**Alma Mater Studiorum - Università di Bologna**

DOTTORATO DI RICERCA IN

INFORMATICA

Ciclo XXVII

**Settore Concorsuale di afferenza**: 01/B1

**Settore Scientifico disciplinare**: INF/01

# MANY-CORE ALGORITHMS FOR COMBINATORIAL OPTIMIZATION

**Presentata da**: Francesco Strappaveccia

**Coordinatore Dottorato**
Prof. Paolo Ciaccia

**Relatore**
Prof. Vittorio Maniezzo

**Esame finale anno 2015**

*“May the wind under your wings bear you where the sun sails and the moon walks”*

J. R. R. Tolkien

UNIVERSITY OF BOLOGNA

# *Abstract*

Facoltà di Scienze Naturali, Fisiche e Matematiche
DISI, Dipartimento di Informatica, Scienze ed Ingegneria

Doctor of Philosophy

**Many-core Algorithms for Combinatorial Optimization**

by Francesco STRAPPAVECCIA

Combinatorial Optimization is becoming ever more crucial, in these days. From natural sciences to economics, passing through urban centers administration and personnel management, methodologies and algorithms with a strong theoretical background and a consolidated real-word effectiveness is more and more requested, in order to find, quickly, good solutions to complex strategical problems. Resource optimization is, nowadays, a fundamental ground for building the basements of successful projects. From the theoretical point of view, CO rests on stable and strong foundations, that allow researchers to face ever more challenging problems. However, from the application point of view, it seems that the rate of theoretical developments cannot cope with that enjoyed by modern hardware technologies, especially with reference to the one of processors industry.

We are witnessing a technological 'golden era', where enormous amount of computational capabilities is available at an affordable cost, even at the consumer market level. Is also true that, to fully exploit these devices, a complete focus shift is necessary in thinking and approaching optimization problems. In this work we propose new parallel algorithms, designed for exploiting the new parallel architectures available.

In our research, we found that, exposing the inherent parallelism of some resolution techniques (like Dynamic Programming), the computational benefits are remarkable, lowering the execution times by more than an order of magnitude, and allowing to address instances with dimensions not possible before.

We approached four CO's notable problems: Packing Problem, Vehicle Routing Problem, Single Source Shortest Path Problem and a Network Design problem.

For each of these problems we propose a collection of effective parallel solution algorithms, either for solving the full problem (Guillotine Cuts and SSSPP) or for enhancing a fundamental part of the solution method (VRP and ND).

We endorse our claim by presenting computational results for all problems, either on standard benchmarks from the literature or, when possible, on data from real-world applications, where speed-ups of one order of magnitude are usually attained, not uncommonly scaling up to 40 X factors.

# *Acknowledgements*

I would like to thank first of all Professor Vittorio Maniezzo for his support during this journey. Dott. Marco Antonio Boschetti for his precious theoretical and human support. Prof. Marcus Poggi and Prof. Geir Hasle for their advices and punctual observations on my work. My family, for their love, since the beginning. Sara for her love, patience and understanding in the deepest dejection and in the brightest joy (this would not have been possible without you). Alessandro, for his sincere and lasting friendship, over the last ten years. Francesco, Stefano and Rossano for the sincere and "politically correct" friendship. Matteo and Samuele for their friendship in this foreign territory from via Machiavelli to the "TwinPeaks-TriDomino-Jenga-Poker" evenings. Elena for every advice, suggestion (scientific or not) and lunch together.

# Contents

**Abstract** iv

**Acknowledgements** vi

**Contents** vii

**List of Figures** xi

**List of Tables** xiii

**1 Introduction** 1

**2 High Performance Computing** 5

2.1 Definition . . . . . 5
2.2 Main fields of application . . . . . 6
2.2.1 Simulation . . . . . 6
2.2.2 Bioinformatics . . . . . 8
2.2.3 Large Scale Data Visualization and Management . . . . . 9
2.2.4 Combinatorial Optimization . . . . . 10
2.3 High Performance Computing Programming Models . . . . . 11
2.3.1 Message Passing (**MPI**) . . . . . 11
2.3.1.1 MPI Execution Model . . . . . 12
2.3.2 Shared Memory (**OpenMP**) . . . . . 14
2.3.2.1 OpenMP Execution Model . . . . . 15
2.3.3 GPGPU Computing (**CUDA, OpenCL**) . . . . . 17
2.3.3.1 NVIDIA CUDA . . . . . 18
2.3.3.2 OpenCL . . . . . 20

**3 Combinatorial Optimization** **23**

3.1 Definition . . . . . 23
3.2 Linear Problems (**LP**) . . . . . 24
3.3 Integer Linear Problems (**ILP**) . . . . . 25
3.4 Nonlinear Problems (**NPL**) . . . . . 25

3.5 Constraints Satisfaction Problems (**CSP**) . . . . . . . . . . . . . . . . 26
3.6 Solution and Approximation Methodologies . . . . . . . . . . . . . . 27
3.6.1 Simplex Algorithm . . . . . . . . . . . . . . . . . . . . . . . 27
3.6.2 Lagrangean Relaxation . . . . . . . . . . . . . . . . . . . . . 28
3.6.3 Column Generation . . . . . . . . . . . . . . . . . . . . . . . 29
3.6.4 Dynamic Programming . . . . . . . . . . . . . . . . . . . . . 30
3.6.5 Branch and Bound, Branch and Cut Techniques . . . . . . . 31

**4 Rectangular Knapsack, 2D Unconstrained Guillotine Cuts 33**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.2 The 2D-GCP: notation, definitions and algorithms . . . . . . . . . . 35
4.2.1 Principle of Normal Patterns . . . . . . . . . . . . . . . . . . 36
4.2.2 A Dynamic Programming algorithm for the 2D-GCP . . . . 37
4.3 The GPU porting of the dynamic programming algorithm . . . . . 39
4.3.1 Exploiting the inherent parallelism . . . . . . . . . . . . . . 39
4.3.2 Recurrence Parallelization on GPU . . . . . . . . . . . . . . 40
4.3.3 Parallel Reduction for the Max Operation . . . . . . . . . . 42
4.3.4 Matrix Update . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.3.5 GPU Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.4 Computational results . . . . . . . . . . . . . . . . . . . . . . . . 47
4.4.1 Test instances . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4.2 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.5 Considerations and Future Work . . . . . . . . . . . . . . . . . . . 53

**5 Vehicle Routing Problem 55**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.2 Problem Description and Mathematical Formulations . . . . . . . . 57
5.2.1 Two Index Formulation . . . . . . . . . . . . . . . . . . . . . 58
5.2.2 Set Partitioning Formulation . . . . . . . . . . . . . . . . . . 59
5.3 Dynamic Programming Relaxations for the Pricing Problem . . . . 59
5.3.1 q-Route Relaxation . . . . . . . . . . . . . . . . . . . . . . . 60
5.3.2 through-q-Route Relaxation . . . . . . . . . . . . . . . . . . 62
5.3.3 ng-Route Relaxation . . . . . . . . . . . . . . . . . . . . . . 65
5.4 Parallel Relaxations on GPU . . . . . . . . . . . . . . . . . . . . . 68
5.4.1 GPU q-Route . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.4.1.1 Exposing Parallelism . . . . . . . . . . . . . . . . . 69
5.4.1.2 Algorithm Description . . . . . . . . . . . . . . . . 70
5.4.2 GPU through-q-route . . . . . . . . . . . . . . . . . . . . . . 73
5.4.2.1 Exposing Parallelism . . . . . . . . . . . . . . . . . 73
5.4.2.2 Algorithm Description . . . . . . . . . . . . . . . . 73
5.4.3 GPU ng-Route . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.4.3.1 Exposing Parallelism . . . . . . . . . . . . . . . . . 75
5.4.3.2 Dominance Management . . . . . . . . . . . . . . . 76
5.4.3.3 Active Sets . . . . . . . . . . . . . . . . . . . . . . 78

5.4.3.4 Algorithm Description . . . 78
5.4.4 Asymmetric Relaxations . . . 80
5.5 Computational Results . . . 84
5.6 Considerations and Future Work . . . 89

**6 Single Source Shortest Path Problem 91**
6.1 Introduction . . . 91
6.2 Problem Definition . . . 92
6.2.1 Single Source Shortest Path Problem . . . 93
6.2.2 Multi-Modal Networks . . . 96
6.2.3 Label Constrained Shortest Path Problem . . . 97
6.2.4 Time-Dependent Networks . . . 102
6.3 Serial Algorithm . . . 104
6.3.1 Time-Dependent Augmented Dijkstra Algorithm . . . 105
6.4 GPU Algorithm . . . 106
6.4.1 GPU Dijkstra Algorithm . . . 106
6.4.1.1 Frontier Set . . . 106
6.4.1.2 GPU Implementation . . . 108
6.4.2 Dynamic Frontier Definition . . . 109
6.4.3 GPU Time-Dependent Algorithm . . . 110
6.5 Shared Memory Algorithm . . . 112
6.5.1 OpenMP Porting . . . 114
6.6 Computational Results . . . 115
6.6.1 Data Sets Creation . . . 115
6.6.2 Results . . . 116
6.7 Considerations and Future Work . . . 121

**7 Membership Overlay Problem 123**
7.1 Introduction . . . 123
7.2 Membership Overlay Problem . . . 124
7.2.1 Definition and Mathematical Formulation . . . 128
7.2.2 MIP Formulation . . . 129
7.3 Linear and Lagrangean SMOP Relaxations . . . 130
7.3.1 LP Relaxation . . . 130
7.3.2 Lagrangean Relaxation . . . 130
7.4 Subgradient Algorithm . . . 131
7.5 Distributed Subgradient . . . 132
7.5.1 Algorithm foundations . . . 134
7.5.2 Quasi-constant step size update . . . 135
7.5.3 Algorithm description . . . 136
7.6 Computational Results . . . 137
7.7 Shared Memory Algorithm . . . 140
7.8 Distributed Memory Algorithm . . . 142
7.9 GPU Algorithm . . . 143

7.10 Computational Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 147
7.11 Considerations and Future Work . . . . . . . . . . . . . . . . . . . . . . 149

**8 Conclusions** **151**

**Bibliography** **153**

# List of Figures

2.1 OpenMP logo. . . . . 14
2.2 Fork-Join Execution Model. . . . . 16
2.3 CUDA logo. . . . . 18
2.4 CUDA execution and memory models. . . . . 19
2.5 OpenCL logo. . . . . 20

4.1 Computation of function $V(x, y)$. . . . . 38
4.2 Parallel tasks executed exploiting the decomposition based on *anti-diagonals*. . . . . 40
4.3 Reduction Tree. . . . . 42
4.4 Naive reduction. . . . . 42
4.5 Naive reduction gives rise to bank conflicts (each column of the shared memory represents a memory bank). . . . . 43
4.6 Strided access allows a fast reduction. . . . . 43
4.7 Strided access avoids bank conflicts. . . . . 43
4.8 Matrix update in the GPU computing approach proposed . . . . . 45
4.9 randcut6000c source. . . . . 53
4.10 randcut6000c solution. . . . . 53
4.11 testcut8000 solution. . . . . 54
4.12 gcut13 solution. . . . . 54

5.1 f(q,i) q-path computation and 2-vertices loops avoiding. . . . . 63
5.2 ng-Path example. . . . . 68
5.3 Stage q parallel computation. . . . . 70
5.4 NG path 3-D states space. . . . . 77

6.1 Types of Routes. . . . . 97
6.2 Automaton managing four states. . . . . 100
6.3 Speed Profiles. . . . . 103
6.4 Test Path in Berlin . . . . . 116
6.5 Serial times vs OpenMP times for the Berlin instance . . . . . 117
6.6 Speed-Up for different h values for the Los Angeles instance . . . . . 117

7.1 P2P network with 4 nodes. . . . . 125
7.2 Edges bandwidth values. . . . . 126
7.3 Edges p' values. . . . . 126
7.4 Edges lower bound $l$. . . . . 126

7.5 Edges upper bound $u$. . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
7.6 MOP Optimal Solution. . . . . . . . . . . . . . . . . . . . . . . . . . . 127
7.7 Nodes deploy in a eight core processor. . . . . . . . . . . . . . . . . . . 141
7.8 Performances evaluation. . . . . . . . . . . . . . . . . . . . . . . . . . 149

# List of Tables

2.1 CUDA Memory Hierarchy. . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.1 Computational results on *gcut* instances. . . . . . . . . . . . . . . . . . 50
4.2 Computational results on *testcut* instances. . . . . . . . . . . . . . . . . 51
4.3 Computational results on *randcut* instances. . . . . . . . . . . . . . . . 52

5.1 Computational results for the symmetric q-path and through-q-route relaxations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.2 Computational results for the asymmetric q-path and through-q-route relaxations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.3 Computational results for the symmetric ng-path relaxation. . . . . 87
5.4 Computational results for the asymmetric ng-path relaxation. . . . 88

6.1 Test instances dimensions . . . . . . . . . . . . . . . . . . . . . . . . . 118
6.2 Computational results on CPU . . . . . . . . . . . . . . . . . . . . . . 119
6.3 Computational results on GPU . . . . . . . . . . . . . . . . . . . . . . 120

7.1 Gaps and execution times for **LP**, **STD**, **DIST**. . . . . . . . . . . . . 139
7.2 Speed-Ups of the CUDA, MPI and OpenMP algorithms. . . . . . . 148

*To my family.*

*To Vera, Marcello and Augusta.*
*To Sara.*

# Chapter 1

# Introduction

The Moore's Law, in the early years, has brought to a turning point the microprocessors industry. In fact, the doubling of the computation capabilities of a CPU every eighteen months was becoming a huge engineering problem, due to the increasing work frequencies, power consumption and expensive cooling systems. For instance, the density of heat on the surface of a high-frequency microprocessor can be compared to the density of heat on the same area on the Sun surface. Thus, the solution has been to decrease the work-frequencies and to add more computation units inside the same silicon die; the Moore's Law then, shifted its prospective from doubling the frequencies to doubling the number of computation units inside the processor, remains still valid.

