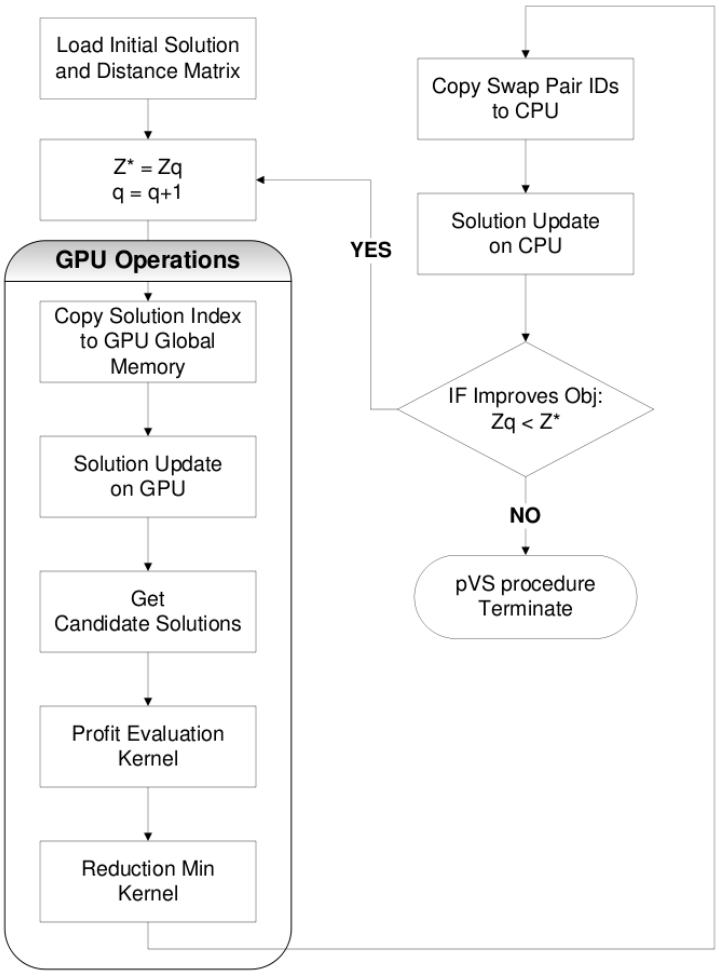


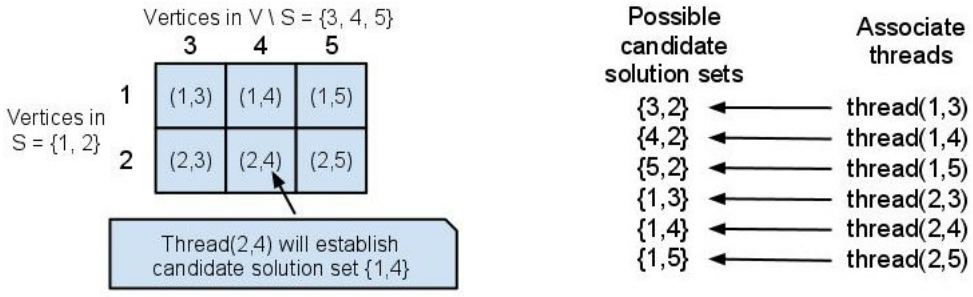
Figure 3.1: pVS GPU implementation flowchart

As an initialization step, a pre-generated random solution set $\mathcal{S}_0$ and the distance matrix D are loaded into the CPU's memory. We assume a strongly connected

network with a symmetric distance matrix. The distance matrix is compressed by only storing its upper triangular matrix in a linear memory location. This compressed distance matrix and the pre-generated random solution are then copied to the GPU's global memory.

Most of the computations are carried out by kernels in the GPU Operations module, which includes updating the solution set in GPU global memory, getting candidate solution sets, profit evaluation, and finding the best pair to swap. Once the best profitable swap pair is found in each iteration, the swap pair of vertices and new objective value are copied back to CPU memory so that the solution information gets updated. The solution update operation is performed on the CPU. This will ensure that the time to copy data between the host and the device memory will be minimized. We now explain three main GPU operations below.

Get candidate solutions: The candidate solution sets are generated by using a two-dimensional CUDA block on the GPU. pVS uses one dimension to represent the set of candidate vertices $N = V \setminus \mathcal{S}$ and another dimension to represent the current solution set $\mathcal{S}$. A thread can be identified by using the thread ID which is associated with a vertex pair (v_r, v_i). By swapping the vertices (v_r and v_i), a new candidate solution set is obtained. Since storing a new solution set $\mathcal{S}_{q+1}$ for each thread on GPU global memory is too expensive, the new solution sets are never generated explicitly in the GPU's memory. Instead, by referring to a thread's ID, each $thread(v_r, v_i)$ acquires the information on which vertex should be removed from $\mathcal{S}_q$ and which should be inserted. Therefore, a candidate solution set can be obtained without storing extra data on the GPU memory and can thus avoid consuming a large amount of memory. As an illustration, suppose we have a network with five vertices and two medians and its current solution set $\mathcal{S}$ contains vertices 1 and 2 and the remaining set N contains $\{3, 4, 5\}$. In Figure 3.2, $thread(2, 4)$ indicates that this thread will replace vertex 2 in $\mathcal{S}$ by vertex 4 in N to obtain a new candidate solution



(a) Thread configuration (b) Link between solution sets and threads

Figure 3.2: pVS representation of candidate solution set

set $\{1, 4\}$. Note that all possible candidate solution sets are established on parallel threads. Once the new candidate solutions are established, pVS moves on to evaluate the objective values of those candidate solutions using the Profit Evaluation Kernel.

Profit Evaluation Kernel: The threads and blocks structure of this kernel is inherited from above. In this kernel, $thread(v_r, v_i)$ will loop over the vertices in the entire network. For each vertex, the nearest median in set $(\mathcal{S}_q \cup \{v_i\})\backslash\{v_r\}$ and the distance between each vertex to its median is returned. Since no negative path is allowed in the network, if a vertex itself is in the set $(\mathcal{S}_q \cup \{v_i\})\backslash\{v_r\}$, it will select itself as the median with zero distance, and thus has no effect to the objective value. The total sum of distances is returned by each $thread(v_r, v_i)$ and it is the objective value for candidate solution $(\mathcal{S}_q \cup \{v_i\})\backslash\{v_r\}$. The maximum profitable (minimum objective value) pair (v_r, v_i) will be selected by Reduction Min Kernel.

Reduction Min Kernel: We now have the objective values for all possible candidate solutions on a GPU. In order to find a solution with the minimum objective value, pVS implements the parallel reduction method, which returns the minimum objective value and its solution set. The index of the resulting pair (v_r, v_i) will be sent back to the CPU so that new solution set can be updated. The loop in Figure 3.1 continues until the maximum profit is less than or equal to zero.

3.5 Computational Results

3.5.1 Experiments Setup

In this section, we benchmark the performance of pVS against its CPU counterpart. We denote pVS as the GPU-based parallel VS algorithm and VS as the CPU-based VS algorithm. The specific CPU we use is an Intel Core2 Quad Q9650 3.00 GHz CPU with 4GB RAM, while the GPU is an NVIDIA GTX 285, which has 240 cores with 1.48 GHz and 1GB onboard global memory. The Operating System is Ubuntu 10.04 and the GPU parallel computing framework is NVIDIA CUDA 3.2.

Two sets of problem instances are used to evaluate the performance of our GPU-based parallel algorithms:
(1) OR-Lib p-median test problems. These test problems were first used by Beasley [82]. The OR-Lib test set includes 40 different undirected networks. In pre-processing, we generated the shortest path for each pair of vertices using the Dijkstra algorithm [83].

(2) Randomly generated networks. These networks are generated based on the OR-lib test set. Two key network parameters (the number of nodes and the number of medians) are obtained from a network instance from the OR-Lib test set. Given these values, ten strongly connected network instances are generated by selecting random points on a two-dimensional space. Then, the distance matrix is calculated and stored. Since the test data set does not contain large network instances, we generated additional large networks that range from 1000 vertices to 5000 vertices; number of medians tested includes 10, 10% and 20% of total vertices.

3.5.2 Numerical Results

We compared the computational performance of VS and pVS on the OR-Lib networks and the results are organized in Table 3.1 and Table 3.2 by different number of medians. Additional results are also provided in Table A.1 through Table A.3 in the

Table 3.1: pVS performance on OR-lib p-median test problems $p = 5$

Problem ID	n	Edge	p	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
1	100	200	5	5718	5718	0.000	0.000	N/A
6	200	800	5	7527	7527	0.010	0.000	N/A
11	300	1800	5	7578	7578	0.040	0.000	N/A
16	400	3200	5	7829	7829	0.040	0.000	N/A
21	500	5000	5	9123	9123	0.070	0.000	N/A
26	600	7200	5	9809	9809	0.120	0.010	12.0
31	700	9800	5	10006	10006	0.140	0.010	14.0
35	800	12800	5	10306	10306	0.240	0.010	24.0
38	900	16200	5	10939	10939	0.310	0.010	31.0

Table 3.2: pVS performance on OR-lib p-median test problems $p = 10$

Problem ID	n	Edge	p	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
2	100	200	10	4069	4069	0.010	0.000	N/A
3	100	200	10	4250	4250	0.010	0.000	N/A
7	200	800	10	5490	5490	0.050	0.000	N/A
12	300	1800	10	6533	6533	0.090	0.000	N/A
17	400	3200	10	6980	6980	0.180	0.010	18.0
22	500	5000	10	8464	8464	0.370	0.020	18.5
27	600	7200	10	8257	8257	0.430	0.010	43.0
32	700	9800	10	9233	9233	0.620	0.030	20.7
36	800	12800	10	9925	9925	0.730	0.020	36.5
39	900	16200	10	9352	9352	1.120	0.040	28.0

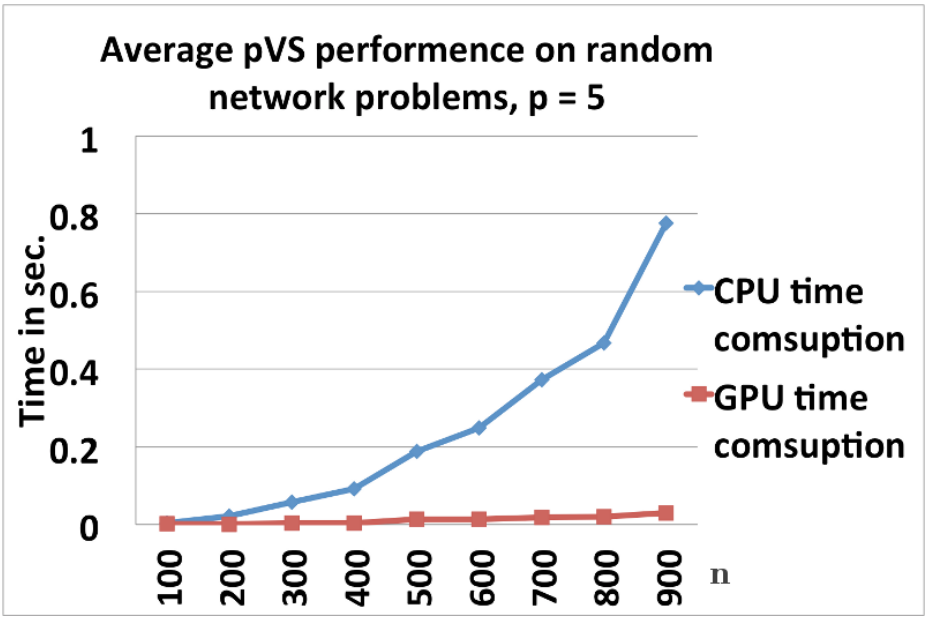


Figure 3.3: Average pVS performance on random network problems, $p = 5$

Appendix. The problem IDs are identical to the OR-lib test problems. The number of total vertices in the network is denoted as n, *Edge* is the number of edges in the network, and p stands for the number of medians. Objective value is the summation of total distances returned by each algorithm. Solution time records CPU and GPU computation time in seconds. Speedup is calculated as CPU time divided by GPU time, $speedup = \frac{CPUtime}{GPUtime}$ and N/A in the speedup column means either time is too short to measure or the iterations did not terminate after five hours. pVS obtained same objective values as VS for all problem instances, which implies that pVS can obtain the same solution quality as VS. Furthermore, pVS outperforms VS in solution time on all problem instances. Table 3.1 and Table 3.2 show that, when the number of medians is small, pVS works 10x to 40x faster than VS. As the number of medians increases to a larger percentage of the total number of vertices, pVS takes greater advantage of its parallel design and gains substantial speedups as seen in Tables A.1, A.2, and A.3. Table A.4 through Table A.8 in the Appendix show the average time consumption over 10 network instances in each size (combinations of number of n and p). Figure 3.3 through Figure 3.7 illustrate the time consumption for both pVS