This paradigm shift forced a massive change in software engineering and in computer science in general. In fact, from building an operative system to the design of new algorithms, the new platforms' high level of parallelism is an ever more important aspect to considerate. Also, the academic research community is every year more interested in the exploitation of this new enormous availability of computing resources, aware that this can be a great opportunity to reach new and significant goals, unexpected until few years ago. On the other hand, to take advantage from these architectures, is required a high level of expertise and a deep knowledge of the features implemented on these devices.

A new kind of parallel Lagrangean Relaxation was analyzed and implemented in [1] related to a network design problem (Membership Overlay Problem), this algorithm achieved a great speed-up improvement by implementing it using three of the main parallel paradigms available (MPI, OpenMP and CUDA). This result encouraged a deeper investigation of this new approach to design optimization algorithms and this topic was chosen as the main focus for this Ph.D. thesis.

Moreover, Combinatorial Optimization is still marginally affected by this new approach and the analysis of the most exploited resolution methods, like Dynamic Programming, Branch and Bound techniques or Column Generation, under this new parallel prospective, is in its infancy. The state-of-the-art micro-processors allow now to treat problems never approached before, due to the data dimensions, to the enormous amount of computations required for their solution or to the computation's granularity.

The contribution of this thesis is a collection of parallel algorithms designed to enhance, sometimes significantly, the execution times of the methods cited above.

We approached some of the most notable problems in CO: The Rectangular Knapsack (Unconstrained 2D Guillotine Cuts Problem) achieving, through a Dynamic Programming algorithm designed for a many-core platform (GPU), a speed-up of 22 X for the biggest instances known in literature and a 30 X speed-up factor for two new sets of instances, generated to test the real effectiveness of the method.

The Vehicle Routing Problem, for which we provided six many-core algorithms for enhancing the relaxations (q-routes, through-q-routes and ng-routes relaxations) relative to the pricing problem inside the Column Generation exact method, based on the Set Partitioning formulation of the problem. In this case, we achieved a maximum speed-up of 40 X for the asymmetric version of the ng-routes relaxation, an average speed-up of 20 X for the q-routes and through-q-routes methods and an average speed-up of 10 X for the ng-routes.

The Single Source Shortest Path Problem, for solving the Earliest Arrival Problem in a Multi-modal, Time-Dependent environment. For this problem, we provide a multi-core (CPU) algorithm, achieving a maximum of 3.5 X speed-up factor on real-word based instances.

A Network Design Problem, the Membership Overlay, relative a the Peer-to-Peer network model. For this problem, we propose a parallel subgradient algorithm for solving the Lagrangean Dual Problem relative to the problem's Lagrangean relaxation, achieving a 29 X speed-up for the bigger instances.

This thesis is structured as follows: in Chapter 2 we will briefly review the High Performance Computing field and the actual most used parallel programming models.

In Chapter 3 we will give some definitions for Combinatorial Optimization problems and we will shortly describe some solution methodologies.

In Chapter 4 we will propose a GPU algorithm for solving the Two Dimension Guillotine Cutting Problem, with computational results and two new sets of instances.

In Chapter 5 we will describe the most effective pricing strategies for Column Generation algorithms for solving some classes of the VRP, proposing the relative parallel algorithms with computational results.

In Chapter 6 we will address the Single Source Shortest Path Problem in a Time-Dependent Multi-Modal environment, proposing two parallel algorithm (CPU and GPU) and a new set of instances.

In Chapter 7 we will propose three parallel subgradients for three platforms, CPU, GPU and cluster, to obtain a valid bound for the Membership Overlay Problem.

In Chapter 8 we will draw the conclusions of our work and give some guidelines for future researches.

# Chapter 2

# High Performance Computing

## 2.1 Definition

By High Performance Computing we mean the use of computers for high throughput computation, for solving large problems, or for getting results faster. High Performance is relative to desktop computers, servers or clusters, characterized by more computation units that can cooperate. Supercomputers were introduced in the 1960s and were designed primarily by Seymour Cray at Control Data Corporation (CDC), and later at Cray Research. While the supercomputers of the 1970s used only a few processors, in the 1990s, machines with thousands of processors began to appear and by the end of the 20th century, massively parallel supercomputers with tens of thousands of "off-the-shelf" or commercial processors were marketed [2].

The design of systems with a massive number of processors generally take one of two paths: in one approach, e.g., in grid computing, the processing power of a large number of computers in distributed, diverse administrative domains, is opportunistically used whenever a computer is available. In another approach, a large number of processors are used in close proximity to each other, for instance, in a computer cluster. The use of multi-core processors combined with centralization is an emerging direction [3]. In the early years of 21st century, another emerging trend is to evaluate the "performance per watt" factor, or the sustainability of a super computer, deserving a separate ranking [4]. The energy consumption of a machine with 100,000 or more cores is obviously relevant. To address also this aspect, the GPU computing or heterogeneous computing seems to be the right way. The most recent super-computer in the Top 500 list [5] are in most cases composed not only by canonical processors but, inside every computation node, by several "accelerators", dramatically more energetically efficient than the canonical CPU.

## 2.2 Main fields of application

The need of such a large amount of computational capabilities is restricted to specific research areas like physics, chemistry, engineering, etc... In the next sections, we will give a brief list of the applications of HPC. But nowadays the parallelism is a common factor that brings together the canonical computer to the most advanced smartphones and the same paradigms, used to exploit the HPC structures, can be applied on smaller systems enhancing their performances, sometimes dramatically.

### 2.2.1 Simulation

Simulation is the imitation of the operation of a real-world process or system over time [6]. The act of simulating something first requires that a model be developed; this model represents the key characteristics or behaviors of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time.

Simulation is used in many fields, such as engineering design, building design, geology, climatology and video games. Simulation can be used to show the eventual real effects of alternative conditions or to predict the behavior of a system (for example, a building during an earthquake or an electronic device like a smartphone). Simulation is also used when the real system cannot be directly studied, because inaccessible, dangerous, not built or simply virtual.

More specifically, a computer simulation models a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables in the simulation, predictions may be made about the behavior of the system. It is a tool to virtually investigate the studied system under changing conditions. Computer simulation has become a useful part for modeling many natural systems in physics, chemistry and biology, as well as in engineering. For example, the effectiveness of exploiting computers for simulating the behavior of a system is particularly useful in the field of network traffic analysis.

Key issues in simulation include acquisition of valid source information about the relevant selection of key characteristics and behaviors, the use of simplifying approximations and assumptions within the simulation, and fidelity and validity of the simulation outcomes. Traditionally, mathematical models are used to formally describe systems, models that attempt to find analytical solutions enabling the prediction of the behavior of the system from a set of parameters and initial conditions. Computer simulation is often used as an adjunct to, or substitution for, modeling systems for which simple closed form analytic solutions are not possible. There are many different types of computer simulation, the common feature they all share is the attempt to generate a sample of representative scenarios for a model in which a complete enumeration of all possible states would be prohibitive or impossible [7]. These are some research areas that uses HPC:

- **Computational Fluid Dynamics**: is a branch of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. Computers are used to perform the calculations required to simulate the interaction of liquids and gases with surfaces defined by boundary conditions. With high-speed supercomputers, better solutions can be achieved. Ongoing research yields software that improves the accuracy and speed of complex simulation scenarios such as transonic or turbulent flows. Initial validation of such software is performed using a wind tunnel with the final validation coming in full-scale testing, e.g. flight tests [8].
- **Computational Chemistry**: is a branch of chemistry that uses principles of computer science to assist in solving chemical problems. It uses the results of theoretical chemistry, incorporated into efficient computer programs, to calculate the structures and properties of molecules and solids [9].
- **Multi Agent Systems**: are systems composed of multiple interacting intelligent agents within an environment. Multi agent systems can be used to solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. Intelligence may include some methodic, functional, procedural or algorithmic search, find and processing approach. The study of multi-agent systems is concerned with the development and

analysis of sophisticated Artificial Intelligence problem-solving and control architectures for both single-agent and multiple-agent systems [10].

- **Computational Astrophysics**: refers to the methods and computing tools developed and used in astrophysics research. It is both a specific branch of theoretical astrophysics and an interdisciplinary field relying on computer science, mathematics, and wider physics. Computational astrophysics is most often studied through an applied mathematics or astrophysics program. Well-established areas of astrophysics employing computational methods include magnetohydrodynamics, astrophysical radiative transfer, stellar and galactic dynamics, and astrophysical fluid dynamics [11].
- **Computational Physics**: is the study and implementation of numerical algorithms to solve problems in physics for which a quantitative theory already exists. It is often regarded as a sub-discipline of theoretical physics but some consider it an intermediate branch between theoretical and experimental physics. It is a subset of computational science (or scientific computing), which covers all of science rather than just physics. Theoretical physicists provide very precise mathematical theory describing how a system will behave. Unfortunately, it is often the case that solving the theory's equations ab initio in order to produce a useful prediction is not practical. This is especially true with quantum mechanics, where only a handful of simple models admit closed-form, analytic solutions. In cases where the equations can only be solved approximately, computational methods are often used [12].

As we can see these scientific disciplines need an enormous computing time due to the intensive and complex kind of calculations required to solving the models on which they are based.

### 2.2.2 Bioinformatics

Bioinformatics is an interdisciplinary field involving disciplines from data mining, machine learning to operational research, that develops and improves methods for storing, retrieving, organizing and analyzing biological data. A major activity in bioinformatics is to develop softwares to generate useful biological knowledge (e.g. GROMACS [13]). Bioinformatics has become a fundamental mean for many areas of biology, mainly the ones characterized by strong mathematical and statistical aspects. In experimental molecular biology for instance, bioinformatics techniques such as image and signal processing, allow extraction of useful results from large amounts of raw data. In the field of genetics and genomics, it aids in sequencing and annotating genomes and their observed mutations [14].

Bioinformatics tools also aid in the comparison of genetic and genomic data, task that otherwise would not be possible, given the enormous amount of data to analyze. Another notable contribution is the simulation and modeling of DNA, RNA and proteins in general as well as molecular interactions, using precise mathematical models, the actual computational resources made available from the microprocessors industry.

The peculiarity of this research field is to design computationally intensive methods to enhance the disciplines mentioned before, influenced by the enormous complexity of the problems treated. Some examples of involved methodologies include: pattern recognition, data mining, machine learning, and visualization. Bioinformatics has brought remarkable contribution in sequence alignment, gene finding, genome assembly, drug design, drug discovery, protein structure alignment, protein structure prediction, prediction of gene expression and protein-protein interactions, genome-wide association studies and the modeling of evolution.

Another challenging goal is to develop software and hardware designed following patterns inspired by the biological word itself or more appropriate to precise biological analysis tasks (networks, processors, etc. . . ).

### 2.2.3 Large Scale Data Visualization and Management

Data visualization is the study of the visual representation of data, meaning information that has been abstracted in some schematic form, including attributes or variables for the units of information [15]. Data visualization and management has become an active area of research, teaching and development, in fact this field become really interesting with the new HPC technologies. Also, considering the enormous quantity of data available nowadays, a correct data interpretation based on solid statistical and mathematical models, has become strategic, from marketing campaigns planning to corporate quantitative analysis. Some related areas are:

- **Data Acquisition** : is the sampling of the real world to generate data that can be manipulated by a computer. Sometimes abbreviated DAQ or DAS, data acquisition typically involves acquisition of signals and waveforms and processing the signals to obtain desired information. The components of data acquisition systems include appropriate sensors that convert any measurement parameter to an electrical signal, which is acquired by data acquisition hardware and stored in digital form [16].
- **Data Analysis**: is the process of studying and summarizing data with the goal to extract useful information. Data analysis is closely related to data mining, but data mining tends to focus on larger data sets with less emphasis on making inference, and often uses data that was originally collected for a different purpose. In statistical applications, is frequent to divide data analysis into descriptive statistics, exploratory data analysis, and inferential statistics (or confirmatory data analysis). For example, the exploratory data analysis focuses on discovering new features in the data, instead confirmatory data analysis aims to endorsing or confuting existing hypotheses [17].
- **Data Mining**: is the process of sorting through large amounts of data and picking out relevant information [18]. It is usually used in business intelligence finance, but is also being used in sciences to extrapolate useful information from enormous data sets (e.g. physics large scale experiments). It has been described as the nontrivial extraction of implicit, previously unknown, and potentially useful information from data [19] and the science of extracting useful information from large data sets or databases [20]. In relation to enterprise resource planning, data mining is the statistical and logical analysis of large sets of transaction data, looking for patterns that can aid decision making.

### 2.2.4 Combinatorial Optimization

Combinatorial Optimization is a field that consists of finding an optimal solution of a constrained problem [21]. In many problems, complete search or solutions enumeration is not possible. CO is focused on dealing with optimization problems, in which the set of feasible solutions is discrete or can be reduced to discrete, and in which the goal is to find the best solution. This field is characterized by dealing with NP-Hard / NP-Complete problems and, obviously, the optimal or near optimal solution of these kind of problems requires a large amount of computing time and large scales data structures to manage the, often, combinatorial explosion of data involved. This field will be treated more deeply in Chapter 3, analyzing and describing various kinds of combinatorial problems and their formulation. Combinatorial Optimization involves some of the research areas early cited. In fact a lot of topics related to physics, chemistry and engineering need the solution of linear or non linear constrained problems (e.g. lattice problems in particles physics).

## 2.3 High Performance Computing Programming Models

In this paragraph we will give a short introduction and description of the "de facto" standards used for the implementation of parallel applications. The spectrum of parallel programming models, indeed, is enormously wide, often bounded to specific types of hardware and software vendors, the three paradigms described in the following are the most used and studied for their portability, effectiveness and generality of model. MPI, OpenMP, CUDA, OpenCL are "de facto" standards because the large part of parallel infrastructures implements these models and the most important parallel libraries (for instance BLAS [22]) and softwares (for instance OpenFOAM [23], Quantum Expresso [24] and many more) are based on these.

### 2.3.1 Message Passing (MPI)

Message Passing Interface (MPI), proposed in 1992 by William Gropp and Ewing Lusk, is a standardized and portable message-passing system designed by a group of researchers from academia and industry to operate on a wide variety of parallel computers [25]. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran 77, Fortran 90 or the C programming languages. Several well-tested and efficient implementations of MPI include some that are free and in the public domain. These fostered the development of a parallel software industry, and there encouraged development of portable and scalable large-scale parallel applications.

MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented. As a result, hardware vendors can build upon this collection of standard low-level routines to create higher-level methods for the distributed-memory communication environment supplied with their parallel machines. MPI provides a simple-to-use portable interface for the basic user, yet powerful enough to allow programmers to use the high-performance message passing operations available on advance machines.

After twenty years MPI is still the most used and effective parallel programming model for clusters. Formally, MPI is defined as a: "message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation" [26]. MPI is not a "de-iure" standard but, due to its portability and well defined behavior and interface, is become the "de facto" standard for distributed memory architectures. MPI is an API referring from 5 to 7 levels of ISO-OSI Communication Stack. The benefit in using MPI is its complete portability in every parallel environment. In fact, every MPI implementation provided by each vendor is optimized for the hardware where the application runs. Moreover, MPI allows the coexistence of portion of other programming languages code inside the same application; common is the hybridization with Open MP in order to exploit easily the shared memory nodes inside the cluster (multi-core CPUs). MPI itself, in any case, allows the programmer to manage a shared memory computation node. In the last years is common also the hybridization with specific language-extensions used to manage the GPU or many-core devices inside the computation node (CUDA or OpenCL).

#### 2.3.1.1 MPI Execution Model

The MPI interface is meant to provide virtual topology, synchronization, and communication functionalities between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec. MPI library functions include, but are not limited to:

- point-to-point,
- rendezvous-type,
- send/receive operations,
- choosing between a Cartesian or graph-like logical process topology,
- exchanging data between process pairs (send/receive operations),
- combining partial results of computations (gather and reduce operations),
- synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session,
- current processor identity that a process is mapped to,
- neighboring processes accessible in a logical topology.

More in detail, the MPI's runtime system creates $n$ processes called tasks and each of these tasks:

- creates an independent copy of the application in the node where is executing,
- has local memory,
- can be mapped on a different processor,
- has an univocal index among the tasks created by the user.

Each process is part of a Communicator. An MPI Communicator is an object connecting groups of processes. Each communicator:

- has a name,
- a cardinality,
- assign to its processes a proper index for the communicator itself,
- arrange the processes in an ordered topology,
- each process is equivalent to the others.

MPI optimizes the deployment of the communicators understanding also when a data transfer or a communication is relative only to a specific communicator.

Each MPI application is a single executable, managed by the run-time system that organizes the communication among the application's processes. The MPI primitives are *blocking* or *non-blocking* for the application execution, obviously the non-blocking are recommended when exploitable. The primitives also can have different modes of communication:

- *Standard*: automatic synchronization and buffering,
- *Buffered*: buffering defined by the user,
- *Synchronous*: strict rendezvous,
- *Ready*: instant communication without hand-shacking.

Actually, the current version of MPI is the 3.0.

### 2.3.2 Shared Memory (OpenMP)

Figure 2.1: OpenMP logo.

OpenMP (Open Multiprocessing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran [27], on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior . OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board (or OpenMP ARB), represented by a group of computer hardware and software vendors: AMD, IBM, Intel, Cray, HP, Fujitsu, NVIDIA, NEC, Microsoft, Texas Instruments, Oracle Corporation, and more [28].

OpenMP is described by a portable, scalable programming model that provides users with a minimal and flexible interface for developing parallel applications for a wide spectrum of machines, from desktop computers to a clusters.

OpenMP obtained a reasonable success due to some interesting characteristics:

- great emphasis on structured parallel programming,
- its modest learning curve. The compiler bears all the most difficult aspects of the shared-memory parallel programming (threads synchronization, etc. . . ),
- portability: OpenMP libraries are available natively for the most common and used programming languages,
- incremental implementation from the serial code by adding only some simple pre-processor's directives.

The most important aspect of OpenMP is the constant development and support from the most important actors of software and hardware industry, making it a 'de facto' standard for the shared memory parallel programming.

The OpenMP's directives allow to programmers to indicate to the compiler which instructions execute in parallel and how to divide the workload among threads. These compiler directives are extremely flexible and don't require to re-write the code in case of platform or compiler changing, besides if a compiler or a platform is not enabled for run OpenMP, the serial code remains the same.

#### 2.3.2.1 OpenMP Execution Model

OpenMP supports the Fork-Join programming model [29]. Under this approach, the program starts as a single thread of execution, just like a sequential program. The thread that executes this code is referred to as the initial thread. Whenever an OpenMP parallel construct is encountered by a thread while it is executing the program, it creates a team of threads (fork), becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct. At the end of the construct, only the original thread, or master of the team, continues; all others terminate (join). Each portion of code enclosed by a parallel construct is called a parallel region. A *thread* is a run-time entity capable to execute independently an instructions stream [30].

More in details, once the operative system executes an OpenMP application:

- a process is created for the program,
- are instantiated the necessary system resources: memory pages and registers.

If more threads are spawned, they will share the resources created by the operative system also the same memory addresses space. Each thread needs only few resources:

- a program counter,
- private memory locations for its specific data (registers and execution stack).

More threads can be executed by the same processor or core through context switching procedures. More threads in a multi-core processor can run in a parallel fashion, properly orchestrated. OpenMP makes transparent a considerable number of interactions among the thread to the programmer, providing a more friendly development environment.

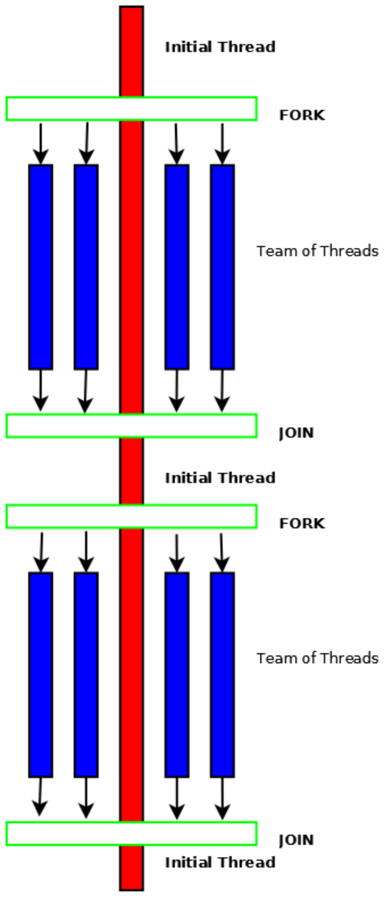


Figure 2.2: Fork-Join Execution Model.

The Fork/Join Execution Model implemented by OpenMP can be described as follows and depicted in figure 2.2:

- the application starts in a serial mode, with only one thread, the initial thread,
- once reached the code sections where the OpenMP directories are located, a team of threads is spawned (Fork),
- the initial thread becomes the master thread and coordinates and cooperate with the other threads in the parallel section,
- at the end of the OpenMP directives, the master thread continues the execution and the other are erased (Join).

In parallel regions, the developer can orchestrate at an higher abstraction level the threads interaction, leaving to the compiler the more low-level implementation details.

The OpenMP model implements, inside the single multi-core/multi-threaded elaboration unit, the same characteristics described for MPI. One of the most used and effective techniques to develop parallel applications on massively parallel architectures is to mix, or hybridizing, the MPI code, exploiting the inter-node communications and workload, with the intra-node or intra-core communications and work-sharing, implemented with OpenMP. This approach guarantees a major scalability of the application. The current release of OpenMP is the 3.0.

### 2.3.3 GPGPU Computing (CUDA, OpenCL)

Since 2004, when Intel decided to cancel the development of its last single core processor in order to concentrate its efforts on multi-core architectures, we have observed a radical change in the design of new generations of CPUs. The focus, indeed, shifted from the increase of the processor's frequency to the implementation of parallel architectures. Following this trend, the market got populated by relatively affordable devices with great, natively parallel, computational resources.

General-purpose computing on graphics processing units (General-purpose graphics processing unit, GPGPU) is the exploitation of graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation commonly handled by the CPU. Moreover, the use of multiple GPUs in the same computer or computation node further parallelizes the already parallel nature of these devices.

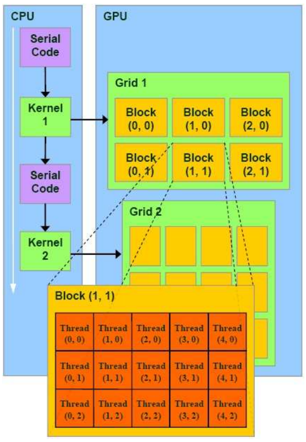
OpenCL is the actually the most used open-source GPU computing language. The proprietary counterpart framework is NVIDIA's CUDA. More in general, this approach is generally called many-core computation, that differs from the ordinary CPU by the fact that a GPU is composed by hundreds of simple computational units, otherwise an ordinary CPU has less but more sophisticated cores. This different prospective in the architecture's design make the GPU the silver bullet for high dense computation that involves thousands and thousands of small entities and a fine grain granularity of computation.