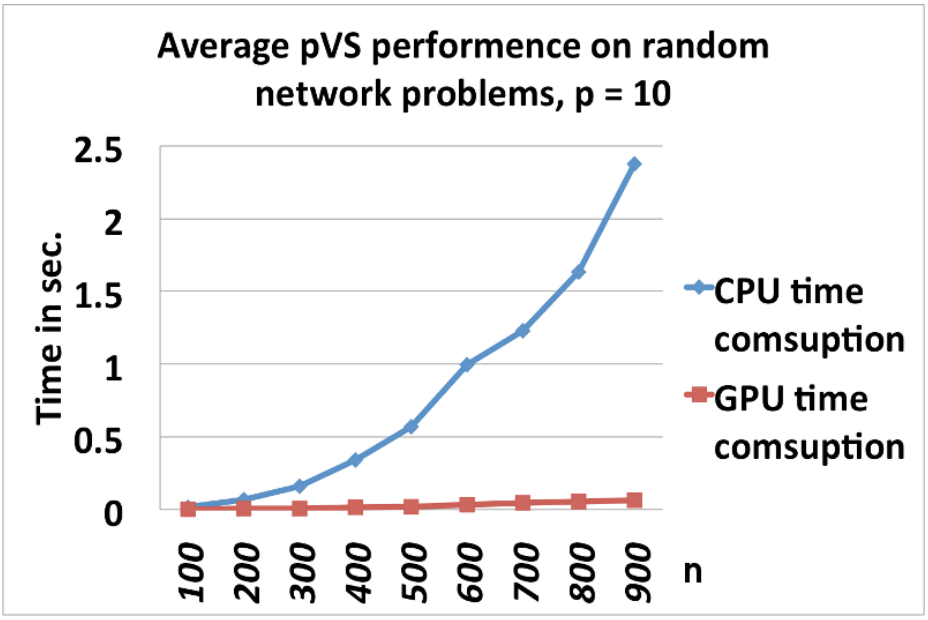


Figure 3.4: Average pVS performance on random network problems, $p = 10$

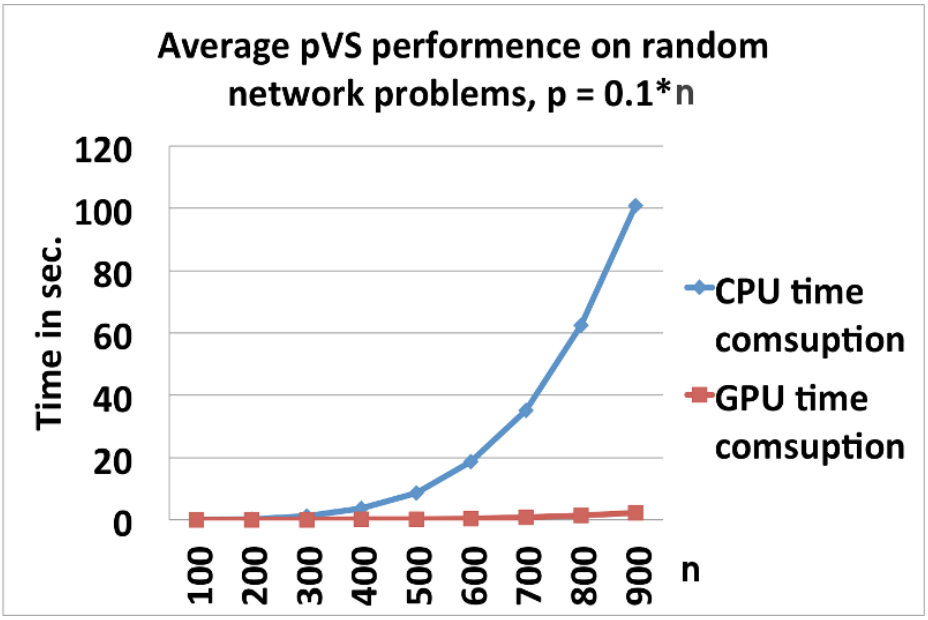


Figure 3.5: Average pVS performance on random network problems, $p = 10\%n$

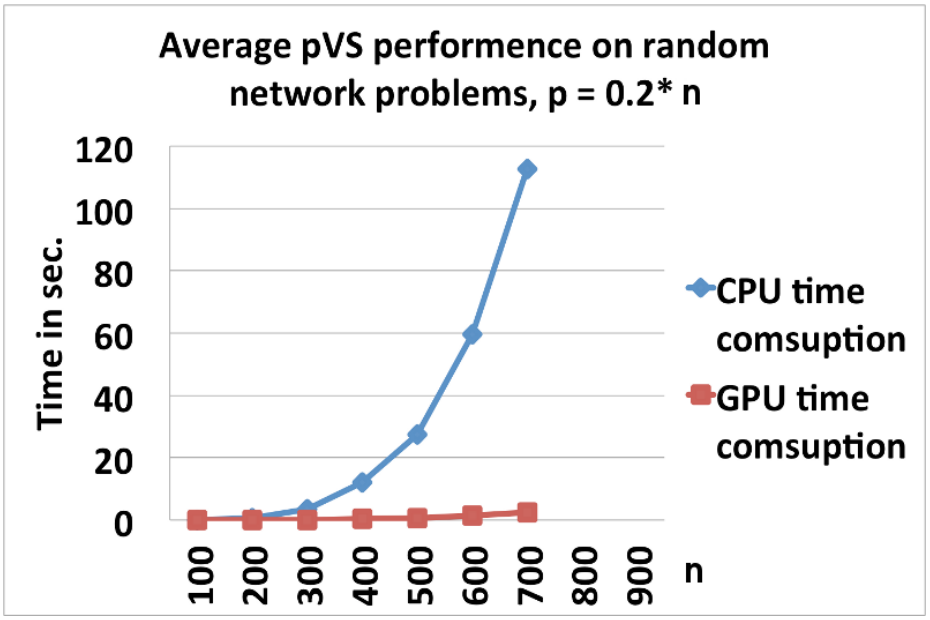


Figure 3.6: Average pVS performance on random network problems, $p = 20\%n$

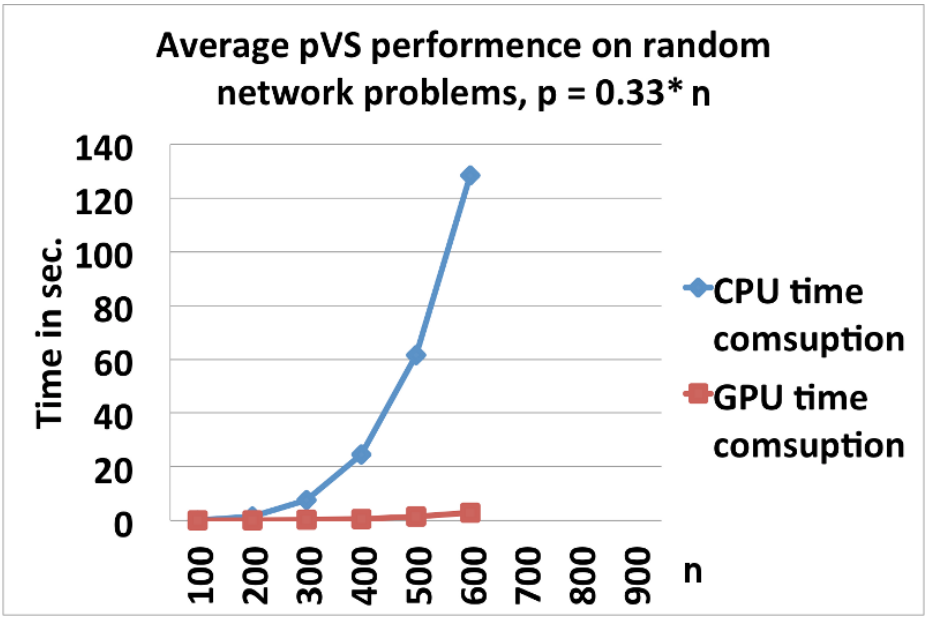


Figure 3.7: Average pVS performance on random network problems, $p = 33\%n$

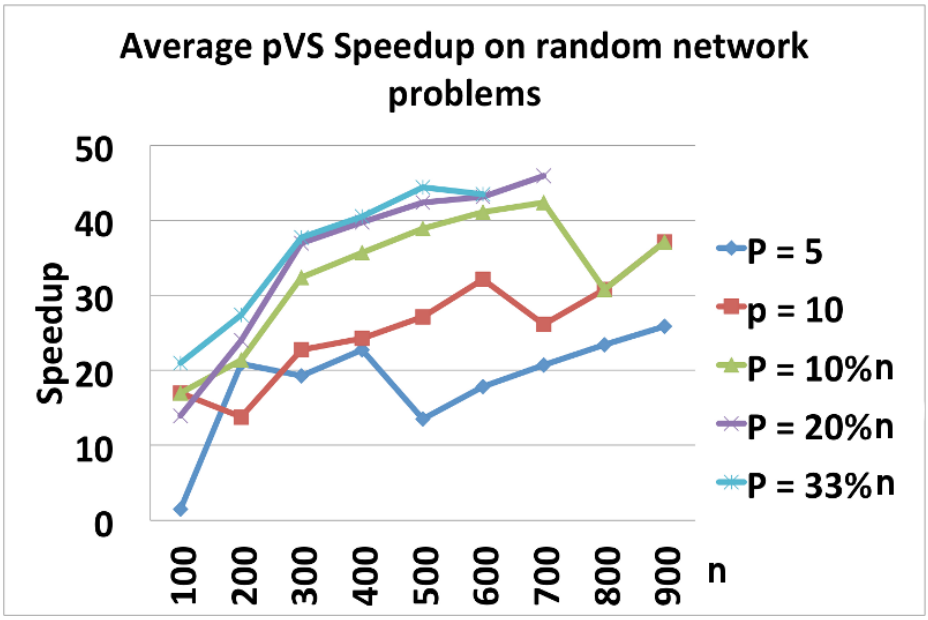


Figure 3.8: Average pVS performance on random network problems

and VS. The solution time of the CPU-based VS increases sharply when the size of the network increases. In contrast, the increasing rate of pVS's solution time is much lower. Figure 3.8 displays the average speedups of pVS over VS on different network sizes and with a different number of medians. The trend of speedup proves that pVS achieves more benefits when the problem size and the number of medians are large, and it is consistent with the results seen in our previous experiments on OR-lib. VS is an efficient algorithm and it still performs reasonably fast for problem instances with less than 1000 nodes. Good solutions could be found by VS in several minutes. When the problem size is beyond 1000 nodes with a large p, VS may fail to obtain good solutions within a reasonable time. Our experiments clearly show that pVS is a better alternative to VS, as it has the ability to solve large problems much faster than VS. Table 3.3 through Table 3.5 show the results of our experiments on large problem instances. When p is small, VS still works well. However, when p is larger than 10% or 20% of the node size, VS usually could not solve the problem within five hours. On the contrary, pVS is capable of solving those problem in a few minutes. In Table 3.4, VS solved a problem instance ($n = 3000, p = 300$) in about 4.2 hours while

pVS found the same solution in less than 5 minutes. In the largest problem we tested ($n = 5000, p = 1000$), the estimated solution time of VS is roughly 5 days (based on the number of iterations evaluated in 5 hours), whereas pVS found the solution in about two hours.

Table 3.3: pVS performance on large network problems $p = 10$

Problem ID	n	p	Objective Value		Solution Time		Speedups
			VS	pVS	VS	pVS	
0	1000	10	364.80	364.80	2.82	0.10	28.2
3	2000	10	754.68	754.68	20.58	0.58	35.5
6	3000	10	1090.53	1090.53	77.92	1.85	42.1
9	4000	10	1475.80	1475.80	89.04	2.09	42.6
12	5000	10	1749.92	1749.92	130.74	2.98	43.9

Table 3.4: Average pVS performance on large network problems, $p = 0.1 * n$

Problem ID	n	p	Objective Value		Solution Time		Speedups
			VS	pVS	VS	pVS	
1	1000	100	23.28	23.28	155.72	3.20	48.7
4	2000	200	23.46	23.46	2905.56	53.28	54.5
7	3000	300	23.68	23.68	15351.30	270.94	56.7
10	4000	400	N/A	23.08	$>$ 5 hours	822.38	N/A
13	5000	500	N/A	22.34	$>$ 5 hours	2103.91	N/A

Table 3.5: Average pVS performance on large network problems,$p = 0.2 * n$

Problem ID	n	p	Objective Value		Solution Time		Speedups
			VS	pVS	VS	pVS	
2	1000	200	7.72	7.72	492.61	10.10	48.8
5	2000	400	8.13	8.13	9036.87	159.33	56.7
8	3000	600	N/A	7.77	$>$ 5 hours	837.12	N/A
11	4000	800	N/A	7.92	$>$ 5 hours	2606.40	N/A
14	5000	1000	N/A	35.78	$>$ 5 hours	7405.54	N/A

3.6 Conclusion

We have developed a GPU-based parallel Vertex Substitution algorithm for the p-Median problem on the NVIDIA CUDA GPU architecture. We designed a GPU-CPU cooperation procedure for pVS which uses the GPU to compute expensive pVS operations in parallel and uses the CPU to coordinate the iterations and the termination of the algorithm. The worst-case complexity of pVS was reduced from sequential vertex substitution's $O(n^2)$ to $O(p * (n - p))$ on each parallel thread. We tested the performance of pVS on two sets of test cases and compared the results with a CPU-based sequential vertex substitution implementation (best-profit search). Those two test data sets include 40 different network instances from OR-lib, 400 similar randomly generated network instances, and 15 randomly generated large network instances. The pVS algorithm on a GPU ran significantly faster than the CPU-based VS algorithm in all test cases. In small network instances such as OR-lib problems and those 400 randomly generated network instances, pVS obtained 10x to 40x speedups. More substantial speedups (28x to 57x) were observed on larger network instances, which have more than 1000 nodes. This is particularly important because CPU-based VS could not solve some of these large instances within five hours, while the GPU-based pVS solved all instances in two hours or less.

Chapter 4

Design and Implementation of GPU-based Bounded Variable Simplex Algorithm

4.1 Linear Programming Model of Fluence Map Optimization

We define the intensity of a beamlet as w_{alp}, which is the intensity of radiation delivered from beam angle a through the $(l,p)^{th}$ beamlet, where $l = 1, 2, 3...m$, and $p = 1, 2, 3, ...n$. m is the number of MLC leaves and n is the number of units that a leaf can be moved. The 3D treatment volume is discretized into small voxels v_{xyz}. We denote $d_{(x,y,z),a,l,p}$ as the dose delivered from beam angle a through the $(l,p)^{th}$ beamlet with a weight of 1 and received by voxels v_{xyz}. The total dose D_{xyz} delivered to a voxel is $\sum_{a,l,p} w_{a,l,p} \cdot d_{(x,y,z),a,l,p}$. There are various formulations for FMO problems. We follow the FMO model used in Lim and Cao [84]

$$\begin{aligned} f(D) &= \lambda_t^+ \parallel (D_T - \theta_U \cdot e_T)_+ \parallel_\infty + \lambda_t^- \parallel (\theta_L \cdot e_T - D_T)_+ \parallel_\infty \\ &\quad + \frac{\lambda_s \parallel (D_S - \phi \cdot e_S)_+ \parallel_1}{\mid S \mid} + \frac{\lambda_n \parallel D_N \parallel_1}{\mid N \mid}, \end{aligned} \tag{4.1}$$

$$
\begin{aligned}
\min_{\omega} \quad & f(D) \\
\text{subject to} \quad D_{x,y,z} &= \sum_{a,l,p} (\omega_{a,l,p} \cdot d_{x,y,z,a,l,p}) && \forall (x,y,z) \in T \cup S \cup N \\
L_T &\leq D_{x,y,z} \leq U_T && \forall (x,y,z) \in T \\
D_{\bar{N}} &\leq U_{\bar{N}} && \forall (x,y,z) \in \bar{N} \\
0 &\leq \omega_{a,l,p} \leq M_{a,l,p} && \forall a \in \mathcal{A}', \;\; l = 1,2,...,m, \\
& && p = 1,2,...,n.
\end{aligned}
\tag{4.2}
$$

Notation	Definition
$\mathcal{A}$	Beam angle set
D	Dose deposited
$\omega_{a,l,p}$	IMRT beamlet weight
T	A set of voxels in PTV
S	A set of voxels in OAR
N	A set of voxels in Normal
θ_L	Cold spot control parameter on PTV
θ_U	Hot spot control parameter on PTV
ϕ	Hot spot control parameter on OAR
L_T	Lower reference bound on PTV
U_T	Upper reference bound on PTV
$U_{\bar{N}}$	Upper reference bound on normal structure
λ_t^+	Penalty coefficient for hot spots on PTV
λ_t^-	Penalty coefficient for cold spots on PTV
λ_s	Penalty coefficient for hot spots on OAR
λ_n	Penalty coefficient for normal structure

Table 4.1: Notations for FMO and BAO Model

4.2 Bounded Variable Simplex Algorithm

Consider a LP model in the following standard form:

$$
\begin{aligned}
\min \quad & c^T x \\
\text{subject to} \quad & \mathbf{A}x \leq b \\
& l \leq x \leq u,
\end{aligned}
\tag{4.3}
$$

where $c \in R^n$,$b \in R^m$ and $\mathbf{A} \in \mathbb{R}^{n*m}$. $l \geq -\infty$ and $u \leq \infty$. c is the vector containing the coefficients of objective function, x is the vector of variables, and l and u are the vectors of the variables' lower bounds and upper bounds. $\mathbf{A}$ is the matrix of constraints coefficients. $\mathcal{B}$ stands for the set of indices for basic variables, and $\mathcal{N}$ is the set of indices for non-basic variables. A simplex basis $\mathbf{B} = \mathbf{A}_{\mathcal{B}}$ is the sub-matrix of $\mathbf{A}$, whose columns are selected based on $\mathcal{B}$. Furthermore, we define two additional sets of indices: $\mathcal{N}_l$ and $\mathcal{N}_u$, such that $\mathcal{N}_l \bigcap \mathcal{N}_u = \oslash$, $\mathcal{N}_l \bigcup \mathcal{N}_u = \mathcal{N}$, $u_j < \infty$ for $j \in \mathcal{N}_u$, $l_j > -\infty$ for $j \in \mathcal{N}_l$. Variables belonging to $\mathcal{N}_l$ will be set to their lower bounds l, and variables in $\mathcal{N}_u$ will take value at their upper bounds u. Variables that can be fixed before the algorithm begins will be removed and responding impact will be added into $\mathbf{A}$ and c^T. We describe a modified variant of bounded variable simplex (BVS) method from Bixby [85] in Algorithm 3.

Our BVS method differs from the original algorithm by integrating steepest-edge pricing [86] for selecting entering variables and the EXPAND procedure [87] for selecting leaving variables. The standard simplex pricing rule selects the entering variable, which minimizes the reduced cost. The steepest-edge pricing method selects the entering variable which minimizes the normalized reduced cost, which was proven [88] to be more effective than the Dantzig rule in most cases. We implement a variant of steepest-edge approximation [86] as our pricing rule. The EXPAND [87] procedure improves the numerical stability and reliability of the simplex method. We now describe the steps of our bounded variable simplex procedure.

Step 0. Initialization

Given a feasible initial basis, compute dual variables π by solving

$$\mathbf{B}^T \pi = \mathbf{c}_{\mathcal{B}}. \tag{4.4}$$

Calculate the reduced costs for non-basic variables:

$$\mathbf{d}_{\mathcal{N}} = \mathbf{c}_{\mathcal{N}} - \pi^T \mathbf{N}. \tag{4.5}$$

Initialize steepest-edge weights δ, feasibility tolerance κ and feasibility increment τ.

Step 1. Optimality Check

If $d_j \geq 0$ for all $j \in \mathcal{N}_l$, $d_j \leq 0$ for all $j \in \mathcal{N}_u$, $\mathcal{B}$ is optimal: Stop.

Otherwise, select the most favorable normalized reduced cost

$$d_q^* = d_q / \sqrt{\delta_q}, \tag{4.6}$$

and select x_q as entering variable.

Step 2. Step direction

Compute the simplex step direction y by solving

$$\mathbf{B}y = \mathbf{A}_q. \tag{4.7}$$

Step 3. Ratio test with the EXPAND procedure

3.1 Calculate simplex step size θ using origin bounds $\mathbf{l}$ and $\mathbf{u}$

If $d_q < 0$, let

$$\theta_i = \begin{cases} +\infty & \text{if } y_i = 0, \\ (x_{\mathcal{B}_i} - l_{\mathcal{B}_i})/y_i & \text{if } y_i > 0, \text{ and} \\ (x_{\mathcal{B}_i} - u_{\mathcal{B}_i})/y_i & \text{if } y_i < 0, \end{cases} \tag{4.8}$$

Else if $d_q > 0$, let

$$\theta_i = \begin{cases} +\infty & \text{if } y_i = 0, \\ (u_{\mathcal{B}_i} - x_{\mathcal{B}_i})/y_i & \text{if } y_i > 0, \text{ and} \\ (l_{\mathcal{B}_i} - x_{\mathcal{B}_i})/y_i & \text{if } y_i < 0, \end{cases} \tag{4.9}$$

Let $\theta_p = \min(\min_i \theta_i, u_q - l_q)$ be the minimum ratio.

3.2 Calculate the minimum acceptable step

$$\theta_{min} = \tau \ y_p. \tag{4.10}$$

If $\theta_p \geq \theta_{min}$, then θ_p is the actual step size, and x_p is the leaving variable.

Otherwise, repeat equations 4.8 and 4.9 with extended bounds $\mathbf{l} - \kappa$ and $\mathbf{u} + \kappa$ to obtain a new θ_p, which will be selected as the actual step size. Then, update feasibility tolerance κ for the next iteration:

$$\bar{\kappa} = \kappa + \tau. \tag{4.11}$$

3.3 If $\theta_p = \infty$, problem is unbounded, stop

Step 4 Solution and matrices update

4.1 If $d_q < 0$, then $x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} - \theta_p y$. Else, $x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} + \theta_p y$;

4.2 If $\theta_p = u_q - l_q$, x_p will not enter basis. Instead, x_p will be moved from one bound to other. e.g. if $p \in \mathcal{N}_u$, then $\mathcal{N}_l \leftarrow \mathcal{N}_l \bigcup \{p\}$ and $\mathcal{N}_u \leftarrow \mathcal{N}_u \setminus \{p\}$ or vice versa.

Otherwise, update $\mathbf{B}$, $\mathbf{N}$, and the value of the entering variable

$$x_{B_i} \leftarrow \begin{cases} u_q - \theta_p & \text{if } q \in \mathcal{N}_u \\ l_q + \theta_p & \text{if } q \in \mathcal{N}_l, \end{cases} \tag{4.12}$$

Step 5. Compute pivot row $\bar{\alpha}$

$$\text{Solve } \bar{\mathbf{B}}^T \eta = e_p \text{ for } \eta, \tag{4.13}$$

$$\bar{\alpha} = -\bar{\mathbf{N}}^T \eta. \tag{4.14}$$

Step 6. Update reduced cost

$$\bar{d_q} = d_q / \alpha_q, \tag{4.15}$$

$$\bar{d_j} = \ d_j - \bar{\alpha_j} d_q \quad \text{, if } j \in \mathcal{N} \setminus \{q\} \ . \tag{4.16}$$

Step 7. Update approximation of steepest-edge weights

$$\bar{\delta}_q = \delta q / \alpha_q^2, \tag{4.17}$$

$$\bar{\delta}_j = \ \max(\delta_j, \bar{\alpha}_j^2 \alpha_q^2 + 1) - 2\bar{\alpha}_j^2 \alpha_q^2 + \bar{\alpha}_j^2 \delta_q \quad \text{, if } j \in \mathcal{N} \setminus \{q\} \ . \tag{4.18}$$

4.3 GPU Implementation of Bounded Variable Simplex Method

4.3.1 Overview of GPU-based Bounded Variable Simplex Solver

The implementation procedures of our main solver are illustrated in Figure 4.1. The rectangular nodes are GPU kernels, which perform parallel operations on the GPU, and diamond nodes, which are decision functions running on the CPU. Before the GPU solver begins, input data sets are prepared in the *host memory* (CPU memory) by CPU. Several pre-solve processes [89] are applied to the model to reduce model size and remove unnecessary constraints and variables. The next step is to

Algorithm 3 Bounded variable simplex method

procedure BOUNDED VARIABLE SIMPLEX METHOD
  repeat
    Solve $\mathbf{B}^T\pi = c_B$ for π
    Compute $d_N = c_N - \mathbf{A}_N^T\pi$. $(d_B = 0)$.
    if $d_j \geq 0$ for all $j \in \mathcal{N}_l$, $d_j \leq 0$ for all $j \in \mathcal{N}_u$ **then**
      $\mathcal{B}$ is optimal, Stop; (Optimality Check)
    else
      select an entering variable $x_{j_e}, j_e \in N$, such that d_{j_e} violates these conditions.(Pricing)
    Solve $\mathbf{B}y = \mathbf{A}_{j_e}$.
    (Ratio Test)
    if $d_{j_e} < 0$, **then** let

$$\theta_i = \begin{cases} +\infty & \text{if } y_i = 0, \\ (X_{B_i} - l_{B_i})/y_i & \text{if } y_i > 0, \text{ and} \\ (X_{B_i} - u_{B_i})/y_i & \text{if } y_i < 0, \end{cases}$$

    else if $d_{j_e} > 0$, **then** let

$$\theta_i = \begin{cases} +\infty & \text{if } y_i = 0, \\ (u_{B_i} - X_{B_i})/y_i & \text{if } y_i > 0, \text{ and} \\ (l_{B_i} - X_{B_i})/y_i & \text{if } y_i < 0, \end{cases}$$

    $\theta = \min\{\min_i \theta_i, u_{je} - l_{je}\}$.
    if $\theta = +\infty$ **then**, (4.3) is unbounded, Stop.
    else
      Update $x_{\mathcal{B}}$
      if $d_{j_e} < 0$ **then** $x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} - \theta y$
      else
        $x_{\mathcal{B}} \leftarrow x_{\mathcal{B}} + \theta y$;
      if $\theta = u_{je} - l_{je}$ **then**
        if $j_e \in \mathcal{N}_u$ **then** $\mathcal{N}_l \leftarrow \mathcal{N}_l \bigcup \{j_e\}$ and $\mathcal{N}_u \leftarrow \mathcal{N}_u \setminus \{j_e\}$
        else$\mathcal{N}_u \leftarrow \mathcal{N}_u \bigcup \{j_e\}$ and $\mathcal{N}_l \leftarrow \mathcal{N}_l \setminus \{j_e\}$
      else if $\theta < u_{je} - l_{je}$ **then** Update $j_l = B_i$, $B_i = j_e$, and set

$$X_{B_i} \leftarrow \begin{cases} u_{j_e} - \theta & \text{if } j_e \in \mathcal{N}_u \\ l_{j_e} + \theta & \text{if } j_e \in \mathcal{N}_l, \end{cases}$$

        if $d_{je} y_i < 0$ **then** $\mathcal{N}_l \leftarrow \mathcal{N}_l \bigcup \{j_l\}$
        else if $d_{je} y_i > 0$ **then** $\mathcal{N}_u \leftarrow \mathcal{N}_u \bigcup \{j_l\}$
  until Optimality

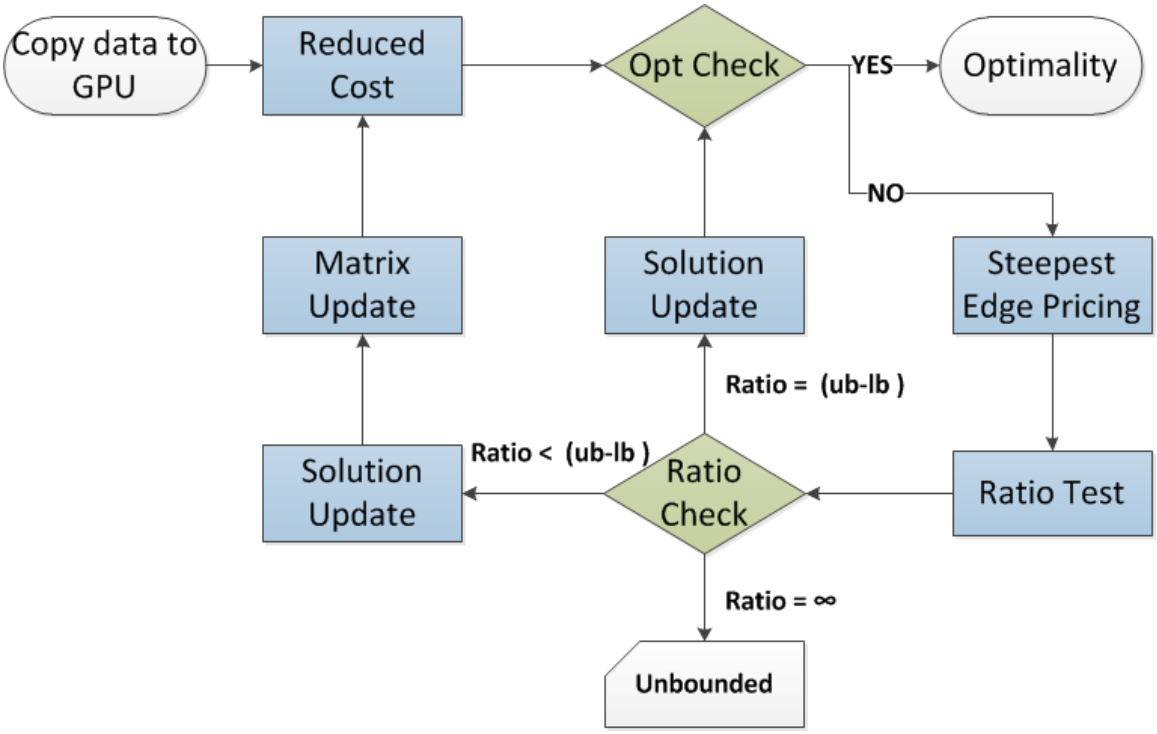


Figure 4.1: GPU-based Bounded Bariable Simplex solver

convert the above input data from dense representation to a sparse matrix format. We used a column major coordinate list (CMCOO) to store our coefficients matrix. Additionally, an initial solution, including $\mathcal{B}$, $\mathcal{N}_l$, $\mathcal{N}_u$, and $x_{\mathcal{B}}$, is taken by the solver as an input. As long as the initial solution passes the *feasibility check*, the simplex model and solution are copied from *host memory* to *device memory* (GPU global memory) as the finishing step of the solver initialization. Based on the current $\mathcal{B}$ and $\mathcal{N}$, the GPU LP solver first updates update $\mathbf{B}$ and $\mathbf{N}$ and then calculates reduced cost. The step of calculating reduced cost involves solving one linear system $\pi^T\mathbf{B} = c_B^T$ for π, and one matrix-vector multiplication $\mathbf{N}^T\pi$ to calculate reduced cost d_N. The pricing step finds best beneficial entering variable based on the steepest-edge rule. Ratio test determines the leaving variable by solving a linear system $\mathbf{B}y = \mathbf{A}_{je}$ for y. After the solver obtains a pair of entering variable and leaving variable, a parallel matrix update is performed to construct the new $\mathbf{B}$ and $\mathbf{N}$ for the next iteration until the

solver reaches optimality or is terminated by the unbounded condition. Cusp [90] is used to solve the linear system in the above procedures.

4.3.2 Source of Parallelism

The bounded variable simplex method is a sequential algorithm; each step is built on the result provided by previous step. Besides, in each iteration, only a pair of entering and leaving variables are selected to update the solution. As a result, there is no task parallelism in this simplex implementation. However, data parallelism is very common in the simplex procedures. This is because the simplex method heavily relies on linear algebra operations. The bounded variable simplex method we implemented involves solving two sparse linear systems in step 3 and 10 in Algorithm 3, two column independent, one row independent, and four element independent parallel operations, as shown in Figure 4.1. In this paper, we focus on maximizing the data parallelism in simplex method and optimize the GPU memory usage to handle large-scale LP problems.

4.3.3 Pre-solve Process

A modern LP solver usually has two components: the pre-solve process and the main solver. The pre-solve process is often critical to reduce solving time because LP models are usually not in their most compact forms. The pre-solve process can provide equivalent models to the original model but save memory and solution time. Common pre-solving rules described by Andersen [89] are implemented, includeing remove redundant constraints, replace constraints by simple bounds, represent columns by bounds on shadow prices, and fix variables at their bounds.

4.3.4 Data Structure

4.3.4.1 Column Major Coordinate List

We implement sparse matrix and dense vector for all of our linear algebra operations. For sparse matrix storage, we design a column major coordinate list (CMCOO) format to provide fast row reference and column selection. An Example of the CMCOO matrix is illustrated in Figure 4.2. The CMCOO format is a modification based on the COO format [91].

```
typedef struct {
    int nCol;
    int nRow;
    int nnz;
    int *col_idx;
    int *row_idx;
    double *val;
    int *colptr;
} CMCOO_Mat;
```

The standard COO matrix is a sparse matrix format with sophisticated parallel implementation in sparse linear algebra operations. The COO matrix uses one array to store the value of each non-zero element and another two arrays to store row and column index for each non-zero element. In CMCOO matrix, all non-zero elements are stored in an one-dimensional array linearized based on columns. Two standard row and column index arrays are still employed. Then, we add another array, *colptr*, to record the beginning position of each column.

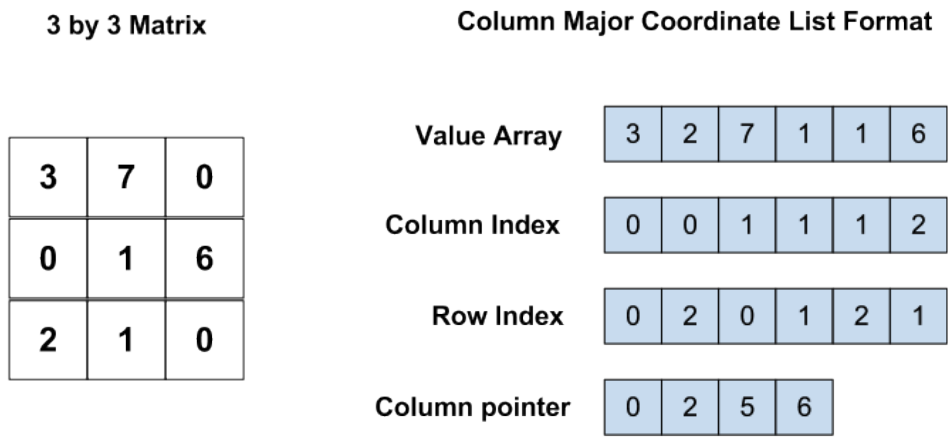


Figure 4.2: Column major coordinate list format

4.3.4.2 Benefits of CMCOO Format

Simplex is a column-based algorithm. In each simplex iteration, the simplex method tries to find an entering variable and a leaving variable, whose indices are going to switch between the sets $\mathcal{B}$ and $\mathcal{N}$. The Matrix $\mathbf{B}$ and $\mathbf{N}$ are defined by selecting the columns of matrix $\mathbf{A}$ in $\mathcal{B}, \mathcal{N}$, respectively. Because the simplex ratio test also relies on the solution of $\mathbf{B}y = \mathbf{A}_{je}$, whose right-hand side is the column of $\mathbf{A}$ corresponding to the entering variable. Furthermore, most pricing rules, such as steepest-edge, are column-related operations. Thus, the simplex data structure should support the algorithm for fast column selection. The *colptr* array of CMCOO format enables the GPU solver to easily locate a column and estimate the column's non-zero elements.

Furthermore, simplex algorithm involves intensive row-based operation, such as Matrix-vector multiplication and vector-vector subtraction. These operations are row independent and can be very important for parallel implementation. Thus, it is also critical that we have row reference in the sparse data representation, which is difficult to obtain using common column-based sparse format, such as Column Compress

Storage (CCS). CMCOO format has the advantage of CSS format. Non-zero elements are stored in linearized memory based on columns and *colptr* provides us necessary information to perform fast column selection. Meanwhile, efficient parallel linear algebra operations optimized for COO format can be applied to CMCOO format without major modification.

4.3.5 Parallel Matrix Update

For each iteration, two major updates may be required. The first is called the *solution update*, which is to update value of $x_{\mathcal{B}}$, and the second is the *parallel matrix update* for generating $\mathbf{B}$ and $\mathbf{N}$ according to the new $\mathcal{B}$ and $\mathcal{N}$.

In *solution update*, only the basic variables will be updated, based on Algorithm 3 step 16. It is straightforward to use one thread $thread_i$ to update one variable in $x_{\mathcal{B}}$. Since each of the non-basic variables in $x_{\mathcal{N}}$ will be set to either its lower bound or upper bound, we do not track the actual value of non-basic variables. Instead, we only track which category each non-basic variable belongs to, i.e., $\mathcal{N}_l$ or $\mathcal{N}_u$, by using array *var-cat*. Non-basic variable $x_{N_i} \in \mathcal{N}_l$ has a positive value in *var-cat* while non-basic variable $x_{N_i} \in \mathcal{N}_u$ has a negative value.

Traditionally, the matrix update is accomplished by implementing *backward transformation* or $BTRAN$, and *forward transformation* or $FTRAN$ [92], which involves iteratively solving or calculating $\mathbf{B}_{q+1} = \mathbf{B}_0 E_1 E_2 ... E_q$ where q is the current number of iterations and E is the eta matrix to update $\mathbf{B}$. In our GPU LP solver, we implement a parallel explicit update to replace $BTRAN$ and $FTRAN$ so as to avoid the sequential linear algebra operations and extra memory usage.