#### 2.3.3.1 NVIDIA CUDA

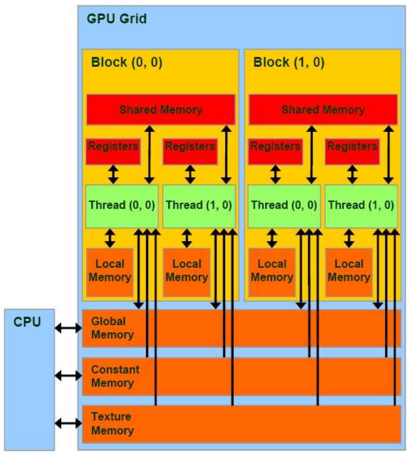
Figure 2.3: CUDA logo.

In 2007, NVIDIA introduced *CUDA* (Compute Unified Device Architecture) [31], a parallel, general purpose programming model designed to exploit the great computational resources available on a GPU. CUDA is implemented as an extension of the C/C++ or Fortran programming languages. NVIDIA *GPU*s follow the *SIMT* (Single Instruction Multiple Threads) execution model, in which the same instruction is executed concurrently on multiple data by different threads.

CUDA programming implements an higher abstraction model than a straight transposition of the hardware architecture of the GPU. The model works at top level on a *kernel*, which is the portion of code executed asynchronously on the GPU, the rest of the user code being directly executed on the CPU. The kernel is executed in parallel on a user-defined number of *threads*. The threads are grouped into *blocks*, which are equal cardinality subsets of threads. The blocks are in turns logically arranged into a *grid*, which is a 1-, 2- or 3-dimensional array of blocks (see Figure 2.4a).



(A) CUDA Execution Model



(B) CUDA Memory Model

FIGURE 2.4: CUDA execution and memory models.

Every block is logically executed on a different *SM* (i.e., *Stream Processor* or *Cluster Processor*) of the GPU, which consists of a set of simple arithmetic cores called *CUDA Cores* (Fermi architecture counts 32 *Cuda Cores* for SM, Kepler 192 and call it SMX and Maxwell 128, calling the processor SMM).

Every block can accept up to 1024 threads [31], even though it will actually simultaneously execute sets of only 32 threads at a time, called *Warps*.

In Figure 2.4a we show in detail this model.

There is a memory hierarchy on these devices, aimed to minimize the communications between the host memory (named *Global Memory* in this context) and the device one, via the PCI-Express bus. In fact, besides the core registries, the GPU has an on-board, low-latency memory module accessible to all threads of each single block. In table 2.1 we summarize the features of CUDA's memory architecture. The access to Global Memory is a significant bottleneck for performance enhancement in the design of a GPU-accelerated application. The Global Memory is the only memory type visible by the entire grid, and sometimes it is needed for the synchronization and data sharing among the blocks, even though its access latency is very high (400-800 memory cycles). There are many techniques used to hide this latency, but they are mainly application-specific, thus often impossible to apply if the peculiar characteristics of the considered algorithm do not allow them. It is therefore crucial to properly manage the access to global memory. The effectiveness of this management is measured by the achieved *memory bandwidth*, which quantifies the amount of relevant data which the application can get from the global memory per second.

Table 2.1: CUDA Memory Hierarchy.

| **Memory Type** | **Scope** | **Lifetime** | **Access** | **Location** | **Cached** |
|---|---|---|---|---|---|
| Global | Grid+host | Application | R-W | Off-chip | No |
| Shared | Block | Kernel | R-W | On-chip | N/A |
| Local | Thread | Thread | R-W | Off-chip | No |
| Register | Thread | Thread | R-W | On-chip | N/A |
| Constant | Grid+host | Application | Read Only | Off-chip | Yes |
| Texture | Grid+host | Application | Read Only | Off-chip | Yes |

#### 2.3.3.2 OpenCL

Figure 2.5: OpenCL logo.

The Open Computing Language (OpenCL) [32] is a heterogeneous programming framework created and sponsored by the nonprofit technology consortium Khronos Group and adopted by some of the most important technology actors like Intel, AMD, NVIDIA, ARM, Apple. OpenCL is a framework for developing applications that execute across a broad spectrum of device types made by different vendors. It supports a wide range of levels of parallelism and efficiently maps to homogeneous or heterogeneous, single or multiple-device systems consisting of CPUs, GPUs, and other types of device (e.g. FPGA, APU. . . ). The OpenCL definition offers both a device-side language (based on C99) and a host management layer composed by ad-hoc designed API, for the devices in a system.

The programming language used to write computation kernels is based on C99 with some limitations and additions. It omits the use of function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files. The language is extended to use parallelism with vector types and operations, synchronization primitives, functions to manipulate work-items/groups. It also implements memory region qualifiers: global, local, constant, and private. To enhance productivity, many built-in functions has been added (e.g. trigonometric functions). One of the most interesting and useful OpenCL's peculiarities is the Device Fission that allows to queue instructions to a specific section or set of cores of the processor. This feature allows to split the device into multiple areas assigned to different tasks, maximizing its utilization or customizing the application for specific types of processors. The actual release of OpenCL is the 2.0.

# Chapter 3

# Combinatorial Optimization

## 3.1 Definition

In mathematics and computer science, an optimization problem is the problem of finding one best among all feasible solutions. Optimization problems can be divided into two categories, depending on whether the variables are continuous or discrete. An optimization problem with discrete variables is known as a combinatorial optimization problem [33]. In a combinatorial optimization problem, we are looking for an object such as an integer, permutation or graph from a finite (or possibly countable infinite) set. As anticipated in the Chapter 1, in this chapter we will explain more in detail the Combinatorial Optimization field. In general, we can describe an optimization problem as:

$$\begin{aligned} z = \min f(x) & \\ s.t.\, g_i(x) \le 0 & \quad i = 1, \dots, m \\ h_i(x) = 0 & \quad i = 1, \dots, p \end{aligned} \tag{3.1}$$

where:

- $f(x) : \mathbb{R}^N \to \mathbb{R}$, is the objective function to be minimized(maximized) over the variable $x$,
- $g_i(x) \le 0$ are the inequality constrains,
- $h_i(x) = 0$ are the equality constrains.

In the next sections we will describe some common problems and some of the most used and known resolution methods.

## 3.2 Linear Problems (LP)

A Linear Problem (LP), can be expressed in this way (canonical form):

$$\begin{aligned} z = \min \quad & \mathbf{c}\mathbf{x} \\ s.t. \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.2}$$

Where:

- $\mathbf{A}$ is the matrix of coefficients that describe the convex set,
- $\mathbf{c}, \mathbf{b}$ are the vector of costs and known coefficients,
- $\mathbf{x}$ is the linear solution.

The problem is characterized by being constrained only by linear inequalities. Another way to describe the problem is:

$$\begin{aligned} z = \min \sum_{j=1}^{n} c_j x_j & \\ s.t. \sum_{j=1}^{n} a_{ij} x_j \leq b_i, & \quad i = 1, \ldots, n \\ x_j \geq 0, & \quad j = 1, \ldots, n \end{aligned} \tag{3.3}$$

Where:

- $a_{ij}$ are the elements of $\mathbf{A}$,
- $b_i$ and $c_j$ are the elements of $\mathbf{b}$ and $\mathbf{c}$ respectively,
- $x_j$ are the elements of the linear solution $\mathbf{x}$

## 3.3 Integer Linear Problems (ILP)

$$\begin{aligned} z = \min \quad & \mathbf{cx} \\ s.t. \quad & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{x\ int} \end{aligned} \tag{3.4}$$

This kind of problems are characterized by another constraint imposing that the solution **x** must be composed only by integer numbers. Two sub-classes of the ILP are:

- **MILP**: where only some $x_j$ variables are integer,
- **Zero-One Problems**: where $x_j \in \{0, 1\}$.

## 3.4 Nonlinear Problems (NPL)

They are distinguished by the presence of non linear constrains inside the system. A simple example can be:

$$\begin{aligned} z = \min \quad & x_1 + x_2 \\ s.t. \quad & x_1^2 + x_2^2 \geq 1 \\ & x_1^2 + x_2^2 \leq 2 \\ & x_1 \geq 0 \\ & x_2 \geq 0 \end{aligned} \tag{3.5}$$

Generally, a non-linear problem can be describe in this way:

$$\begin{aligned} z = \min f(x) & \\ s.t.\, g_i(x) \leq 0 \quad & i \in I = 1, \ldots, m \\ h_j(x) = 0 \quad & j \in J = 1, \ldots, p \end{aligned} \tag{3.6}$$

where:

- $f(x) : \mathbb{R}^N \to \mathbb{R}$, is the objective function to be minimized(maximized) over the variable $x$,
- $x \in \mathbb{R}^N$, that makes the model non-linear,
- $g_i(x) \leq 0$ are the inequality constrains,
- $h_i(x) = 0$ are the equality constrains.

## 3.5 Constraints Satisfaction Problems (CSP)

Constraint Satisfaction arose mainly from Artificial Intelligence. A Constraint satisfaction problem (CSP) is a problem defined as a set of objects that must satisfy a number of constraints or relations among the problem's decision variables [34]. Constraint programming is defined "programming" in a double meaning: not only "mathematical programming", in the sense of declaration of constraints and decision variables, but also in the sense of "computer programming", in the sense of programming a search strategy. The methods used to solve this kind of problems are various: Branch and Bound algorithms, Backtracking, Local Search, typically all embedded in a properly designed solver.

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$, where $X$ is a set of variables, $D$ is a domain of values, and $C$ is a set of constraints. Every constraint is in turn a pair $\langle t, R \rangle$ (usually represented as a matrix), where $t$ is an $n$-tuple of variables and $R$ is an $n$-ary relation on $D$ . An evaluation of the variables is a function from the set of variables to the domain of values, $v : X \to D$. An evaluation $v$ satisfies a constraint $\langle (x_1, \ldots, x_n), R \rangle$ if $(v(x_1), \ldots, v(x_n)) \in R$. A solution is an evaluation that satisfies all constraints [34].

Contraint Programming has been used to solve combinatorial problems like:

- Eight queens puzzle,
- Map coloring problem,
- Sudoku and many other logic puzzles,
- DNA sequencing,
- Scheduling.

Often CP deal with problems really hard to solve using canonical OR or CO techniques.

## 3.6 Solution and Approximation Methodologies

### 3.6.1 Simplex Algorithm

Following some notable theoretical results of linear algebra, it can be shown that for a linear program, if the objective function has a minimum value in the feasible region then it has this value on (at least) one of the extreme points of the convex set described by the inequalities constraining the problem [35]. It's has been also proven that there is a finite number of extreme points in the convex set, but the number of extreme points is extremely large also for small linear programs, making the enumeration of these points inapplicable. It can also be shown that exists an edge that connects an extreme point to another where the objective function decreases, if the starting point isn't a minimum. If the edge is finite, it brings to another extreme point where the objective function has a smaller value, otherwise the objective function is unbounded and the linear program has no solution.

The simplex algorithm proposed by Danzig [36] exploits these results and by walking along edges of the polytope to extreme points with lower and lower objective values, until the minimum value is reached or an unbounded edge is visited, concluding that the problem has no solution. The great contribution given by this algorithm is that it always terminates because the number of vertices in the polytope is finite; moreover, since we jump between vertices always in the same direction (that of the objective function), we hope that the number of vertices visited will be small. The solution of a linear program is accomplished in two steps. In the first step, we need to find an extreme point for starting. The possible results of the first step are either a basic feasible solution is found or that the feasible region is empty. In the latter case the linear program is called infeasible. In the second step, the simplex algorithm is applied using the basic feasible solution found as a starting point. The possible results from the second step are either an optimal feasible solution to the linear program or an infinite edge on which the objective function is unbounded.

Due to the wide spectrum of applications of this algorithm or its variations, commercial and free softwares implementing this method has been proposed. The most used and effective is the IBM ILOG CPLEX solver [37]. It also implements methods for solving MIP problems, Quadratic Problems and Quadratically Constrained Problems. Its free and open source counterpart is the CoinOR, Computational Infrastructure for Operative Research [38] that is a project developed and maintained by the operative research community, aimed to provide a free environment for developing and testing OR algorithms.

### 3.6.2 Lagrangean Relaxation

Lagrangean relaxation proposed by Polyak [39] and used for the first time by Held et al. [40] and Held and Karp [41, 42] and is a relaxation method which approximates a difficult optimization problem by a 'simpler' one. Solving the relaxed problem can give an approximate solution to the original one.

The method penalizes violations of inequality constraints using a Lagrangean multiplier, which imposes a cost on constraints violations in the objective function. In practice, this relaxed problem can often be solved more easily than the original one by using polynomial algorithms like the Subgradient [43], providing useful information for its solution. The problem of maximizing the Lagrangean function of the dual variables is the Lagrangean dual problem (if we are searching for the function's minimum). Under certain conditions regarding function's convexity and constraints, we can state that the solution of primal and dual Lagrangean problems are the same, avoiding the Duality Gap.

Taking as example a linear problem:

$$\begin{aligned} z = \min \quad & \mathbf{cx} \\ s.t. \quad & \mathbf{A}_1\mathbf{x} \leq \mathbf{b}_1 \\ & \mathbf{A}_2\mathbf{x} \leq \mathbf{b}_2 \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.7}$$

where the constraints in $\mathbf{A}_2$ are considered 'difficult'. We can relax the problem adding a penalty $\lambda$ and bringing the constraints in the objective function:

$$
\begin{aligned}
z = \min \quad & \mathbf{c}\mathbf{x} + \lambda(\mathbf{b}_2 - \mathbf{A}_2\mathbf{x}) & \\
s.t. \quad & \mathbf{A}_1\mathbf{x} & \leq \mathbf{b}_1 \\
& \mathbf{x} & \geq \mathbf{0}
\end{aligned}
\tag{3.8}
$$

This is a relaxed problem different from the initial one, but can be used like a bound to the optimal value inside other solution techniques.

### 3.6.3 Column Generation

One of the most difficult aspects related to many linear programs is that the number of all decision variables is too large to be consider explicitly. Since most of the variables will be non-basic and assuming a value of zero in the optimal solution, only a subset of variables need to be considered when solving the problem. Column generation [44, 45] exploits this idea: it generates only the variables which have the potential to improve the objective function, finding variables with negative reduced cost (assuming without loss of generality that the problem is a minimization problem).

The original problem is split into two problems: the master problem and the subproblem. The master problem is composed only by a subset of variables selected from the original problem (core problem). The subproblem is a new problem created to identify a new variable (pricing problem). The objective function of the subproblem is the reduced cost of the new variable with respect to the current dual variables. The process, iteratively, behaves as follows:

- The master problem is solved taking in consideration only the selected variables, then we are able to obtain dual prices for each of the constraints in the master problem. This information is then utilized in the objective function of the subproblem.
- The subproblem is solved and if the objective value of the subproblem is negative, a variable with negative reduced cost has been identified. This variable is then added to the master problem, and the master problem is solved again. Solving the master problem with the new variable, will generate a new set of dual values, and the process is repeated until no negative reduced cost variables are identified. On the other hand, if the subproblem returns a

solution with non-negative reduced cost, we can conclude that the solution to the master problem is optimal.

In many cases, this approach makes tractable large linear or integer programs that had been previously considered intractable. An example of a problem where this technique is effective is the cutting stock problem where the number of all possible feasible cuts is intractable. Additionally, column generation has been applied to many problems such as crew scheduling, vehicle routing, etc . . . .

### 3.6.4 Dynamic Programming

Dynamic programming (DP) is a technique used for solving complex problems by dividing them into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure (e.g. the Shortest Path Problem). When applicable, the method takes far less time than naive methods. The basic idea behind dynamic programming is simple: in general, to solve a given problem, we need to solve different parts of the problem (subproblems), then we combine the solutions provided by the subproblems to compute the global one. Often, many of these subproblems are identical. The dynamic programming approach seeks to solve each subproblem only once, trying to reduce the computations required. Once a subproblem has been solved, its solution is stored.

This kind of algorithms works iteratively: once, during a computation step, a solution computed before is needed, it is simply retrieved from the ones stored. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input. The Dynamic Programming method is based on the theoretical infrastructure based on R. Bellman's principle of optimality [46]: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision". More generally, all DP recursions (or recurrences) are based on the Bellman Equation describing the transitions from a state to another:

$$V(x) = max_{a\in\Gamma(x)} \{F(x,a) + \beta V(T(x,a))\} \tag{3.9}$$

This approach is also used for solving optimization problems or for computing valid bounds (e.g. VRP or Cutting Stock Problem). The main disadvantage of this class of algorithms is the extensive use of memory: the states space created by the recursion is often too big to manage. For some class of problems, like the Knapsack 0-1, the dynamic programming is a very fast and simple solution method, but only for instances with limited dimensions.

### 3.6.5 Branch and Bound, Branch and Cut Techniques

A branch-and-bound (BB) algorithm [47] consists of a systematic enumeration of solutions, where large subsets of fruitless candidates are discarded, typically using bounds to the optimal solution. The search is performed as a tree search or similar approaches. The key idea of the BB algorithm is: if a lower bound for some node is greater than an upper bound for some other node, then may be safely discarded from the search (pruning). Any node whose lower bound is greater than the best lower bound achieved during the search, can be discarded. The branching phase is the generation of other (improved) solutions from a node, then, new bounds are computed on these new solutions (bounding). The search stops when the solution's set is reduced to a single element, or when the upper bound matches the lower bound. Either way, the found value will be the minimum (or the maximum) of the function.

# Chapter 4

# Rectangular Knapsack, 2D Unconstrained Guillotine Cuts

In this chapter we investigate the application of GPU computing to the two-dimensional guillotine cutting problem, using a dynamic programming approach. We show a possible implementation and we discuss a number of technical issues.

Computational results on test instances available in the literature and on new larger instances show the effectiveness of the dynamic programming approach based on GPU computing for this problem.

## 4.1 Introduction

The Two-Dimensional Guillotine Cutting Problem (2D-GCP) consists in cutting a rectangular surface, called *stock rectangle* or *master surface* , into a number of smaller rectangular pieces, each with a given *size* and *value*, using *guillotine cuts.* A guillotine cut on a rectangle consists in a cut from one edge of the rectangle to the opposite edge which is parallel to the two remaining edges. A feasible cutting pattern must be obtained by applying a sequence of guillotine cuts to the original master surface or to the rectangles obtained in the previous cuts.

The number of pieces of each size and value to be cut can be *unconstrained* or *constrained* within a given minimum and maximum value. The objective is to maximize the total value of the pieces cut. The related problem of minimizing the amount of waste produced can be trivially converted into this maximization problem by taking the value of a piece to be proportional to its area.

A special case of the 2D-GCP is the *staged* guillotine cutting, where at each stage we associate a cut direction, i.e., the cuts alternate at each stage between being parallel to the $x$-axis and being parallel to the $y$-axis. In many practical applications the number of stages is restricted and we say that the guillotine cutting is *k-staged*. Often, the number of stages is only two or three (i.e., two- or three-staged guillotine cutting).

The 2D-GCP can be found in several industrial settings. For example, glass plates are cut into smaller pieces to produce windows or wood sheets are cut to produce furniture, and so on.

According to the classification of [48] the 2D-GCP corresponds to the *Two-Dimensional Rectangular Single Large Object Packing Problem*.

The 2D-GCP is a well known problem, it was considered for the first time by Gilmore and Gomory ([49, 50]) who discussed possible mathematical formulations and proposed dynamic programming algorithms to solve the problem for the two- and multi-stage versions.

In [51], Herz proposed a recursive tree-search procedure where the search space is reduced by means of a preliminary discretization using the so-called *canonical dissections*. Moreover, Herz pointed out an error in an algorithm proposed by [50].

Christofides and Whitlock ([52]) considered the constrained version of the 2D-GCP and described a tree-search algorithm, where a valid upper bound is computed by a dynamic programming procedure applied to the unconstrained version of the 2D-GCP. Christofides and Whitlock also used the canonical dissections, but they used the term *normal patterns* for them.

Beasley [53] proposed heuristic and exact algorithms based on dynamic programming to solve the unconstrained version of the 2D-GCP. He considered both the staged version and the general non-staged version of the problem and used the normal patterns to reduce the search space.

Focusing our attention on the approaches for the 2D-GCP based on dynamic programming, recently, Cintra et al. [54] proposed an improved implementation of the algorithm proposed by Beasley [53] and Russo et al. [55] proposed an improved version of the algorithm of Gilmore and Gomory ([50]).

GPU Computing used for optimization has a recent and yet sparse literature. Due to its structure, it seems to be fit to be combined with dynamic programming procedures. In fact Kats et al. [56] and Lund et al. [57] proved the effectiveness of this new paradigm of massively parallel processors on the very well known All-Pairs Shortest Path Problem (APSP) solved with the Floyd-Warshall algorithm. Boyer et al. [58] proved that GPU computing is effective also in the resolution of the Knapsack Problem (KP) using the well-known dynamic programming algorithm proposed by Bellman [46].

This chapter presents one of the first contributions applying GPU computing to state of the art research algorithms. We consider the *unconstrained* and *non-staged* version of the 2D-GCP and we focus on the implementation of a dynamic programming algorithm using a GPU computing approach to gauge the effectiveness of this new paradigm on optimization problems. We have chosen the dynamic programming algorithm proposed by Cintra [54] because is quite clean and it well suited for our purpose. The other best performing approach, namely the algorithm proposed by Russo et al. [55], is surely interesting and effective, but it is quite complex and it may divert the attention of the reader on aspects that are beyond the scope of this paper, which is the implementation of a dynamic programming algorithm using the GPU computing.

We have organized this chapter as follows. In Section 4.2 we describe the problem and we introduce the dynamic programming algorithm used. In Section 4.3 we present the GPU porting of the dynamic programming algorithm for th 2D-GCP. Computational results are reported and discussed in Section 4.4. Finally, in Section 4.5 we draw some conclusions and we give some ideas for future developments.

## 4.2 The 2D-GCP: notation, definitions and algorithms

A large rectangular master surface $M = (W, H)$ of width $W$ and height $H$ must be cut into a number of smaller rectangular pieces chosen from $n$ types of pieces available. Let $P = \{1, \dots, n\}$ be the index set of piece types. Each piece of type

$i \in P$ has dimensions $(w_i, h_i)$ and value $v_i$. The orientation of the pieces is fixed (i.e., no rotations are allowed). The objective is to construct a guillotine cutting pattern of $M$ that maximizes the total value of pieces cut, using the given piece types.

The master surface $M$ is located in the positive quadrant of the Cartesian coordinate system with its origin (bottom left-hand corner) placed in position $(0, 0)$ and with its bottom and left-hand edges parallel to the $x$-axis and the $y$-axis, respectively. The position of a piece within $M$ is defined by the coordinates of its bottom left-hand corner, referred to as the origin of the piece.

### 4.2.1 Principle of Normal Patterns

The origin of a piece of type $i \in P$ can be located at every integer point $(x, y)$ of the master surface, such that $0 \leq x \leq W - w_i$ and $0 \leq y \leq H - h_i$. However, this set of points $(x, y)$ can be reduced by applying the discretization principle proposed by Hertz [51], who - as mentioned - speaks of *canonical dissections*, and used by Christofides [52], who introduced for the first time the term *normal patterns*. The principle of normal pattern has been thereafter used by Beasley [53] and many other authors.

The principle of normal patterns is based on the observation that, in a given feasible cutting pattern, the position where any piece is cut can be moved to the left and/or downward as much as possible until its left-hand edge and its bottom edge are both adjacent to other cut pieces or to the edges of the master surface.

Let $X$ and $Y$ denote the subsets of all $x$-positions and $y$-positions, respectively, where a piece can be positioned applying the principle of normal patterns. These sets can be computed as follows:

$$X = \left\{x = \textstyle\sum_{k \in P} w_k \xi_k \ : \ 0 \leq x \leq W, \ \xi_k \geq 0 \text{ integer}, \ k \in P\right\} \tag{4.1}$$

and

$$Y = \left\{y = \textstyle\sum_{k \in P} h_k \xi_k \ : \ 0 \leq y \leq H, \ \xi_k \geq 0 \text{ integer}, \ k \in P\right\} \tag{4.2}$$

The $x$-positions contained in $X$ are sorted so that given $x_i, x_j \in X$, we have $x_i < x_j$ for each $1 \leq i < j \leq |X|$. The $y$-positions contained in $Y$ are also similarly sorted.