The *parallel matrix update* is divided into four parts: (1) estimates number of non-zero elements of each new column (nnz_i), (2) finds starting position for each new column ($sptr_i$), (3) updates non-zero elements and their row and column indices of each new column, and (4) calculates the total number of non-zero elements in the

new matrix.

Algorithm 4 Pseudocode for parallel matrix update

1: **procedure** PARALLEL MATRIX UPDATE
2: Activate $thread_i$ for $i \in \{0, 1, 2, ..., m-1\}$.
3: Each $thread_i$ calculates $nnz_i = colptr_{i+1} - colptr_i$.
4: $thread_i$ initialize $sptr_i = 0$.
5: **for** $j = 0 \rightarrow i$ **do**
6: $thread_i$ calculates $sptr_i = sptr_i + nnz_j$.
7: Activate $block_j$ for $j \in \{0, 1, 2, ..., m-1\}$.
8: Activate $thread_i$ for $i \in \{0, 1, 2, ..., nnz_j - 1\}$ in each $block_j$.
9: $thread_i$ copies $A.val[colptr_j + i]$ to $B.val[sptr_j + i]$.

The example in Figure 4.3 explains how the parallel matrix update was implemented to update $\mathbf{B}$ according to new $\mathcal{B}$. On our *device global memory*, we have the CMCOO format $\mathbf{A}$. $\mathbf{B}$ and $\mathbf{N}$ are sub-matrix of $\mathbf{A}$ by selecting different columns based on $\mathcal{B}$ and $\mathcal{N}$. Assume that we have m basic columns, n non-basic columns, and total $n + m$ columns in $\mathbf{A}$. For updating the basic matrix $\mathbf{B}$, this kernel uses m threads and each thread $thread_i$ calculates the number of non-zero elements in column $\mathbf{B}_i$, i.e., nnz_i. Note that the i^{th} column in $\mathbf{B}$ is the $\mathcal{B}_i^{th}$ column in $\mathbf{A}$, i.e., $\mathbf{B}_i = \mathbf{A}_{\mathcal{B}_i}$. $thread_i$ can easily obtain $nnz_i = colptr_{\mathcal{B}_i+1} - colptr_{\mathcal{B}_i}$. The results are stored in array *nnz-map*. Given the fact that the starting position for the first column is always $\mathbf{B}_0$, the starting position of each column in $\mathbf{B}$ is then obtained by running a prefix sum on the *nnz-map* array. Now, each $thread_i$ has matched the beginning position $sptr_i$ in $\mathbf{B}$ with that column beginning position $colptr_{\mathcal{B}_i}$ in $\mathbf{A}$. By looping through 0 to nnz_i, $thread_i$ copies the non-zero element values and indices of the corresponding column from $\mathbf{A}$ to $\mathbf{B}$. Once all threads have finished copying, the new $\mathbf{B}$ is generated. Furthermore, the total number of non-zero elements will be the last column's start position plus its number of non-zero elements.

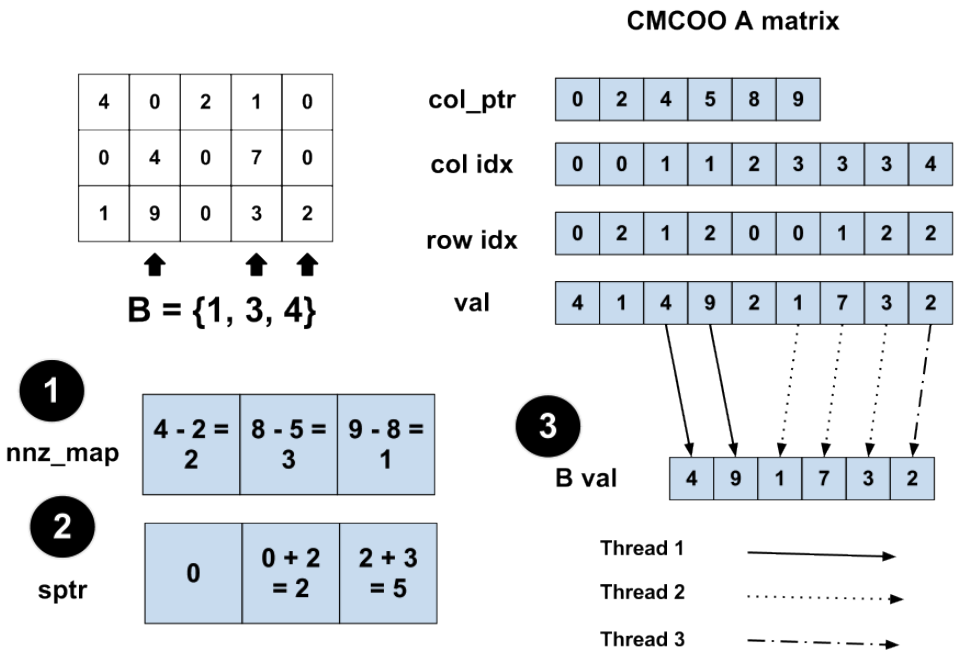


Figure 4.3: Parallel matrix update

4.3.6 Optimality Check

The bounded variable simplex algorithm reaches optimality *iif* the reduced cost d_i is positive for all non-basic variable $x_{N_i} \in \mathcal{N}_l$ and d_i is negative for all non-basic variable $x_{N_i} \in \mathcal{N}_u$. The *Optimality Check Kernel* takes the array of reduced cost d_N as input. Each thread $thread_i$ will multiply d_{N_i} with its *var-cat* value, and the multiplication results are stored in an array called *opt-condition*. At this point, any negative value in *opt-condition* is a violation of optimality condition, and such a variable will be considered as a candidate entering variable. A parallel reduction is then performed on *opt-condition* to find the minimum element. If the minimum element in *opt-condition* is non-negative, then the algorithm reaches optimality.

4.3.7 Ratio Test

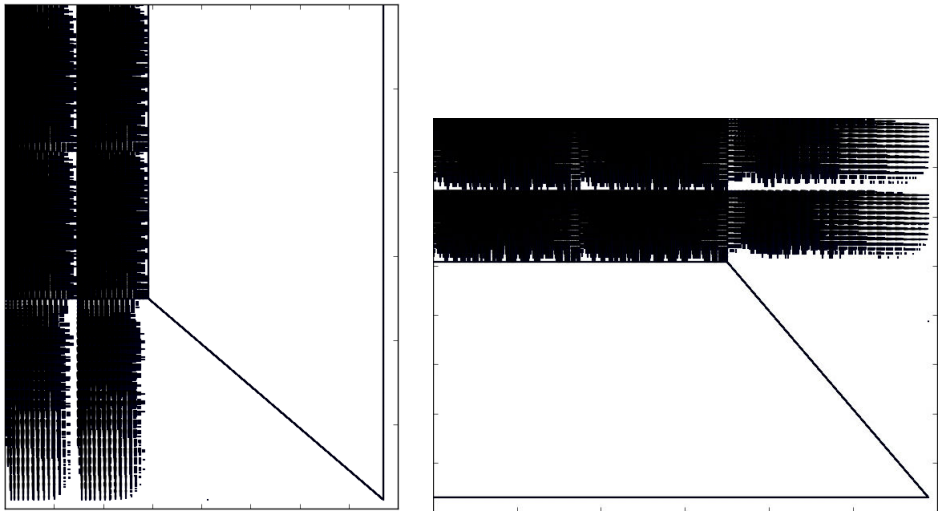
The ratio test is to find the maximum value that an entering variable can be increased from its lower bound or decreased from its upper bound while not violating any constraints or feasibility. The ratio test first solves a linear system $\mathbf{B}y = \mathbf{A}_{je}$ for y via the Cusp iterative solver. Then, depending on the entering variable's *var-cat* value, we determine if d_{je} is positive or negative. Each thread $thread_i$ in ratio test kernel calculates the ratio of each $X_{\mathcal{B}_i}$ according to Algorithm 3 step 10. The results are stored in an array called *theta*. A parallel reduction is performed on the *theta* to find the minimum ratio. The index of the minimum ratio refers to the position of the leaving variable in $\mathcal{B}$. This minimum ratio $theta_m$ is copied back to *host memory*. The value of $theta_m$ needs to be compared with the difference between entering variable's upper bound and lower bounds, $u_{je} - l_{je}$, in order to ensure the feasibility of the entering variable. This comparison is done on the CPU. At this point, the algorithm has obtained a pair of indices of the entering and leaving variables, and it will continue to the *Update Kernel*. However, if the $minimumratio = \infty$, the algorithm will be terminated because the problem is unbounded.

4.4 Experiments on Radiation Treatment Planning Problem with GPU-based LP Solver

4.4.1 Pre-Solve Processes

Fluence Map Optimization (FMO) is a large-scale LP problem due to its number of constraints and variables. Current GPUs usually have less than 4GB on-board memory on each video card. Given the constraint that FMO problem size is large but GPU memory is limited, it is quite important to have memory-efficient model to handle the FMO problem on a GPU.

In our FMO model, one important component of our objective is that we want



(a) Primal FMO (b) Dual FMO

Figure 4.4: Non-zero data structure of FMO problem

to panelize overdose on each OAR voxel by minimizing $\frac{\lambda_s \|(D_S - \phi \cdot e_S)_+\|_1}{|S|}$ in Function 4.1, which provides good control of OARs' over doses. However, this objective forces our model to have one extra constraint and one extra variable to control the total dose and overdose for each OAR voxel. Figure 4.4 visualizes the non-zero data structure of a typical FMO problem. The bottom half of 4.4(a) are the constraints for the OAR voxels and the diagonal sub-matrix is the coefficient of the control variable for each OAR voxel's over dose. In order to have a better representation of the diagonal OAR over doses, we first convert the FMO model to its dual form, as shown in Figure 4.4(b). The diagonal structure now has been transformed to a group of constraints with only one non-zero element per row. Such rows are much easier removed from the LP model by replacing them with simple bounds on corresponding variables [89]. Then, we performed the other pre-solve process method described in Chapter 5 to further reduce the problem size.

Table 4.2: Voxel information

Pancreas case			Prostate case		
Voxel set	Set size	Voxels in Use	Voxel set	Set size	Voxels in Use
Pancreas	1244	1244	Prostate	1767	1767
Spinal Cord	489	332	Rectum	5848	4141
Left kidney	9116	945	Bladder	10603	2923
Right Kidney	5920	587	Normal	9865	6483
Liver	50391	1998			
Normal	8787	7609			

Table 4.3: Problem size and matrix density

	Pancreas case	Prostate case
Original	4662×13959	8226×17085
Pre-solved	202×13511	646×17016
Density after pre-solve	7.23%	6.70%

4.4.2 Numerical Experiments

Currently, we have tested our solver on two clinical cases: one pancreas case with four beam angles and one prostate case with six beam angles. Table 4.2 lists the number of voxels of each organ in both cases. Two processes are applied before we generate the simplex model. First, we sample the voxels in certain organs to reduce the data size without compromising the treatment quality [84]. We uniformly select 1/8 voxels from the liver set in pancreas case. For the normal set, we only sampled voxels which surrounded the PTV area for both cases. Secondly, we only sampled the voxels which have positive values in the dose-contribution matrix from the angles we are interested in. The simplex models are generated based on the information listed in Table 4.2, and pre-solve processes are applied to the models. Pre-solve results and matrix densities are listed in Table 4.3.

The CPU we used was an Intel Core i7 950 3.0 GHz with 24 GB RAM, and the GPU was an Nvidia GTX 480 with 1.5 GB GPU memory. We solved the pancreas case using CPLEX 12.1 with GAMS interface using a single CPU thread as a benchmark.

Table 4.4: Time consumption and objective value

Case	Pancreas		Prostate	
	Objective value	Time (sec)	Objective value	Time(sec)
CPLEX 1 thread	0.317	40.10	0.447	65.92
CPLEX 2 threads	0.317	40.04	0.447	66.19
CPLEX 4 threads	0.317	40.37	0.447	66.60
GPU	0.317	18.56	0.447	39.54
Speedup	N/A	2.16	N/A	1.71

The computational results, listed in Table 4.4, show that the GPU-based simplex solver reached the same optimal solutions when compared to CPLEX, but at a faster rate. We also tested CPLEX with one, two, and four threads on both cases as benchmarks. We control the CPLEX threads by using the *threads* parameter in *CPLEX options*. The results show that using multiple threads in CPLEX did not obtain any benefits. This is because CPLEX *threads* is a parameter designed for parallel MIP solver. Although multiple threads are assigned to CPLEX, only one thread is used to solve the LP model in execution using primal simplex method. We calculated the speedups based on single-thread CPLEX solution time. We observed 2.1x and 1.7x speedups obtained by the GPU solver for the pancreas and prostate case, espectively.

Figures 4.5 and 4.6 are the dose distribution plots for both problems, which are used to visualize the solution quality. The dotted line is the CPLEX solution and the solid line is the solution of GPU LP solver. We can see that the GPU LP solver provided the same solution quality compared to CPLEX while taking much less time. However, it is difficult to conclude the performance and quality of the GPU LP solver at this point. We need to test the solver on more cases as well as different selected beam angles to evaluate the quality and speed.

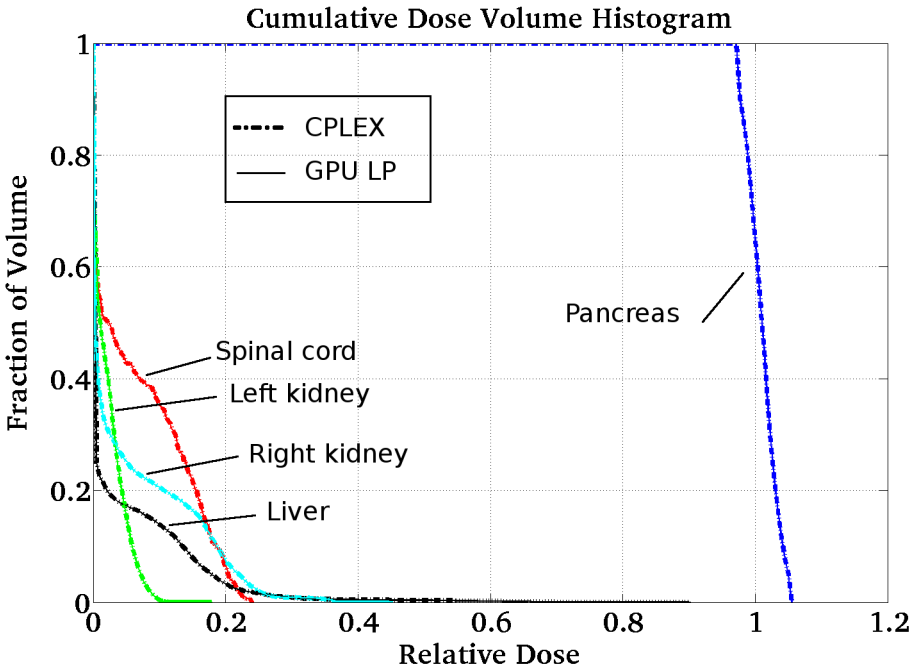


Figure 4.5: DVH plot for IMRT pancreas cases

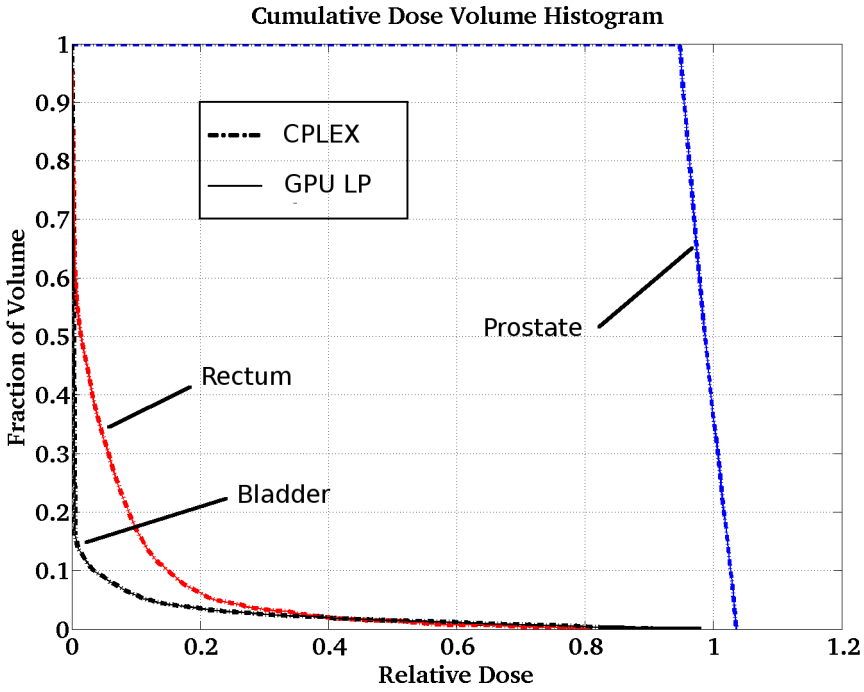


Figure 4.6: DVH plot for IMRT prostate cases

Chapter 5

MPI-Based Parallel Framework for Beam Angle Optimization in Radiation Treatment Planning

5.1 MPI-based Parallel Genetic Algorithm

5.1.1 Global Parallel Genetic Algorithm for a Type of Combinatorial Optimization Problem

We consider a type of combinatorial optimization problem in the integer programming model

$$\begin{aligned} \min \quad & Z = \mathbf{ax} + \mathbf{by} \\ s.t. \quad & \mathbb{A}\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \leq \mathbf{c}, \\ & \mathbf{u} \cdot \mathbf{y} = d, \end{aligned} \tag{5.1}$$

where $\mathbf{x}$ is the vector of continuous variables and $\mathbf{y}$ is vector of binary variables. $\mathbf{u}$ is a unit vector. There are general constraints on both type of variables as well as constraints to limit the number of selected binary variables. We assume there are much more continuous variables than binary variables in this model.

In the proposed Global Parallel Genetic Algorithm (GpGA), we only consider the binary variables $\mathbf{y}$ in the chromosome so that each chromosome represents a

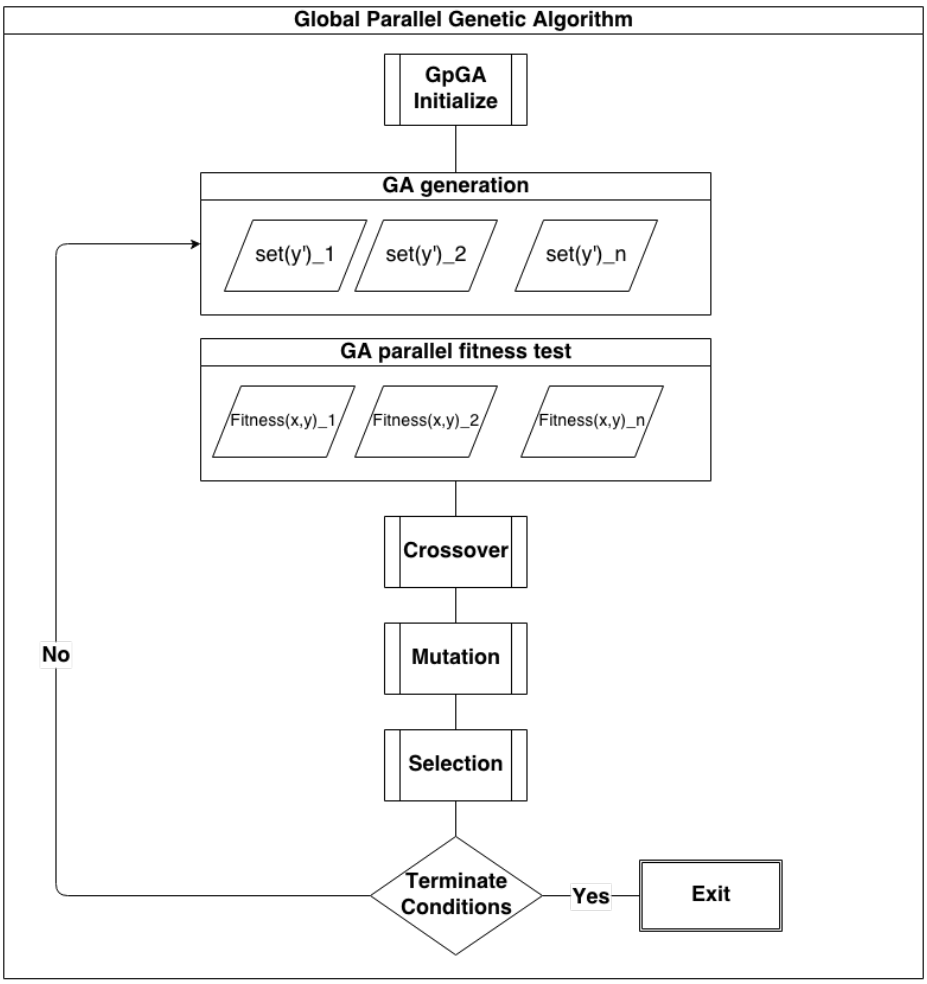


Figure 5.1: Global Parallel Genetic Algorithm

certain combination of binary variables $\mathbf{y}'$. The initialization step generates a set S', where $\{S' = \mathbf{y}'_1, \mathbf{y}'_2, ..., \mathbf{y}'_p | S' \in S\}$ and p is number of individuals in each population. Taking a certain $\mathbf{y}'$ as parameters, we can obtain a subproblem $fitness(\mathbf{x}, \mathbf{y}')$. Each $fitness(\mathbf{x}, \mathbf{y}')$ in a certain generation is then evaluated in parallel. As illustrated in Figure 5.1, GpGA starts by creating a initial generation of $\mathbf{y}'$. Each chromosome $\mathbf{y}'$ is a randomly selected binary combination which meets the general constraints of $\mathbf{y}$. The fitness value of each chromosome is evaluated in parallel. The fitness results are collected to apply Crossover and Mutation operations. Elitism selection is used to generate the next iteration if the current generation does not meet the terminate conditions.

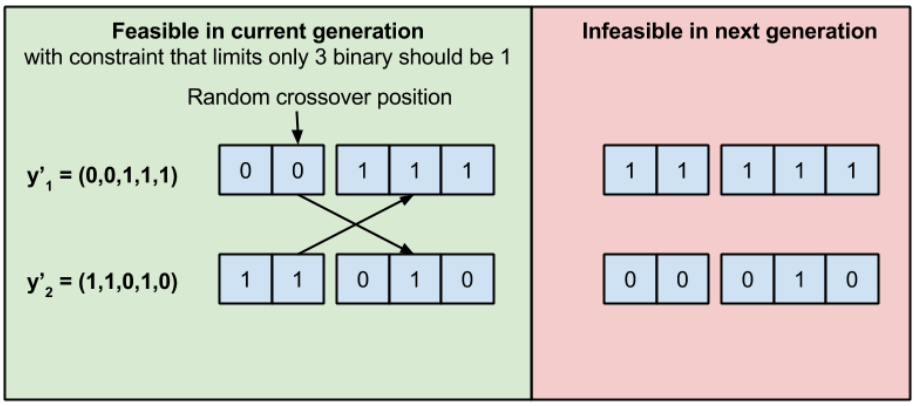


Figure 5.2: Binary Encoding

5.1.1.1 GpGA Components

Fitness Test For each $\mathbf{y}'_i \in S'$, a linear programming subproblem $LP(\mathbf{x}, \mathbf{y}')$ is obtained by fixing the value of $\mathbf{y}$ to be $\mathbf{y}'_i$. In our GpGA, we solve the subproblem $LP(\mathbf{x}, \mathbf{y}')$ to its optimal solution and the objective function value is returned as a fitness value of chromosome $\mathbf{y}'_i$. Compared to the traditional approximate fitness test, this approach ensures that there won't be any misleading direction caused by fitness approximation.

Encoding Method The original $\mathbf{y}'$ is a binary string which fits the general GA chromosome format. However, binary encoding is not aware of any constraints of the binary variables. Typical GA operations, such as crossover and mutation, will easily generate infeasible $\mathbf{y}'$ as the example shows in Figure 5.2. Besides, repeat fitness tests are too expensive as the fitness test is a time-consuming operation. Binary string is not intuitive for identification or avoidance of repeats. Furthermore, when problem size grows, the communication time used for delivering genes to the parallel computer increases linearly with gene string length.

To overcome the above issues, we designed two layers of encoding for each $\mathbf{y}'$. The first layer (index encoding) using indices of non-zero binary variables as the value of each allele in a gene. The length of the gene becomes a fixed property in the gene

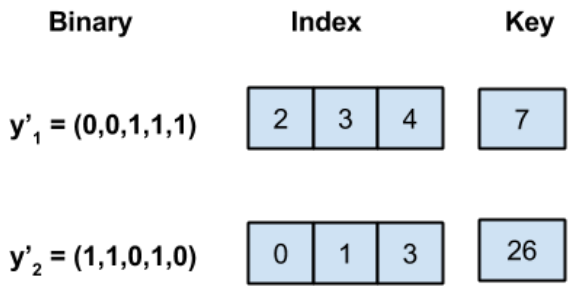


Figure 5.3: Index and Key encoding

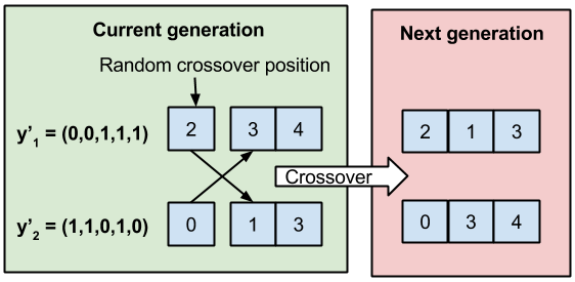


Figure 5.4: Single position crossover

to represent the constraint on $\mathbf{y}$. As a result, the GA operations always inherit this information and will never alter the number of non-zero variables in the evolution process. A second layer of encoding (key encoding) is applied to the gene so that each will be assigned a unique key, as described in Figure 5.3 and Algorithm 5. The key encoding helps to easily identify a gene and simplify the distribution of the gene for parallel computing.

Algorithm 5 Key Encoding Pseudocode

```
1: procedure KEY ENCODING(key)
2:     if |y| < 64 then
3:         i ← 0
4:         while i < |y| do
5:             key ← key * 2 + y'[i]
6:             i ← i + 1
7:     else
8:         key ← bitset(y), where bitset is C++ std::bitset
       return key
```

Crossover As described in Algorithm 6, proposed GpGA uses single-position crossover between a pair of two chromosomes. In the crossover operation, a random pair of chromosomes are selected from the current generation, and a random position is selected so that the two chromosomes exchange a portion of their gene, as described in Figure 5.4.

Algorithm 6 Single position crossover pseudocode

```
1: procedure SINGLE POSITION CROSSOVER
2:     cr ← fixed crossover rate
3:     S_j ← {y_1, y_2, ..., y_p} in j^th iteration.
4:     while S_j ≠ ∅ do
5:         Randomly select pair(y_n, y_m), where n, m ≤ p, n ≠ m
6:         cr' ← RandomNumber ∈ [0, 1]
7:         K ← {0, 1, 2, ...|y|}
8:         if cr' ≤ cr then
9:             while K ≠ ∅ do
10:                k ← RandomPosition ∈ K
11:                (y'_n, y'_m) ← crossover(y_n, y_m) at k^th position.
12:                if (y'_n, y'_m) is valid then
13:                    S_{j+1} ← S_{j+1} ⋃ {y'_n, y'_m}
14:                    S_j \ {y_n, y_m}
15:                    Break
16:                else
17:                    K ← K \ {k}
18:        else
19:            S_{j+1} ← S_{j+1} ⋃ {y_n, y_m}
20:            S_j ← S_j \ {y_n, y_m}
       return S_{j+1}
```

Mutation We consider three types of single-position mutation in our GpGA. For each $\mathbf{y}$ in S_{j+1} returned by crossover, a random number mr' is generated. If this mr' is greater than the fixed mutation rate mr, mutation is applied to this chromosome. (1) Random single-position mutation is selected as our default mutation scheme. A random position in the chromosome is selected; the index value on the selected position will then be replaced by a valid random index which does not exist in this chromosome. (2) Score-based mutation selectes the position in the chromosome

which has the lowest performance score subject to the fitness value. In the application of radiation treatment planning, this performance score is calculated based on a linear combination of function values using Formula 5.2. The $\min Score(a), a \in \mathcal{A}'$ is selected to be mutated to a valid random value. (3) We also use a combination of the above two types of mutation where score-based mutation is applied to GpGA and random mutation is requested by MPI-framework to generate extra candidate chromosomes when idle computing resources are available.

$$\begin{aligned} Score(a) &= (s_a^T / \sum_{a \in \mathcal{A}'} s_a^T) - w_s \cdot (s_a^S / \sum_{a \in \mathcal{A}'} s_a^S), \quad w_s \geq 0 \\ uD(T)_a &= \sum_{(x,y,z) \in T} \sum_{l,p} d_{x,y,z,a,l,p}, \qquad a \in \mathcal{A}' \\ D(\psi)_a &= \sum_{(x,y,z) \in \psi} \sum_{l,p} w_{a,l,p} \cdot d_{x,y,z,a,l,p}, \qquad \psi \in \{T, S\} \\ s_a^T &= D(T)_a / uD(T)_a \\ s_a^S &= D(S)_a / |S| \end{aligned} \tag{5.2}$$

5.1.2 MPI-based Master-Worker framework for GpGA

A MPI-based master-worker framework is designed to implement our proposed GpGA algorithm. As illustrated in Figure 5.5, the Master Processor hosts major GpGA operations, including crossover, mutation, and selection. Each worker processor only works on one assigned fitness test.

5.1.2.1 Solution History Database and Load Balance

As with many master-worker frameworks, every worker should receive a similar amount works in order to maximize parallel efficiency. At the same time, worker processors should never receive repeat jobs due to the complexity of the fitness test in our case. To achieve such objectives, the master processor also controls the load-balance scheduler and the fitness solution database. The solution database uses chromosome's key-encoding as a primal key and records returned fitness value and LP solution time

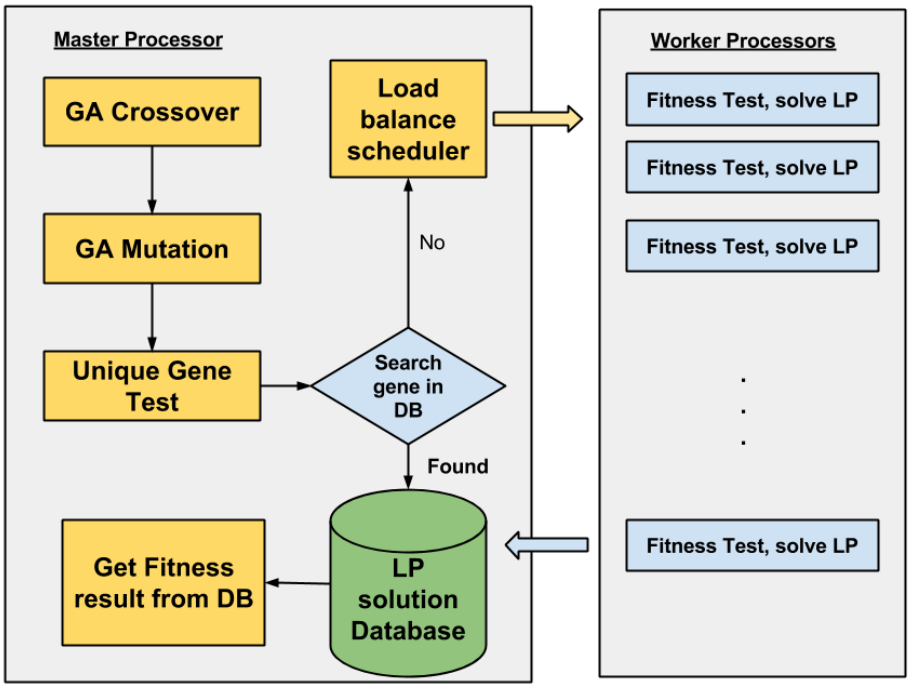


Figure 5.5: Generalized MPI GpGA framework for combinatorial optimization problem

from each worker.

All candidate chromosomes generated on the master processor will need to pass two level filers based on their key encoding. The first filter is to select only unique chromosomes among the current generation. The second filter applies a key search in the solution database. Any existing solution in the database is immediately returned with the fitness value and the remaining unvisited candidates form a task pool.

Traditional load-balance optimizations such as Dynamic Adjusting Factoring (DAF) or Dynamic Predictive Factoring (DPF) [93] do not apply to our situation for the following reasons. (1) Each fitness test is a LP problem, whose solution time is hard to predict since the solution algorithms can be non-polynomial, like Simplex. (2) Each fitness test cannot be divided into sub-tasks, since the complexity of each sub-task cannot be accurately estimated and there's no MPI-based parallel LP solver outperforming leading sequential LP solvers in general [56].

As a result, we use a minimum fixed chunk size, single LP fitness test, in our load-balance scheduler. The master processor sends one candidate chromosome in

the task pool to each worker at a time and the returning fitness value is stored in the solution database. Once a worker returns a fitness value, the master processor will then assign a new chromosome from the task pool until no chromosome is available.

In order to offset a situation where certain workers take much more time to obtain the fitness value than others when the task pool is already empty, the idling workers can be optionally assigned extra candidates which are not initially in the task pool. These extra candidate chromosomes are generated based on current generation. The master processor will temporarily increase the population size of GpGA in this iteration and dynamically create extra chromosomes by forcing extra mutation.