A simple dynamic programming recursion for computing $X$ and $Y$ is described both by Christofides [52] and by Cintra [54].

### 4.2.2 A Dynamic Programming algorithm for the 2D-GCP

The 2D-GCP considered in this chapter can be solved using the following dynamic programming algorithm, originally proposed by Cinra et al. [54], and based on the recurrence formula proposed by Beasley [53].

Let $V(w,h)$ be the value of an optimal guillotine pattern of a rectangle of size $(w,h)$, evaluated by means of the following recurrence formula:

$$V(w,h) = \max \left\{ \begin{array}{l} v(w,h) \\ \max\{V(w',h) + V(p(w-w'),h) : w' \in X \text{ and } 0 < w' \leq \frac{w}{2}\} \\ \max\{V(w,h') + V(w,q(h-h')) : h' \in Y \text{ and } 0 < h' \leq \frac{h}{2}\} \end{array} \right\} \tag{4.3}$$

where $p(w) = \max\{x \in X : x \leq w\}$, $q(h) = \max\{y \in Y : y \leq h\}$ and where $v(w,h)$ denotes the value of the most valuable piece that can be cut in a rectangle of size $(w,h)$ ($v(w,h) = 0$ if no piece can be cut in such rectangle). Since the only $x$ and $y$-positions considered are the ones contained in the subsets $X$ and $Y$, for the sake of ease, we use the notation $V(x_i, y_j)$ and $V(i,j)$ interchangeably. The optimal solution value is $V(p(W), q(H)) = V(|X|, |Y|)$.

Let $cut(i,j)$ be the position of the optimal cut within a rectangle of size $(x_i, y_j)$, $x_i \in X$ and $y_j \in Y$. It is equal to 0 if guillotine cuts are not applied, it is $> 0$ if a horizontal cut is applied in position $cut(i,j)$, and it is $< 0$ if a vertical cut is applied in position $-cut(i,j)$.

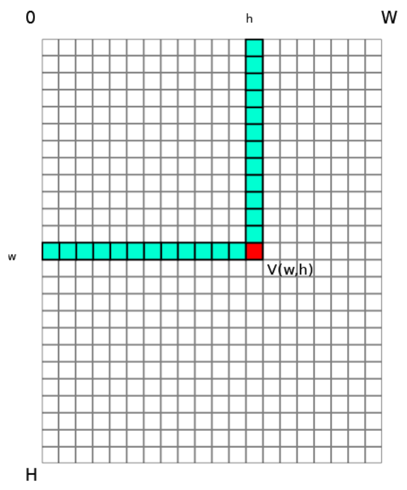
A pseudocode for the dynamic programming algorithm for the 2D-GCP proposed by Cintra et al. [54] is as follows:

**Algorithm** *DP-2D-GCP*$(W, H, \mathbf{w}, \mathbf{h}, \mathbf{v})$

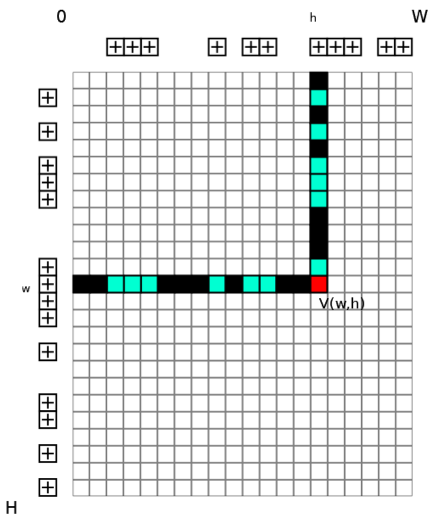
```
1.  //Compute the sets $X$ and $Y$ using the normal pattern principle
2.  //Initialization
3.  for $i = 1$ to $|X|$ do
4.      for $j = 1$ to $|Y|$ do
```

```
5.          $V(i,j) = max\,\{\{v_k : k \in P, w_k \leq x_i \text{ and } h_k \leq y_j\} \cup \{0\}\}$
6.          $cut(i,j) = 0$
7.   // Recurrence
8.   for $i = 2$ to $|X|$ do
9.      for $j = 2$ to $|Y|$ do
10.         for $i' = 1$ to $\max\{k : x_k \in X, x_k \leq x_i/2\}$ do
11.            $i'' = \max\{k : x_k \in X, x_k \leq x_i - x_{i'}\}$
12.            if $V(i,j) \leq V(i',j) + V(i'',j)$
13.              then $V(i,j) = V(i',j) + V(i'',j)$
14.                   $cut(i,j) = -i'$
15.         for $j' = 1$ to $\max\{k : y_k \in Y, y_k \leq y_j/2\}$ do
16.            $j'' = \max\{k : y_k \in Y, y_k \leq y_j - y_{j'}\}$
17.            if $V(i,j) \leq V(i,j') + V(i,j'')$
18.              then $V(i,j) = V(i,j') + V(i,j'')$
19.                   $cut(i,j) = j'$
```

The algorithm starts by computing the sets $X$ and $Y$ using the normal pattern principle. Then, it initializes $V(i,j)$, for every $x_i \in X$ and $y_j \in Y$, by setting $V(i,j) = v(x_i, y_j)$. The recurrence tries to improve each value $V(i,j)$ applying to the rectangle of size $(x_i, y_j)$ a horizontal or a vertical guillotine cut. If the sum of the values associated to the resulting two rectangles improves over $V(i,j)$, its value is updated and the applied cut is saved in *cut*$(i,j)$.



(A) Not using normal patterns



(B) Using normal patterns

FIGURE 4.1: Computation of function $V(x,y)$.

The usage of the subsets $X$ and $Y$ instead of the full sets of the available positions can heavily reduce the computational complexity. In Figure 4.1 we show the difference between the computation of function $V(x, y)$ not using and using the normal pattern principle. Normal patterns are shown by "+" icons adjacent to the corresponding rows and columns. The figure depicts how the computation of the $V(w, h)$ values is made based on all gray cells, each representing a partial solution. It is evident how the number of partial solutions needed for the computation is much lower using normal patterns than otherwise.

## 4.3 The GPU porting of the dynamic programming algorithm

The dynamic programming algorithm for the 2D-GCP proposed by Cintra et al. [54] and described in Section 4.2.2 is natively suitable for an implementation using GPU computing, due to its matrix-like structure.

In this section we show how to exploit the inherent parallelism of this algorithm, we describe the parallelization process, the porting to a GPU environment and its possible implementation using a CUDA model.

### 4.3.1 Exploiting the inherent parallelism

The objective of the algorithm is to compute the different $V(i, j)$ values. In order to do this, we need all the intermediate solutions evaluated in the previous iterations of the dynamic recursion (see Figures 4.1a and 4.1b), as described in Section 4.2.2. This process can be effectively decomposed into independent tasks, as shown in the following.

Each index $d$, corresponding to a stage of the dynamic recursion and such that $1 \leq d \leq |X|+|Y|-1$, identifies implicitely a corresponding anti-diagonal on matrix $V$. The anti-diagonal is composed by the set of cells $A_d = \{(k, d-k+1) : 1 \leq k \leq |X|, 1 \leq d-k+1 \leq |Y|, k = 1, \ldots, d\}$. At each stage $d = 1, \ldots, |X| + |Y| - 1$, we can concurrently compute the values $V(i, j)$ belonging to the corresponding anti-diagonal $A_d$ (i.e., $(i, j) \in A_d$) (see Figures 4.2a and 4.2b), and to evaluate each $V(i, j)$, we can perform a parallel *max operation* over the solution values of the

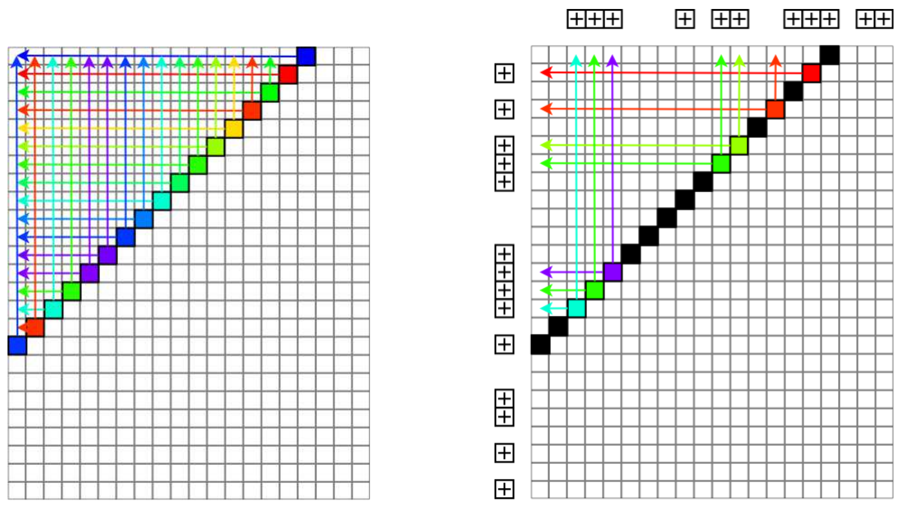

(A) Without discretization (B) With discretization

Figure 4.2: Parallel tasks executed exploiting the decomposition based on *anti-diagonals*.

composing subproblems. Notice that other approaches based on decomposing the problem by columns or rows do not allow such an effective concurrent evaluation of the $V(i, j)$ values.

### 4.3.2 Recurrence Parallelization on GPU

The parallel implementation of the recursion has been designed to fit the CUDA programming model. As described in 4.3.1, we can exploit two different granularities of parallelism, one for evaluating the $V(i, j)$ values belonging to the anti-diagonal $A_d$, and one for the max operation required for finding the cut that maximizes the $V(i, j)$ value. The mapping on CUDA becomes then straightforward:

1. inter-grid parallelism among blocks to concurrently evaluate all cells of $A_d$.
2. inter-block parallelism among threads to compute the reduction (max operation) for each $V(i, j)$.

At each recursion stage $d$, $1 \leq d \leq |X| + |Y| - 1$, each element $(i, j)$ of the anti-diagonal $A_d$ is assigned to a different GPU block in order to concurrently evaluate the corresponding $V(i, j)$ value. Therefore, the maximum number of blocks required to compute an anti-diagonal is $b = \min\{|X|, |Y|\}$.

A block computing cell $(i, j)$ runs threads to evaluate Beasley's recursive formula 4.3:

$$\begin{array}{ll} V_{ij}^X(i') = V(i', j) + V(i''(i, i'), j), & x_{i'} \in X \text{ and } 0 < x_{i'} \leq \frac{x_i}{2} \\ V_{ij}^Y(j') = V(i, j') + V(i, j''(j, j')), & y_{j'} \in Y \text{ and } 0 < y_{j'} \leq \frac{y_j}{2} \end{array} \tag{4.4}$$

where $i''(i, i') = \max\{k : x_k \in X, x_k \leq x_i - x_{i'}\}$ and $j''(j, j') = \max\{k : y_k \in Y, y_k \leq y_j - y_{j'}\}$.

Ideally, the block evaluating the values $V_{ij}^X(i')$ and $V_{ij}^Y(j')$ has a thread for each index $i'$ and $j'$. The obtained values can be stored in the same shared memory location. This step is crucial to maximize the global memory bandwidth because, for example, the thread evaluating $V_{ij}^X(i')$, instead of loading the two values $V(i', j)$ and $V(i''(i, i'), j)$ in the shared memory and performing the plus operation, wasting a large number memory cycles, performs the addition inside one instruction requiring two global memory accesses, doubling the memory bandwidth for each kernel call.