5.2 MPI-based Master-Worker Hybrid Parallel Framework

5.2.1 Parallel Simulated Annealing

Based on the MPI-based framework for proposed GpGA, we developed a hybrid parallel framework that generates candidate subproblems from different algorithms. We propose a parallel Simulated Annealing (pSA) algorithm for solving a type of Combinatorial Optimization Problem defined in Equation 5.1. pSA only considers the binary variable $\mathbf{y}$ in its process, and the evaluation of a candidate $\mathbf{y}'$ is obtained by solving its corresponding LP subproblem optimally. This pSA shares the same structure for variable vector $\mathbf{y}$ with GpGA as well as the same expensive evaluation function (LP subproblem). As a result, pSA can be implemented on our MPI-framework without major modification. The basic SA approach for parallel implementations proposed here is briefly described as follows.

Initial solution Multiple candidates $\mathbf{y}'$ are randomly generated on the Master processor. Each candidate will start a pSA instance independent from others.

Neighbor generation scheme Neighbor solution $\mathbf{y'_{j+1}}$ is obtained by using functions described in Algorithm 7

Algorithm 7 SA Neighbor generation Pseudocode

```
1: procedure SA NEIGHBOR GENERATION
2:     y'_j is a SA candidate in j^th iteration.
3:     k ← random position in y'_j.
4:     α ← random step size generated by Geometry Distribution.
5:     d ← random direction ∈ {−1, 1}
6:     y'_{j+1} ← y'_j
7:     y'_{j+1}[k] = y'_j[k] + d · α
```

Accept a neighbor solution The newly generated neighbor solution $\mathbf{y'_{j+1}}$ is evaluated and this solution is accepted if the associated LP subproblem has an improvement in its objective function value. Otherwise, SA uses the geometric cooling scheme to generate an accept rate for the non-improving solution as described in Formula 5.3.

$$PA(\Delta, t_j) = exp(-(\Delta/(z(\mathbf{x_j}, \mathbf{y'_j}) \cdot t_j))), \tag{5.3}$$

where, PA is the probability of acceptance, $z(\mathbf{x_j}, \mathbf{y'_j})$ is the objective function value of the LP subproblem associated with $\mathbf{y'_j}$. Δ is the difference between $z(\mathbf{x_{j+1}}, \mathbf{y'_{j+1}})$. And t_j is the SA temperature at j^{th} iteration. If a random number generated between zero and one is less than PA, this non-improving move will be accepted. Initially, the temperature $t_0 = 1$ and once a non-improving move is accepted, the temperature is reduced as Formula 5.4.

$$t_{j+1} = t_j/(1 + \beta t_j), \tag{5.4}$$

where β is a non-negative small constant value less than one. The purpose of this temperature cooling function is to direct the search away from unfavorable regions each time a non-improving solution is accepted. A threshold temperature is determined. Once temperature drops below such threshold, re-annealing takes place and

resets the temperature to one.

5.2.1.1 Parallel SA Implementation

We implemented our pSA uses Synchronous Approach with Occasional Enforcement of Best Solution-Fixed Intervals (SOEB-F). In this approach, the master processor randomly generates multiple initial solutions for independent SA chains. We note each SA chain as a pSA instance. The master processor broadcasts the best solution among all pSA instances and each worker starts to develop its SA chain until a fixed number of iterations when the master broadcast another best solution. We also implemented two types of cooling schemes; the first type uses a global shared temperature and the second maintains independent temperature for each pSA instance.

5.2.2 Framework Architecture Overview

Figure 5.6 illustrated the overview structure of this hybrid framework. We consider a parallel Simulated Annealing algorithm (pSA) to supply our GpGA in the framework. Both Genetic Algorithm and Simulated Annealing have independent initialer to generate random starting candidates. Similar to GpGA, the proposed pSA only controls binary variables and generates corresponding LP subproblems for evaluation. Each pSA candidate is an independent process but can be connected with others via the temperature-cooling model or migration operations as described in Section 5.2.1. All candidates from GpGA and pSA are organized into the single task queue on the master processor and distributed to workers by the load-balance scheduler. Once the optimal objective values of the candidates are obtained, they will be sent back to their origin algorithm model. The migration process is the communication model that exchange the best candidate among individual candidates, or between two algorithms. GpGA needs to wait for the fitness value of all candidates

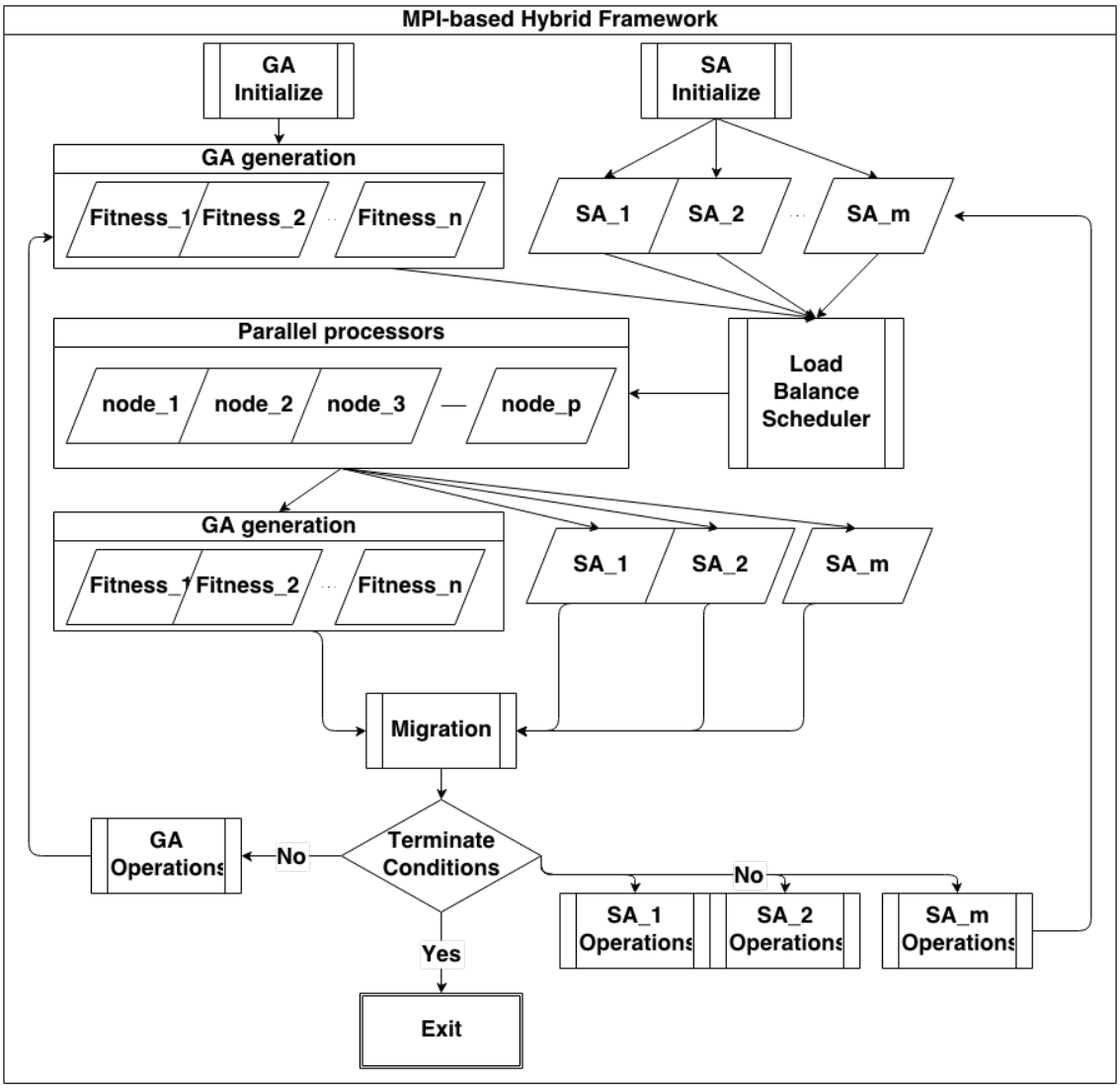


Figure 5.6: Hybrid Framework Overview

in the current generation to precede the GA operations, such as crossover and mutation. A pSA instance can perform SA operation to generate the next candidate independently from the others.

Load-Balance Scheduler and Task Queue The load-balance scheduler uses a similar First In First Out rule as the GpGA framework. However, one important difference is the task queue in this hybrid framework. In GpGA framework, the task pool is generated per iteration by collecting unique and unvisited candidates in the current GA generation. The task pool has a fixed number of tasks for each generation. All tasks in the pool must be finished in order to proceed to next GA operations. On contrary, the task queue in this hybrid framework is assumed to be an infinite queue. Hybrid algorithms will continuously add candidates to the queue for evaluation. The GpGA still adds tasks per generation, but pSA adds a new task to the queue as soon as the previous candidate is evaluated and new candidate is obtained by SA operations.

Migration The migration process controls the communication between pSA instances and GpGA generation. A global best candidate $\mathbf{y_g}$ is maintained on the master processor. Each time the framework discovers a better candidate $\mathbf{y'_g}$, the migration will identify the origin algorithm of $\mathbf{y'_g}$. If it's a GA candidate, the migration will find the worst-performance pSA instance and set $\mathbf{y'_g}$ as its next candidate. If $\mathbf{y'_g}$ is a pSA candidate, migration will replace the worst GA chromosome in the next GA generation.

5.2.3 Numerical Experiments

5.2.4 Experiments Environment

We prepared two BAO problem in real-life prostate cancer cases to test our MPI-based GpGA framework. Table 5.1 listed the number of voxels in each organ set and beamlet per angle for both cases. Two typical angle settings, selecting 6 from 12 angles or selecting 6 from 36 angles, are used in this experiments. The experiment computer has 2 AMD Opteron 8-core 2.8GHz CPUs and 256GB RAM on a NUMA architecture with Ubuntu 12.04 LTS operation system.

	Prostate	Rectum	Bladder	Beamlet per angle
Case 1	3269	5848	10603	220
Case 2	6375	5719	7850	250

Table 5.1: Test Problems: IMRT Beam Angle Optimization for Prostate cases

5.2.4.1 Experiment Benchmarks

Table 5.2.4.1 illustrated benchmark solution time of solving BAO using CPLEX 12.5 (Barrier LP method). The MIP column in Table 5.2.4.1 displays the objective value and time of CPLEX using 16 parallel threads. The LP relaxation column is the results of the relaxed BAO problem where all binary variables are forced to be one, and the problem becomes a FMO LP problem. The results show that in 12-angle cases, CPLEX can solve both cases within half an hour for case1 and two and a half hours for case2. However, for 36 angles, CPLEX cannot reach a good solution even after a week. The MIP-LP gap column shows the quality of using LP relaxation as BAO lower bound. For BAO problem, the LP relaxation usually has a much lower objective function value compared to the MIP model.

Table 5.3 shows the parallel efficiency of CPLEX's multi-thread LP solver in a sequential GA using CPLEX barrier LP method. It is clear that CPLEX's multi-thread already provides noticeable speedup when using barrier method, and hence

	MIP		LP relaxation		MIP-LP gap
	Obj. val.	Time(min)	Obj. val.	Time(min)	
Case 1-12	0.164714	28.5	0.163326	0.43	0.8%
Case 2-12	0.218666	151	0.213735	0.47	2.3%
Case 1-36	NA	>7days	0.156682	1.8	NA
Case 2-36	NA	>7days	0.199979	4.8	NA

Table 5.2: CPLEX 12.5 using 16 threads, MIP benchmarks

Threads	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
1	0.218305	60	82	19	9.16%
16	0.215615	45.18	199	33	7.81%

Table 5.3: Sequential GA, average of 10 runs

all the following comparisons and speedups will based on CPLEX using multi-thread. Since the total resource is fixed, when more processors are assigned to the MPI framework, less CPLEX can be used by CPLEX LP solver.

5.2.5 The Performance of GpGA on MPI Framework

Although GA has been actively researched for several decades, there is still no general optimal setting for its parameters that work best for all problems []. We determined the following GA parameters by previous successful implementations in the literature in this area.

- Population size: twice the number of treatment beam angles [94, 95]
- Stopping Criteria (1) 1% gap to LP relaxation of BAO for 12 angle cases, 5% gap for 36 angle cases. (2) Ten consecutive non-improvement iterations. (3) One hour total time. (4) Theoretical upper bound to reach optimal solution with 80% confidence.[safe, 2004]
- All results are based on average of 10 runs for each combination.

As to the crossover and mutation rate, we performed preliminary experiments to find the best combination for our cases. As listed in Figure 5.7 and Figure 5.8, each data

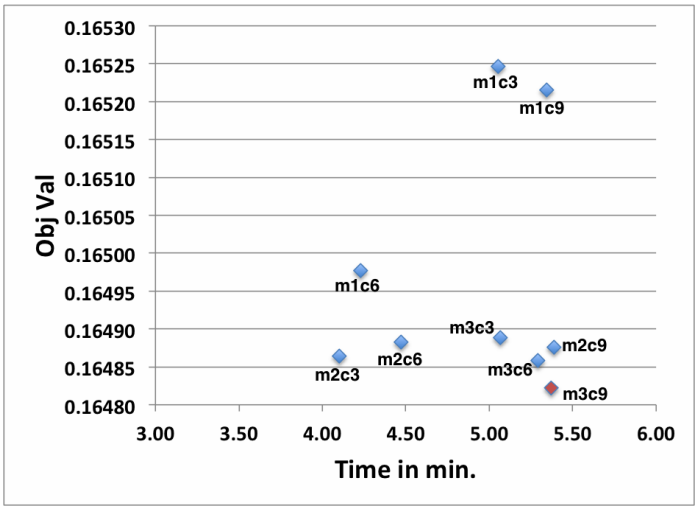


Figure 5.7: Case1, Experiments on Mutation and Crossover Rate

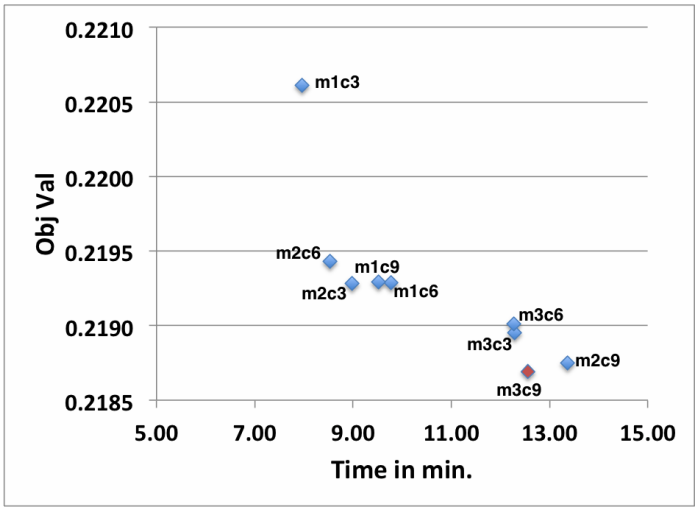


Figure 5.8: Case2, Experiments on Mutation and Crossover Rate

point represented the average objective function value of GpGA. For each data point label, m is mutation rate and c is crossover rate, e.g. m1c3 = 10% mutation and 30% crossover rate. All experiment results are based on average of 10 runs.

The best combination towards best objective function value for GpGA was found with 30% mutation rate and 90% crossover rate.

5.2.5.1 Parallel Speedup

Table 5.4 lists the BAO solutions from the GpGA framework using a m3c9 rate combination with 1 master processor and 7 workers. The results demonstrated that the GpGA MPI framework obtained near-optimal solutions, which are less than 0.1% to the MIP solution with 5x and 10x speedups compared to CPLEX solver. Note that the CPLEX MIP solver is allowed to use all threads on the computer by setting *lpthreadLabel*=0. Table 5.5 and Table 5.6 illustrate the speedup using GpGA framework from

	Gap to MIP	Time(min)	Speedups	FMO solved	Iterations
Case 1	0.06%	5.3	5x	56	9
Case 2	0.01%	12.5	12x	103	22

Table 5.4: Population 12, Crossover 90%, Mutation 30%, 12 angle cases

1 master 1 worker to 1 master 7 workers in 36 angle cases. For case1, all settings were able to terminate within 5% to the LP relaxation gap and for case2, all settings terminated by reaching ten consecutive non-improvement iterations. In both cases, allowing more worker processors in the MPI framework obtained speedups of 2x.

5.2.5.2 Comparison in mutation types

We observed that the three types of mutation have similar performance. Table 5.7 shows that, in case1, score-based mutation reached 5% termination gap slightly faster than random mutation. The combined mutation reaches terminate condition

nproc	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
2(1-1)	0.164081	12.84	93	12	4.72%
4(1-3)	0.164220	7.75	103	13	4.81%
8(1-7)	0.164249	6.95	120	16	4.83%

Table 5.5: Case 1 Average Speedup in MPI framework

nproc	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
2(1-1)	0.21561	45.18	199	33	7.81%
4(1-3)	0.21589	25.98	179	29	7.95%
8(1-7)	0.21539	21.40	189	29	7.71%

Table 5.6: Case 2 Average Speedup in MPI framework

Type	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
Random	0.16393	5.06	92	11	4.61%
Score	0.16418	4.48	83	9	4.76%
Score+R	0.16416	3.50	74	7	4.74%

Table 5.7: Case 1, 36 angle mutation type comparison

Type	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
Random	0.215396	21.40	189	29	7.70%
Score	0.215849	21.58	185	27	7.93%
Score+R	0.215775	22.40	226	24	7.89%

Table 5.8: Case 2, 36 angle mutation type comparison

Type	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
Random	0.214819	60	458	96	7.42%
Score	0.214828	60	417	101	7.43%
Score+R	0.214179	60	590	75	7.10%

Table 5.9: Case 2, 36 angle mutation type, terminated at 1 hour

Type	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
Separate Cooling	0.216542	60	617	88	8.28%
Global Cooling	0.216674	60	618	88	8.34%

Table 5.10: Case 2, 36 angle pSA, terminated at 1 hour

30% faster than random mutation. In Table 5.8, all three types of mutation have similar performance on case2 and failed to reach the target 5% gap, stopped by having maximum number of non-improvement iterations. We extended the experiments on case2 by removing all terminate conditions except the one-hour limit. Table 5.9 shows that combined mutation was be able to improve the objective value, while random and score-based mutation have very similar performance.

5.2.5.3 The Performance of pSA on MPI framework

We evaluated pSA performance on our MPI framework on test case2. The results show that the two types of pSA obtained a similar objective function value. Both global cooling and separate cooling pSA performed slightly worse than GpGA, as listed in Table 5.10.

Type	Obj. val.	Time(min)	FMO solved	Iterations	Gap to LP relax
Case1	0.164268	3.5	81	4	4.84%
Case2	0.213718	60	576	48	6.87%

Table 5.11: Hybrid Framework 36 angle performance

Iteration	GpGA improves obj.val	pSA improves obj.val
1	0.223518	
2	0.221423	
3	0.220326	
6	0.216338	
18		0.216269
25	0.214648	
27		0.21446
29		0.213748

Table 5.12: GpGA and pSA improves global objective function value

5.2.5.4 The performance of Hybrid Framework

Table 5.11 lists the performance of the hybrid framework on both test cases. For both cases, hybrid framework obtained better performance compared to the stand-alone GpGA or pSA. In case1, the hybrid framework uses less time to reach the 5% gap terminate condition. In case2, only the one-hour terminate condition is used. The hybrid framework obtained a better objective solution compared to all types of our GpGA implementation. Table 5.12 lists details of GpGA and pSA communications in one of the hybrid framework experiments in case2. In the beginning iterations, GpGA maintained a better solution than pSA, and the later iterations showed that pSA obtained better a global solution and contributed the solution to GpGA.

5.2.5.5 Quality of the treatment plan in DVH plot

Figure 5.9 and Figure 5.10 are the dose distribution histogram (DVH) for the experiment problems, which are used to evaluate the solution quality. We can see that the MPI framework provides very good coverage on target organs and delivered low dose on organ-in-risk.

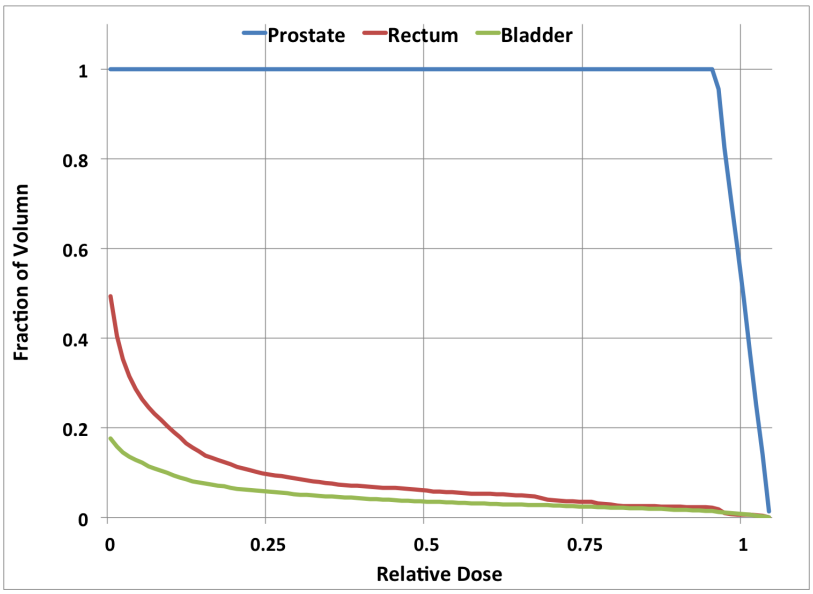


Figure 5.9: Case1, DVH plot

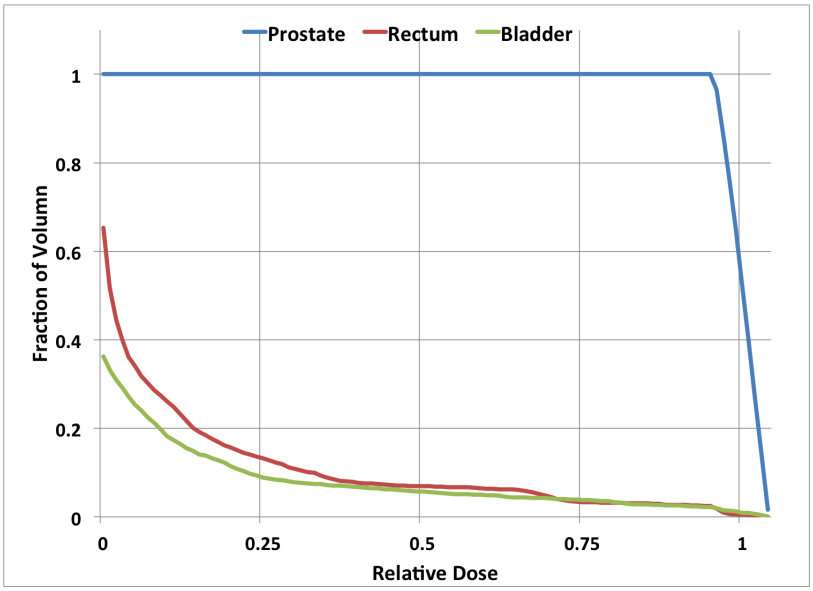


Figure 5.10: Case2, DVH plot

5.2.6 Summary

We have presented a MPI-based Global parallel Genetic Algorithm for solving a type of combinatorial optimization problem. This pGA uses two layers of real-encoding and the optimal solution of linear programming subproblem as the fitness test. We implemented this pGA on a MPI-based master-worker framework. A solution database was deployed on the master processor to maintain the uniqueness of the expensive fitness test and load balancing. The results of the numerical experiments in two large-scale BAO problems showed the effectiveness of this framework. By allowing more processor resources to be controlled by proposed GpGA MPI framework, more than 2x speedups were observed when compared to CPLEX's multi-thread techniques when using the same total processing resources. The experiments also showed that this framework can obtain clinical-standard solutions for radiation therapy within one hour, while the traditional MIP solver will take more than one week to reach the solution. We also observed in the mutation type comparison that, by using combined mutation, the framework has a better schedule for the same total amount of resources and solved around 20% more fitness evaluation (LPs). As for the pSA , we observed that both pSA implementations are able to reach good solutions within one hour. However, pSA solution is not as good as GpGA. As a result, we decided to use GpGA as the base algorithm for the hybrid framework and set pSA aside to cooperate with GpGA. The experiments show that the hybrid framework can obtain a better solution than any stand-alone algorithm in this experiment.

Chapter 6

Conclusions and Future Work

6.1 Current Findings

Recognizing the challenges in solving large-scale LP and MIP problems, this dissertation investigated three types of parallel solution approaches: (1) a parallel heuristic algorithm to solve the IP problem (pVS for the p-median problem), (2) an exact-solution algorithm to solve LP models (pBVS for FMO problem), and (3) a hybrid approach using optimal LP solutions as evaluating functions for parallel heuristic frameworks (hybrid framework for the BAO problem). We also explored different parallel computing platforms. The design and implementation of pVS and pBVS are to probe the merging platform of general-purpose computation on GPUs. The MPI-based Master-Worker Hybrid Framework for solving the BAO problem is designed to provide a framework that can be implemented on widely adopted parallel computing architectures from a single non-uniform memory access (NUMA) workstation to a computer cluster.

We presented a GPU-based parallel Vertex Substitution algorithm for the p-Median problem. pVS adopted a GPU-CPU cooperation procedure which used the GPU to compute expensive pVS operations in parallel and used the CPU to coordinate the iterations and the termination of the algorithm. The worst-case complexity

of pVS was reduced from $O(n^3)$ to $O(p*(n-p))$ on each parallel thread. The performance of pVS was evaluated on two sets of test cases and the results were compared with a CPU-based sequential vertex substitution implementation with best-profit search. The pVS algorithm on GPU ran significantly faster than the CPU-based VS algorithm in all test cases. pVS obtained 10x to 40x speedups in small network instances. The speed gain was more substantial for larger network instances having more than 1000 nodes with a larger number of medians, we observed a speed gain of 28 to 57 times. Furthermore, the CPU-based VS could not solve some of the larger problem instances within five hours (estimated solution time for sequential VS CPU is at least two to three days), while the GPU-based pVS solved all instances in two hours or less.

We implemented a GPU-based bounded variable simplex method to solve large-scale sparse linear programming problem, such as the FMO problem in IMRT treatment planning. The GPU LP solver is designed for general linear programming problems, but we implemented specific pre-processing techniques to optimize FMO model and GPU memory usage. This GPU LP solver uses a column major coordinate list to store sparse matrix data, which benefits the solver with fast column-selection ability as well as efficient sparse linear algebra operations. We solved the FMO problem using the proposed GPU LP solver on a Pancreas cancer case and a Prostate cancer case. The result was compared to CPLEX 12.1 with the GAMS interface. The computational result shows that the GPU LP solver obtained a similar optimal objective value as CPLEX, while the solution time is twice faster than CPLEX in this case.

In the MPI-based Master-Worker Hybrid Framework, we proposed a Global Parallel Genetic Algorithm, a Parallel Simulated Annealing Algorithm and a hybrid approach for solving Beam Angle Optimization problem. We designed this framework to handle a type of Combinatorial Optimization problem, where integer variables can be separated from the MIP model and linear programming subproblems can be

generated using any feasible combination of integer variables. By using the optimal LP solution as the evaluation test, the framework ensures that the search direction is accurately contributed by each evaluation. Such optimal evaluation tests were previously considered computationally too expensive to be implemented in heuristic algorithms. However, our parallel framework offsets the cost of the evaluations and achieved at least 5x-12x speedup compared to traditional solution techniques. As a result, we were able to obtain clinically acceptable BAO and FMO solutions in a fast manner.

6.2 Future Work

Based on the current progress on MPI-based parallel computation, one possible research direction can be to generalize the MPI-based hybrid framework in three areas; (1) Develop a standard interface for the Master-Worker framework so that additional algorithms can be added as a component to provide an extra source of LP subproblems. We observed that the current hybrid framework is benefited by the communications between GpGA and pSA. However, there are also various other algorithms available in the literature that have the potential to improve the solution quality and speed. By providing an interface for the framework to include new algorithms, we can rapidly develop the ability to evaluate different algorithms and promote framework performance. (2) Develop a Graphic User Interface (GUI) for an end user so that it is convenient for end user to modify parameters for the component algorithms in the framework. In the context of cancer treatment planning, the optimization requires agility in order to synchronize with different planning emphases in different treatment phases. (3) Generalize the MPI framework for other COP applications. Although our framework was designed for the general type of COP, the experiments were focused on the applications in IMRT problems. Based on the above improvement, the future development can provide a robust solution framework for a

generalized COP problem, which can be classified in the MIP model as described in Formula 5.1.

The solution database maintained on the master processor is an important feature of the proposed MPI-based Master-Worker Hybrid Framework. This database currently only serves as a record of visited notes so that the framework never evaluates the same expensive subproblem repeatedly. However, as the framework continuously accumulates the amount of solution history for a specific problem instance, such as a Prostate cancer case or for a type of problems such as Beam Angle Optimization, various solution techniques, such as data mining or statistical inference, can be applied to identify the relationships between beam angle candidate sets and their associated LP subproblem solutions. One can apply logistic regression to the history of binary variable combinations and their LP subproblem objective values, in which the regression function can be used to predict the quality of unvisited candidate solutions or at least suggest better solution candidates that have not been explored.

References

[1] T. Berhold, G. Gamrath, T. Koch, and D. Steffy. Mixed Integer Problem Library. `http://miplib.zib.de/`, 2010. Accessed: 2014-07-01.

[2] Varian Medical Systems Inc. Varian Medical Systems Newsroom - Image Gallery. `http://newsroom.varian.com/index.php?s=31899&cat=2475`, 2014. Accessed: 2014-07-01.

[3] NVIDIA Corporation. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz36BAUXcAq`, 2014. Accessed: 2014-07-01.

[4] G. B. Dantzig. Programming in a Linear Structure. *Econometrica*, 17:73–74, 1949.

[5] R. E. Bixby. Solving Real-World Linear Programs: A Decade and More of Progress. *Operations Research*, 50(1):3–15, 2002.

[6] P. B. Mirchandani and R. Francis. *Discrete Location Theory*. John Wiley & Sons, New York, 1990.

[7] C. J. Friedrich. *Alfred Weber's Theory of the Location of Industries*. University of Chicago Press, Chicago, 1929.

[8] S. L. Hakimi. Optimum Locations of Switching Centers and the Absolute Centers and Medians of a Graph. *Operations Research*, 12(3):450–459, 1964.

[9] S. L. Hakimi. Optimum Distribution of Switching Centers in a Communication Network and Some Related Graph Theoretic Problems. *Operations Research*, 13(3):462–475, 1965.

[10] N. Christofides. *Graph Theory. An Algorithm Approach*. Academic Press, New York, 1975.

[11] P. Hansen and B. Jaumard. Cluster Analysis and Mathematical Programming. *Mathematical Programming*, 79:191–215, 1997.

[12] J. Krarup and P. Pruzan. The Simple Plant Location Problem: Survey and Synthesis. *European J. Oper. Res.*, 12:36–81, 1983.

[13] N. Mladenović, J. Brimberg, P. Hansen, and J. A. Moreno-Perez. The p-median Problem: A Survey of Metaheuristic Approaches . *European Journal of Operational Research*, 179(3):927 – 939, 2007.

[14] O. Kariv and S. L. Hakimi. An Algorithmic Approach to Network Location Problems. *SIAM Journal on Applied Mathematics*, 37(3):539–560, 1979.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, 1979.

[16] American Cancer Society. Cancer Facts and Figures. `http://www.cancer.org`, 2014. Accessed: 2014-07-01.

[17] L. Mell, A. Mehrotra, and A. Mundt. Intensity-modulated radiation therapy use in the us, 2004. *Cancer*, 104(6):1296–1303, 2005.

[18] P. S. Cho, S. Lee, and R. J. Marks and. IMRT: A Review And Preview. *Physics in Medicine and Biology*, 51(13):R363, 2006.

[19] N. G. Burnet, S. J. Thomas, K. E. Burton, and S. J. Jefferies. Defining the Tumour and Target Volumes for Radiotherapy. *Caner Imaging*, 4(2):153–161, 2004.

[20] M. Ehrgott, C. Guler, H. Hamacher, and L. Shao. Mathematical Optimization in Intensity Modulated Radiation Therapy. *4OR*, 6(3):199–262, 2008.

[21] G. K. Bahr, J. G. Kereiakes, H. Horwitz, R. Finney, J. Galvin, and K. Goode. The Method of Linear Programming Applied to Radiation Treatment Planning. *Radiology*, 91(4):686–693, 1968. PMID: 5677503.

[22] D. Shepard, M. Ferris, G. Olivera, and T. Mackie. Optimizing the Delivery of Radiation Therapy to Cancer Patients. *SIAM Review*, 41(4):721–744, 1999.

[23] G. J. Lim, M. C. Ferris, S. J. Wright, D. M. Shepard, and M. A. Earl. An Optimization Framework for Conformal Radiation Treatment Planning. *INFORMS Journal on Computing*, 19(3):366–380, 2007.

[24] Paul S. Cho, Shinhak Lee, Robert J. Marks, Seho Oh, Steve G. Sutlief, and Mark H. Phillips. Optimization of Intensity Modulated Beams with Volume Constraints Using Two Methods: Cost Function Minimization and Projections onto Convex Sets. *Medical Physics*, 25(4):435–443, 1998.

[25] Spiridon V. Spirou and Chen-Shou Chui. A Gradient Inverse Planning Algorithm with Dose-volume Constraints. *Medical Physics*, 25(3), 1998.

[26] Qiuwen Wu and Radhe Mohan. Algorithms and Functionality of an Intensity Modulated Radiotherapy Optimization System. *Medical Physics*, 27(4), 2000.