As mentioned before, the CUDA programming model allows to spawn a maximum of $\beta$ threads per block (the value of $\beta$ depends by the hardware configuration, e.g., $\beta = 1024$ for Fermi and Kepler and $\beta = 512$ for older architectures), therefore each thread can be forced to evaluate more than one value $V_{ij}^X(i')$ or $V_{ij}^Y(j')$ at each stage. In particular, a thread of index $t \leq \lfloor \beta/2 \rfloor$ evaluates every value $V_{ij}^X(i')$ where $i' \% \lfloor \beta/2 \rfloor = t$ (i.e., the remainder of division of $i'$ by $\lfloor \beta/2 \rfloor$). Whereas a thread of index $t > \lfloor \beta/2 \rfloor$ evaluates every value $V_{ij}^Y(j')$ where $\lfloor \beta/2 \rfloor + j' \% \lfloor \beta/2 \rfloor = t$. For sake of ease, we define $T_x = \{1, \ldots, nt_x = \lfloor \beta/2 \rfloor\}$, $T_y = \{\lfloor \beta/2 \rfloor + 1, \ldots, nt_y = \beta\}$, and the following set of position assigned to each thread:

$$\begin{array}{ll} P_{ij}^X(t) = \{i' : x_{i'} \in X, 0 < x_{i'} \leq \frac{x_i}{2}, i' \% \lfloor \beta/2 \rfloor = t\}, & t \in T_x \\ P_{ij}^Y(t) = \{j' : y_{j'} \in Y, 0 < y_{j'} \leq \frac{y_j}{2}, \lfloor \beta/2 \rfloor + j' \% \lfloor \beta/2 \rfloor = t\}, & t \in T_y \end{array} \tag{4.5}$$

While each thread evaluates the values $V_{ij}^X(i')$, $i' \in P_{ij}^X(t)$, or $V_{ij}^Y(j')$, $j' \in P_{ij}^X(t)$, it also performs a max operation among them. The resulting maximum value obtained is used in the parallel reduction for the max operation among threads, described in the next section, which uses the values saved in the shared memory.

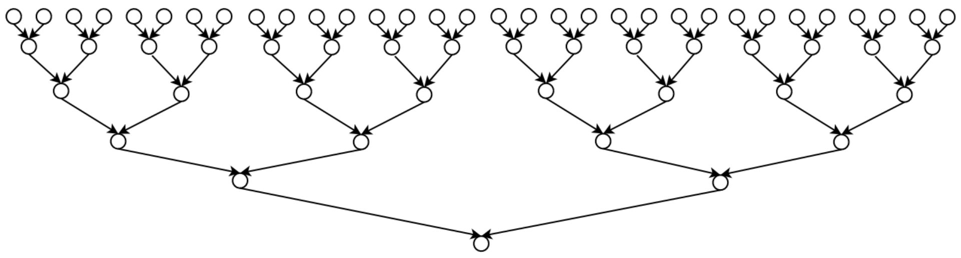


FIGURE 4.3: Reduction Tree.

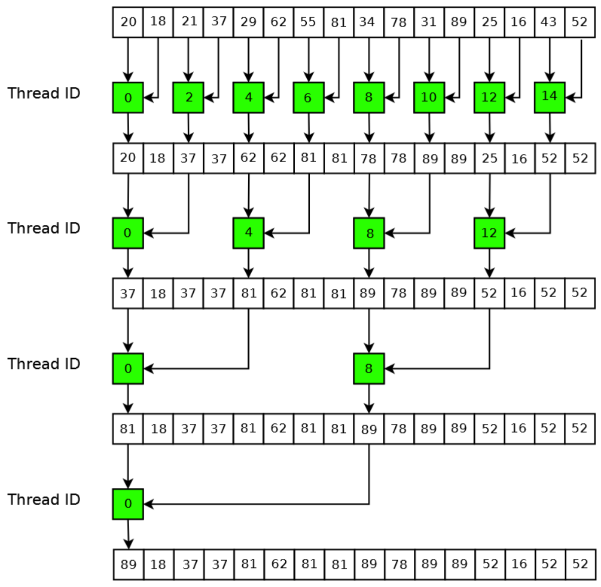


FIGURE 4.4: Naive reduction.

### 4.3.3 Parallel Reduction for the Max Operation

It is possible to exploit an inter-block parallelism among threads to compute the reduction required by the max operation for each $V(i,j)$. The reduction combines all the elements in a collection into a single one, using an associative two-input, one-output operator, which in our case is the max operator.

In general, a reduction is a low intensity arithmetic operation, but it has some critical aspects to analyze to get an efficient parallel algorithm. It is in fact crucial to be able to exploit all computational resources available on the the GPU device. The most effective solution we found to parallelize this max operation is a tree-based approach. We can represent the max operation as a binary tree, where we can do the operation in parallel at each level (Figure 4.3). The complexity of this algorithm becomes $O(N/P + logN)$ where $N$ is the number of elements in every level, and $P$ the number of processors. In our case $N = P$ and the complexity is $O(logN)$.

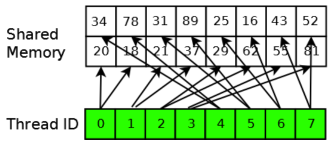


FIGURE 4.5: Naive reduction gives rise to bank conflicts (each column of the shared memory represents a memory bank).

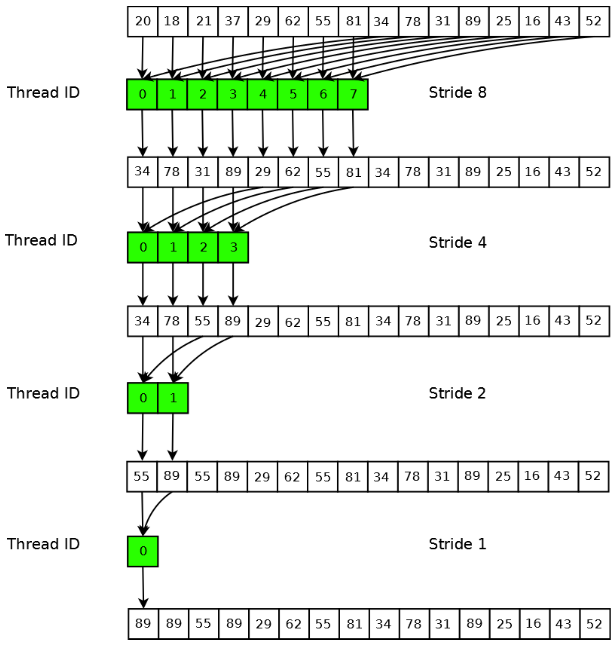


FIGURE 4.6: Strided access allows a fast reduction.

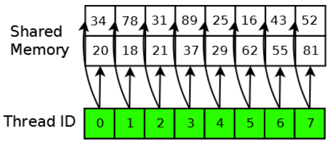


FIGURE 4.7: Strided access avoids bank conflicts.

The shared memory is subdivided into small arrays of 32 locations of 32-bit words each, and its latency is significantly lower ($\approx 4$ memory cycles) than that of the global memory. The shared memory is the only means to enable communications among threads of the same block and, given its bank-like structure, *bank conflicts* are the most important aspect to avoid for enhancing the kernel performance. A bank conflict occurs when more threads of the same warp access the same memory bank, thus forcing an access serialization. Avoiding this is an implementation constraint which imposes a specific management of threads communication.

A naive algorithm, as shown in Figures 4.4 and 4.5, implemented with interleaved addressing creates a large number of conflicts, serializing most of the operations in the shared memory. On the contrary, a strided access (see Figures 4.6 and 4.7) resolves this problem leading to a conflict-free access to the shared memory, to a maximization of the device memory bandwidth and finally to a good parallel execution of the threads [59]. In our kernel, once the loading stage described in section 4.3.2 is finished, we can perform this reduction step, retrieving the maximization value for $V(i, j)$.

### 4.3.4 Matrix Update

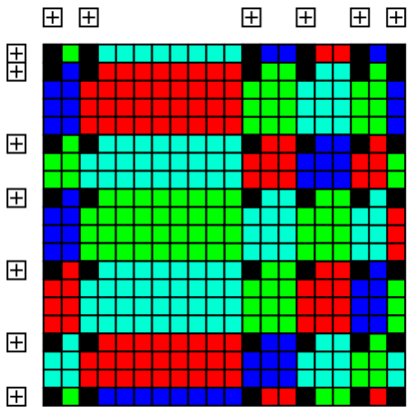
The normal pattern principle allows to reduce the set of values $V(i, j)$ to be evaluated at each stage, this results in an improvement of the performance of the corresponding *serial* algorithm, presented in section 4.2.2. Unfortunately, the same approach is not suitable for the GPU, because the resulting algorithm would induce sparse, not linear, and serialized access to the global memory.

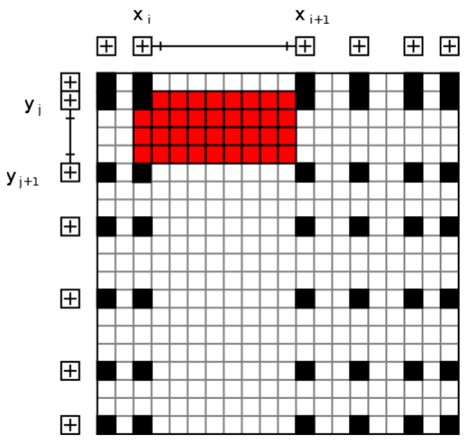
New NVIDIA architectures, such as *Fermi* or *Kepler*, partially resolve the problem of *coalesced access* to global memory, that is the quest for loading data which resides in adjacent positions even in the global memory. In any case, a more efficient memory access can be obtained by transforming the double $i$, $j$ indexing of the $V$ values into a one-index access, which maximizes the bandwidth on the PCI-Express bus, structuring the data in a way more suitable for a coalesced access.

A plain and sequential access to the partial solution values required for evaluating each $V(i, j)$ can be obtained by filling the discretization induced by the normal patterns (see Figure 4.1a), as described below. Moreover, in order to maximize coalescence, we stored the $V$ matrix twice, in two mono-dimensional arrays: one in row-major order $V_{rows}$ and one in a column-major order $V_{cols}$. When the algorithm evaluates the $V^X_{ij}(i')$ it uses matrix $V_{cols}$, whereas when it evaluates the values $V^Y_{ij}(j')$ it uses matrix $V_{rows}$ (see section 4.3.2).

Using these structures, we can take advantage of discretization when we evaluate the $V(i, j)$ only for the normal pattern positions, and we can use a complete matrix representation by means of the arrays $V_{rows}$ and $V_{cols}$ defined for every integer positions $0 \leq x \leq W$ and $0 \leq y \leq H$, when we compute the maxima.

Given two adjacent normal pattern positions $x_i, x_{i+1} \in X$ and $y_j, y_{j+1} \in Y$, we have that $V(x_i, y_j) = V(x, y)$ for every $x_i \leq x < x_{i+1}$ and $y_j \leq y < y_{j+1}$. Therefore, when the value $V(x_i, y_j)$ is evaluated, it can be directly copied in every


(A) Matrix structure



(B) Update for the value V(i,j)

FIGURE 4.8: Matrix update in the GPU computing approach proposed

integer positions $x_i \leq x < x_{i+1}$ and $y_j \leq y < y_{j+1}$ (see Figures 4.8a and 4.8b), this permits to fill the gaps in the matrix storage and obtain a full linear access when computing the corresponding maximum.

As mentioned, the value $V(x_i, y_j)$ is then copied in both linearized matrices $V_{rows}$ and $V_{cols}$.

### 4.3.5 GPU Algorithm

In this section we present a pseudo-code where the structure of GPU algorithm is fully detailed.

The core of the algorithm is the dynamic stage recursion, which is implemented in a main loop, working at each iteration on an anti-diagonal $d$. Every element $V(i,j)$, $(i,j) \in A_d$, is assigned to a different block, which concurrently evaluates it by expression 4.3. Each block splits the computation among threads. Half of them consider the values $V^X_{ij}(i')$ and the remaining ones consider the values $V^Y_{ij}(j')$ (see section 4.3.2). At the end, each block makes the reduction corresponding to the max operation of expression 4.3 and updates the corresponding entries of both linearized matrices $V_{rows}$ and $V_{cols}$. The cut positions are saved in the linearized matrix $V_{cut}$.