[27] M. Alber, G. Meedt, F. Nusslin, and R. Reemtsen. On the Degeneracy of the IMRT Optimization Problem. *Medical Physics*, 29(11):2584–2589, 2002.

[28] J. Llacer, N. Agazaryan, T. D. Solberg, and C. Promberger. Degeneracy, Frequency Response and Filtering in IMRT Optimization. *Physics in Medicine and Biology*, 49(13):2853, 2004.

[29] J. O. Deasy. Multiple Local Minima in Radiotherapy Optimization Problems with DoseâĂŞvolume Constraints. *Medical Physics*, 24(7), 1997.

[30] C. Borgers and E. T. Quinto. On the Non-uniqueness of Optimal Radiation Treatment Plans. *Inverse Problems*, 15(5):1115, 1999.

[31] Q. Wu and R. Mohan. Multiple Local Minima in IMRT Optimization Based on Dose-volume Criteria. *Medical Physics*, 29(7), 2002.

[32] J. Llacer, J. O. Deasy, T. R. Bortfeld, T. D. Solberg, and C. Promberger. Absence of Multiple Local Minima Effects in Intensity Modulated Optimization with Dose-volume Constraints. *Physics in Medicine and Biology*, 48(2):183, 2003.

[33] T. Bortfeld and W. Schlegel. Optimization of Beam Orientations in Radiation Therapy: Some Theoretical Considerations. *Physics in Medicine and Biology*, 38(2):291, 1993.

[34] A. B. Pugachev, A. L. Boyer, and L. Xing. Beam Orientation Optimization in Intensity-Modulated Radiation Treatment Planning. *Medical Physics*, 27(6), 2000.

[35] B. Barney. Introduction to Parallel Computing. `https://computing.llnl.gov/tutorials/parallel_comp`. Accessed: 2014-07-01.

[36] U. Diewald, T. Preußer, M. Rumpf, and R. Strzodka. Diffusion Models and Their Accelerated Solution in Computer Vision Applications. *Acta Mathematical Universitatis Comenianae*, 70(1):15–31, 2001.

[37] M. J. Harris and W. V. Baxter. Simulation of Cloud Dynamics on Graphics Hardware. In *ACM AIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, 2003.

[38] J. D. Hall and J. C. Hart. Abstract GPU Acceleration of Iterative Clustering. The ACM Workshop on General Purpose Computing on Graphics Processors, and SIGGRAPH 2004, Aug. 2004.

[39] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph.*, 22(3):777–786, 2003.

[40] C. Men, X. Gu, D. Choi, H. Pan, A. Majumdar, and S. B. Jiang. GPU-based Ultra-fast Dose Calculation Using a Finite Size Pencil Beam Model. *Physics in medicine and biology*, 54:6287–6297, 2009.

[41] J. Reese. Solution Methods for the p-median Problem: an Annotated Bibliography. *Networks*, 48(3):125–142, 2006.

[42] A. A. Kuehn and M. J. Hamburger. A Heuristic Program for Locating Warehouses. *Management Science*, 9(3):643–666, 1963.

[43] F. E. Maranzana. On the Location of Supply Points to Minimize Transport Costs. *Operations Research Quarterly*, 15(3):261–270, 1964.

[44] A. W. Neebe and M. R. Rao. A Subgradient Approach to the m-median Problem. Technical Report 75-12, University of North Carolina, Chapel Hill, N.C., 1975.

[45] P. Hansen and N. Mladenović. Variable Neighborhood Search for the p-median. *Location Science*, 5(4):207–226, 1997.

[46] O. Alp, E. Erkut, and Z. Drezner. An Efficient Genetic Algorithm for the p-median Problem. *Annals of Operations Research*, 122:21–42, 2003.

[47] M.B. Teitz and P. Bart. Heuristic Methods for Estimating the Generalized Vertex Median of a Weighted Graph. *Operations Research*, 16(5):955–961, 1968.

[48] G. Lim, J. Reese, and A. Holder. Fast and Robust Techniques for the Euclidean p-median Problem with Uniform Weights. *Computers & Industrial Engineering*, 57:896–905, 2009.

[49] A. Holder, G. Lim, and J. Reese. The Relationship Between Discrete Vector Quantization and the p-median Problem. Technical Report 102, Trinity University, San Antonio, TX, 2007.

[50] F. G. Lopez, B. M. Batista, J. A. M. Perez, and J. M. M. Vega. The Parallel Variable Neighborhood Search for the p-Median Problem. *Journal of heuristics*, 8(3):375–388, 2002.

[51] T. G. Crainic, M. Gendreau, P. Hansen, and N. Mladenović. Cooperative Parallel Variable Neighborhood Search for the p-Median. *Journal of Heuristics*, 10(3):293–314, 2004.

[52] F. Cao, A. Tung, and A. Zhou. Scalable Clustering Using Graphics Processors . *Lecture Notes in Computer Science*, 4016:372–384, 2006.

[53] A.Myojyoyama and H. Saitoh. Monte Carlo Dose Calculation using GPU-Based parallel processing . In *World Congress on Medical Physics and Biomedical Engineering*, volume 25, pages 704–707, SEPTEMBER 2009.

[54] S. Hissoiny and B. Ozell. A Convolution-superposition Dose Calculation Engine for GPUs. *Medical physics*, 37(3):1029–1037, 2010.

[55] C. Men, X.Gu, D. Choi, A. Majumdar, Z. Zheng, K. Mueller, and S. Jiang. GPU-based Ultrafast IMRT Plan Optimization. *Physics in Medicine and Biology*, 54(21):6565, 2009.

[56] J. A. J. Hall. Towards a Practical Parallelization of the Simplex Method. *Computational Management Science*, 7(2):139–170, 2010.

[57] S. A. Zenios. Parallel Numerical Optimization, Current Status and Annotated Bibliography. *ORSA Journal on Computing*, 1(1):20–43, 1989.

[58] J. Eckstein. Data Parallel Implementations of Dense Simplex Methods on the Connection Machine CM2. *ORSA Journal on Computing*, 7(4):402–416, 1995.

[59] S. A. Zenios. Parallel Implementation of a Sparse Simplex Algorithm on MIMD Distributed Memory Computers. *Journal of Parallel and Distributed Computing*, 31(1):25–40, 1995.

[60] J. A. J. Hall and K. I. M. McKinnon. PARSMI, a Parallel Revised Simplex Algorithm Incorporating Minor Iterations and Devex Pricing. *Applied Parallel Computing*, 1184:67–76, 1996.

[61] R. E. Bixby and A. Martin. Parallelizing the Dual Simplex Method. *INFROMS Journal on Computing*, 12:45–56, 2000.

[62] D. G. Spampinato. Linear Optimization with CUDA. Technical report, Norwegian University of Science and Technology, 2009.

[63] M. E. Lalami. Efficient Implementation of the Simplex Method on a CPU-GPU System. Technical report, University de Toulouse, 2011.

[64] J. Bieling. An Efficient GPU Implementation of the Revised Simplex Method. Technical report, University of Bonn, Germany, 2010.

[65] X. Meyer. A Multi-GPU Implementation and Performance Model for the Standard Simplex Method. Technical report, University of Applied Science of Western Switzerland, 2011.

[66] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[67] I Boussaid, J. Lepagnot, and P. Siarry. A survey on Optimization Metaheuristics. *Information Sciences*, 237(0):82 – 117, 2013.

[68] C Koulamas, SR Antony, and R Jaen. A Survey of Simulated Annealing Applications to Operations Research Problems. *Omega*, 22(1):41 – 56, 1994.

[69] N. Mladenović and P. Hansen. Variable Neighborhood Search. *Computers & Operations Research*, 24(11):1097 – 1100, 1997.

[70] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449 – 467, 2001.

[71] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Number 2. Addison-Wesley, Reading, MA, 1989.

[72] E. Cantu-Paz. A Summary of Research on Parallel Genetic Algorithms. Technical Report 95007, Illinois Genetic Algorithms Laboratory, 1995.

[73] E. Cantu-Paz. Designing Efficient Master-Slave Parallel Genetic Algorithms. Technical Report 97004, Illinois Genetic Algorithms Laboratory, 1997.

[74] E. Onbasoglu and L. Ozdamar. Parallel Simulated Annealing Algorithms in Global Optimization. *Journal of Global Optimization*, 19(1):27–50, 2001.

[75] O. C. L. Haas, K. J. Burnham, and J. A. Mills. Optimization of Beam Orientation in Radiotherapy Using Planar Geometry. *Physics in Medicine and Biology*, 43(8):2179, 1998.

[76] Q Hou, J. Wang, Y. Chen, and J. M. Galvin. Beam Orientation Optimization for IMRT by a Hybrid Method of the Genetic Algorithm and the Simulated Dynamics. *Medical Physics*, 30(9):2360–2367, 2003.

[77] D. Djajaputra, Q. Wu, Y. Wu, and R. Mohan. Algorithm and Performance of a Clinical IMRT Beam-angle Optimization System. *Physics in Medicine and Biology*, 48(19):3191, 2003.

[78] Y. Li, J. Yao, and D. Yao. Automatic Beam Angle Selection in IMRT Planning Using Genetic Algorithm. *Physics in Medicine and Biology*, 49(10):1915, 2004.

[79] R. A. Whitaker. A Fast Algorithm for the Greedy Interchange of Large-scale Clustering and Median Location Problems. *INFOR*, 21(2):95–108, 1983.

[80] P. Hansen and N. Mladenović. Variable Neighborhood Search for the p-median. *Location Science*, 5(4):207–226, 1997.

[81] M. Harris. Optimizing Parallel Reduction in CUDA. Technical report, NVIDIA Developer Technology, 2007.

[82] J. E. Beasley. A Note on Solving Large p-median Problems. *European Journal of Operational Research*, 21:270–273, 1985.

[83] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[84] G. Lim and W. Cao. A Two-phase Method for Selecting IMRT Treatment Beam Angles: Branch-and-Prune and Local Neighborhood Search. *European Journal of Operational Research*, 217(3):609–618, 2012.

[85] R. E. Bixby. Implementing the Simplex Method, the Initial Basis. Technical Report TR90-32, Department of Mathematical Sciences, Rice University, 1995.

[86] A. Swietanowski. A New Steepest Edge Approximation for the Simplex Method for Linear Programming. *Computational Optimization and Applications*, 10:271–281, 1998.

[87] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A Practical Anti-cycling Procedure for Linearly Constrained Optimization. *Mathematical Programming*, 45:437–474, 1989.

[88] J. Forrest and D. Goldfarb. Steepest-edge Simplex Algorithms for Linear Programming. *Math. Program.*, 57(3):341–374, December 1992.

[89] E. D. Andersen and K. D. Anderson. Presolving in Linear Programming. *Mathematical Programming*, 71:221 –245, 1995.

[90] Nathan Bell and Michael Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. `http://cusp-library.googlecode.com`, 2012. Version 0.3.0.

[91] S. S. Sane and N. A. Deshpande. *Data Structures and Algorithms*. Technical Publication Pune, Pune, India, 1st edition, 2006.

[92] W. Orchard-Hays. *Advanced Linear-programming Computing Techniques*. McGraw-Hill, 1968.

[93] E. Cesar, A. Moreno, J. Sorribes, and E. Luque. Modeling Master/Worker Applications for Automatic Performance Tuning. *Parallel Computing*, 32(7âĂŞ8):568 – 589, 2006. Algorithmic Skeletons.

[94] X. Wu and Y. Zhu. A Mixed-encoding Genetic Algorithm with Beam Constraint for Conformal Radiotherapy Treatment Planning. *Medical Physics*, 27(11), 2000.

[95] Y. Li, J. Yao, and D. Yao. Automatic Beam Angle Selection in IMRT Planning Using Genetic Algorithm. *Physics in Medicine and Biology*, 49(10):1915, 2004.

Appendices

Appendix A

Computational results for pVS

Table A.1: pVS performance on OR-lib p-median test problems $p = 0.1 * n$

Problem ID	n	Edge	p	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
2	100	200	10	4069	4069	0.010	0.000	N/A
3	100	200	10	4250	4250	0.010	0.000	N/A
8	200	800	20	4410	4410	0.180	0.010	18.0
13	300	1800	30	4357	4357	0.690	0.020	34.5
18	400	3200	40	4643	4643	2.480	0.070	35.4
23	500	5000	50	4503	4503	5.900	0.170	34.7
28	600	7200	60	4447	4447	12.100	0.330	36.7
33	700	9800	70	4628	4628	24.580	0.680	36.1
37	800	12800	80	4986	4986	39.820	1.090	36.5
40	900	16200	90	5055	5055	0.630	0.010	63.0

Table A.2: pVS performance on OR-lib p-median test problems $p = 0.2 * n$

Problem ID	n	Edge	p	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
4	100	200	20	2999	2999	0.030	0.000	N/A
9	200	800	40	2709	2709	0.500	0.020	25.0
14	300	1800	60	2919	2919	2.100	0.060	35.0
19	400	3200	80	2823	2823	8.130	0.220	37.0
24	500	5000	100	2892	2892	19.270	0.510	37.8
29	600	7200	120	3006	3006	38.680	1.010	38.3
34	700	9800	140	2939	2939	65.890	1.750	37.7

Table A.3: pVS performance on OR-lib p-median test problems $p = 0.33 * n$

Problem ID	n	Edge	p	Objective value		Solution time		Speedup
				VS	pVS	VS	pVS	
5	100	200	33	1355	1355	0.060	0.000	N/A
10	200	800	67	1247	1247	0.880	0.030	29.3
15	300	1800	60	2919	2919	2.100	0.060	35.0
20	400	3200	133	1781	1781	14.140	0.400	35.4
25	500	5000	167	1828	1828	36.460	1.000	36.5
30	600	7200	200	1966	1966	76.840	2.000	38.4

Table A.4: Average pVS performance on randomly generated networks $p = 5$

Problem ID	N	p	Objective Gap	Average Solution Time		Speedups
			$\lvert z_{pvs} - z_{vs} \rvert$	VS	pVS	
1	100	5	0	0.003	0.002	1.5
6	200	5	0	0.021	0.001	20.9
11	300	5	0	0.058	0.003	19.3
16	400	5	0	0.091	0.004	22.8
21	500	5	0	0.189	0.014	13.5
26	600	5	0	0.249	0.014	17.8
31	700	5	0	0.373	0.018	20.7
35	800	5	0	0.468	0.020	23.4
38	900	5	0	0.776	0.030	25.9

Table A.5: Average pVS performance on randomly generated networks $p = 10$

Problem ID	n	p	Objective Gap	Average Solution Time		Speedups
			$\lvert z_{pvs} - z_{vs} \rvert$	VS	pVS	
2	100	10	0	0.015	0.003	5.0
3	100	10	0	0.017	0.001	17.0
7	200	10	0	0.069	0.005	13.8
12	300	10	0	0.159	0.007	22.7
17	400	10	0	0.340	0.014	24.3
22	500	10	0	0.569	0.021	27.1
27	600	10	0	0.997	0.031	32.2
32	700	10	0	1.228	0.047	26.1
36	800	10	0	1.633	0.053	30.8
39	900	10	0	2.377	0.064	37.1

Table A.6: Average pVS performance on randomly generated networks $p = 0.1 * n$

Problem ID	n	p	Objective Gap	Average Solution Time		Speedups
			$\lvert z_{pvs} - z_{vs} \rvert$	VS	pVS	
2	100	10	0	0.015	0.003	5.0
3	100	10	0	0.017	0.001	17.0
8	200	20	0	0.214	0.010	21.4
13	300	30	0	1.167	0.036	32.4
18	400	40	0	3.637	0.102	35.7
23	500	50	0	8.681	0.223	38.9
28	600	60	0	18.672	0.454	41.1
33	700	70	0	35.017	0.827	42.3
37	800	80	0	62.946	1.426	44.1
40	900	90	0	100.915	2.223	45.4

Table A.7: Average pVS performance on randomly generated networks $p = 0.2 * n$

Problem ID	n	p	Objective Gap	Average Solution Time		Speedups
			$\lvert z_{pvs} - z_{vs} \rvert$	VS	pVS	
4	100	20	0	0.042	0.003	14.0
9	200	40	0	0.648	0.027	24.0
14	300	60	0	3.436	0.093	36.9
19	400	80	0	12.163	0.306	39.7
24	500	100	0	27.503	0.649	42.4
29	600	120	0	59.624	1.383	43.1
34	700	140	0	112.597	2.453	45.9

Table A.8: Average pVS performance on randomly generated networks $p = 0.33 * n$

Problem ID	n	p	Objective Gap	Average Solution Time		Speedups
			$\lvert z_{pvs} - z_{vs} \rvert$	VS	pVS	
5	100	33	0	0.084	0.004	21.0
10	200	67	0	1.480	0.054	27.4
15	300	100	0	7.577	0.201	37.7
20	400	133	0	24.565	0.607	40.5
25	500	167	0	61.584	1.386	44.4
30	600	200	0	128.341	2.955	43.4