**Algorithm** *GPU DP-2D-GCP*$(H, W, \mathbf{w}, \mathbf{h}, \mathbf{v})$

```
1.  //Compute the sets $X$ and $Y$ using the normal pattern principle
2.  //Initialization
3.  **for** $i = 1$ **to** $|X|$ **do**
4.      **for** $j = 1 \ldots, |Y|$ **do**
5.          $V(i,j) = \max\{\{v_k : k \in P, w_k \leq x_i \text{ and } h_k \leq y_j\} \cup \{0\}\}$
6.          // Initialize the linearized matrices $V_{cols}$ and $V_{rows}$
7.          **for** $x = x_i$ **to** $x_{i+1} - 1$ **do**
8.              **for** $y = y_j$ **to** $y_{j+1} - 1$ **do**
9.                  $V_{rows}[yW + x] = V(i,j)$
10.                 $V_{cols}[xH + y] = V(i,j)$
11.                 $V_{cut}[xH + y] = 0$
12. // Recurrence
13. **for** $d = 1$ **to** $|X| + |Y| - 1$ **do**
14.     **for** each $(i,j) \in A_d$ **do**
15.         // CUDA Kernel: each $(i,j)$ is assigned to a different block
16.         // Threads evaluate $V^X_{\max} = \max\{V^X_{ij}(i') : x_{i'} \in X, 0 < x_{i'} \leq \frac{x_i}{2}\}$
17.         **for** each thread $t \in T_x$ **do**
18.             $V'[t] = 0$, $C'[t] = nil$ // Shared Memory Initialization
19.             **for** each $i' \in P^X_{ij}(t)$ **do**
20.                 **if** $V'[t] < V_{rows}[y_jW + x_{i'}] + V_{rows}[y_jW + (x_i - x_{i'})]$
21.                    **then** $V'[t] = V_{rows}[y_jW + x_{i'}] + V_{rows}[y_jW + (x_i - x_{i'})]$
22.                         $C'[t] = -x_{i'}$
23.         // Reduction: at the end $V'[1] = V^X_{\max}$
24.         **for** $s = \lfloor nt_x/2 \rfloor$ **to** 1, $s = \lfloor s/2 \rfloor$ **do**
25.             **if** $(t \leq s)$ and $(V'[t] < V'[t+s])$
26.               **then** $V'[t] = V'[t+s]$, $C'[t] = C'[t+s]$
27.         // Threads evaluate $V^Y_{\max} = \max\{V^Y_{ij}(j') : y_{j'} \in Y, 0 < y_{j'} \leq \frac{y_j}{2}\}$
28.         **for** each thread $t \in T_y$ **do**
29.             $V'[t] = 0$, $C'[t] = nil$ // Shared Memory Initialization
30.             **for** each $j' \in P^Y_{ij}(t)$ **do**
31.                 **if** $V'[t] < V_{cols}[x_iH + y_{j'}] + V_{cols}[x_iH + (y_j - y_{j'})]$
32.                    **then** $V'[t] = V_{cols}[x_iH + y_{j'}] + V_{cols}[x_iH + (y_j - y_{j'})]$
33.                         $C'[t] = y_{j'}$
34.             // Reduction: at the end $V'[nt_x + 1] = V^Y_{\max}$
35.             **for** $s = \lfloor nt_y/2 \rfloor$ **to** 1, $s = \lfloor s/2 \rfloor$ **do**
```

```
36.                 **if** $(t \leq nt_x + s)$ and $(V'[t] < V'[t+s])$
37.                     **then** $V'[t] = V'[t+s]$, $C'[t] = C'[t+s]$
38.         **if** $V'[1] < V'[nt_x + 1]$
39.             **then** $MaxV = V'[1]$, $MaxC = C'[1]$
40.             **else** $MaxV = V'[nt_x + 1]$, $MaxC = C_x[nt_x + 1]$
41.         // Update $V_{cols}$ and $V_{rows}$
42.         **for** $x = x_i$ **to** $x_{i+1} - 1$**do**
43.             **for** $y = y_j$ **to** $y_{j+1} - 1$ **do**
44.                 $V_{rows}[yW + x] = MaxV$
45.                 $V_{cols}[xH + y] = MaxV$
46.                 $V_{cut}[xH + y] = MaxC$
```

The initialization section at the beginning of the algorithm includes the setup of the $V_{rows}$, $V_{cols}$, and $V_{cut}$ data structures. This is presented in the same loops where we access the matrices linearized by rows to simplify the presentation, but in the actual implementation we initialize $V_{rows}$ by rows and $V_{cols}$ and $V_{cut}$ by columns.

In any given iteration $d$ of the recurrence, the concurrent execution of the blocks is managed by the GPU scheduler and at the end of the iteration a synchronization is performed (i.e., iteration $d + 1$ is performed only after every blocks ended at iteration $d$).

Notice that the reduction performed by each thread also requires a synchronization among threads.

## 4.4 Computational results

This section reports the computational results and, in particular, the speed-up factors obtained running the serial and the parallel versions of the algorithm. The code was implemented in C/C++ with CUDA extensions using the NVCC Compiler by Nvidia for the GPU version and using Microsoft Visual Studio 2010 with full optimization for the serial version.

All tests were run on a Intel i7 920 Bloomfield QuadCore @2.8 GHz with 6 Gigabytes of RAM and two different graphic card: an Nvidia GeForce GTX 570 Fermi with 1 Gigabyte of GDDR5 RAM and 480 CUDA Cores @1.464 GHz and an Nvidia Geforce GTX 770 Kepler with 2 Gigabytes of GDDR5 RAM and 1536 CUDA Cores @1.046 GHz.

We tested the algorithm on two different hardware configurations to compare the scalability of the method proposed on two different generations of GPUs.

### 4.4.1 Test instances

We tested our algorithm on three different instance sets. The first set is the well-known *gcut* set, generated by Beasley [53] and upgraded by Cintra et al. [54]. In order to check the effectiveness and the correctness of the parallel algorithm, we generated by means of two opposite methodologies the other two sets, named *testcut* and *randcut*. These two sets were generated as follows.

Every instance of the *testcut* set is composed by three types of items, where the $w$ and $h$ dimensions are as follows:

- 25% of items with $h \in [1, H/4[$ and $w \in [1, W/4[$.
- 50% of items with $h \in [H/4, H/2[$ and $w \in [W/4, W/2[$.
- 25% of items with $h \in [H/2, 3H/4[$ and $w \in [W/2, 3W/4[$.

$H$ and $W$ are the dimensions of the master bin.

The instances of the *randcut* set are composed by items retrieved from randomly generated guillotine cuts on the original master bin. The peculiarity of this set is that the objective function $z$ is equal to the bin area, $W \times H$, giving us the possibility to check the correctness of our algorithm on big instances with known optimal solutions value. These two new sets are available on the website: www.sitoistanze.com, together with the images of all the instances.

### 4.4.2 Experiments

The objective of our experiments is the comparison of our implementations of the sequential and of the GPU versions of the algorithm, described in sections 4.2.2 and 4.3.5, respectively. The GPU version was run with 256 threads per block.

In Tables 4.1, 4.2, and 4.3 we report the computational results obtained on the three considered sets of instances. For each instance we indicate its $Name$, the size of the master surface $H$ and $W$, the number of type of items $n$, the number of normal pattern positions $|Y|$ and $|X|$, the optimal solution value $z_{OPT}$, and the percentage waste $Waste\%$. For the algorithms we report the computing times $T_{CPU}$ of the sequential version and $T_{GPU}$ of the GPU version, and the resulting *SpeedUp* defined by the ratio between $T_{GPU}$ and $T_{CPU}$, i.e., $SpeedUp = T_{CPU}/T_{GPU}$.

In Table 4.1 we ignored the first eleven instances due to non significant computational time required; in fact, the times required to solve these instances are below 0.0001 seconds. Analyzing the results, we can highlight the method's scalability as a function of the dimension of the instances. The GTX 570's decreasing performances over the 8000x8000 dimensions of the master bin is the effect of the device's limited amount of on-board memory. Over that dimensions, we are forced to keep in the system's main memory some data structures, increasing the time required to access these structure and, significantly, slowing down the performances. The tests on the GTX 770 graphic card avoid this problem, due to the greater amount of on-board memory available. As we can see, in this case, the speed-up factor is almost constant.

Figures 4.9 and 4.10 display the original random generated instance *randcutc6000* and its solution, respectively. As we can see, the calculated solution is almost the same, except the cuts' positions on the bin. Figure 4.11 displays *testcut8000*'s solution, while Figure 4.12 shows the solution of *gcut13*.

TABLE 4.1: Computational results on *gcut* instances.

| Instances | | | | | | | | Core i7 920 | NVIDIA GTX 570 | | NVIDIA GTX 770 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | $H$ | $W$ | $n$ | $\lvert Y\rvert$ | $\lvert X\rvert$ | $z_{OPT}$ | *Waste%* | $T_{CPU}$ | $T_{GPU}$ | *SpeedUp* | $T_{GPU}$ | *SpeedUp* |
| gcut12 | 1000 | 1000 | 50 | 155 | 124 | 979,986 | 2.001 | 0.016 | 0.011 | 1.455 X | 0.011 | 1.455 X |
| gcut13 | 3000 | 3000 | 32 | 1457 | 2310 | 8,997,780 | 0.025 | 12.199 | 0.681 | 17.913 X | 0.574 | 6.959 X |
| gcut14 | 3500 | 3500 | 42 | 2390 | 2861 | 12,245,410 | 0.037 | 26.754 | 1.390 | 19.247 X | 1.168 | 21.251 X |
| gcut15 | 3500 | 3500 | 52 | 2422 | 2933 | 12,246,032 | 0.032 | 27.627 | 1.440 | 19.185 X | 1.206 | 22.907 X |
| gcut16 | 3500 | 3500 | 62 | 2559 | 2943 | 12,248,836 | 0.010 | 28.985 | 1.514 | 19.145 X | 1.267 | 22,878 X |
| gcut17 | 3500 | 3500 | 82 | 2676 | 2953 | 12,248,892 | 0.009 | 30.015 | 1.580 | 18.997 X | 1.331 | 22.555 X |

TABLE 4.2: Computational results on *testcut* instances.

| Instances | | | | | | | | Core i7 920 | NVIDIA GTX 570 | | NVIDIA GTX 770 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | $H$ | $W$ | $n$ | $\|Y\|$ | $\|X\|$ | $z_{OPT}$ | *Waste%* | $T_{CPU}$ | $T_{GPU}$ | *SpeedUp* | $T_{GPU}$ | *SpeedUp* |
| testcut6000 | 6000 | 6000 | 80 | 4881 | 5384 | 35985098 | 0.041 | 163.145 | 6.797 | 24.003 X | 6.314 | 25.839 X |
| testcut6500 | 6500 | 6500 | 80 | 5610 | 6350 | 42241403 | 0.020 | 230.397 | 9.442 | 24.401 X | 8.724 | 26.410 X |
| testcut7000 | 7000 | 7000 | 80 | 6242 | 6757 | 48997730 | 0.004 | 295.356 | 11.669 | 25.311 X | 10.900 | 27.097 X |
| testcut7500 | 7500 | 7500 | 80 | 4988 | 6441 | 56201826 | 0.085 | 253.937 | 10.066 | 25.227 X | 9.795 | 25.925 X |
| testcut8000 | 8000 | 8000 | 100 | 7365 | 7426 | 63993589 | 0.010 | 437.347 | 17.004 | 25.720 X | 16.733 | 26.137 X |
| testcut8500 | 8500 | 8500 | 80 | 8413 | 8040 | 72249152 | 0.001 | 518.404 | 35.351 | 14.664 X | 16.826 | 30.744 X |
| testcut9000 | 9000 | 9000 | 80 | 7495 | 8546 | 80980280 | 0.024 | 569.479 | 37.899 | 15.026 X | 20.868 | 27.290 X |
| testcut9500 | 9500 | 9500 | 80 | 8784 | 8124 | 90231106 | 0.020 | 673.266 | 45.342 | 14.849 X | 24.807 | 27.140 X |
| testcut10000 | 10000 | 10000 | 80 | 9365 | 9426 | 99998425 | 0.001 | 866.644 | 57.579 | 15.051 X | 32.283 | 26.845 X |

Table 4.3: Computational results on *randcut* instances.

| Instances | | | | | | | | Core i7 920 | NVIDIA GTX 570 | | NVIDIA GTX 770 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | $H$ | $W$ | $n$ | $\|Y\|$ | $\|X\|$ | $z_{OPT}$ | *Waste%* | $T_{CPU}$ | $T_{GPU}$ | *SpeedUp* | $T_{GPU}$ | *SpeedUp* |
| randcut6000a | 6000 | 6000 | 34 | 4971 | 5747 | 36000000 | 0.000 | 180.898 | 7.275 | 24.866 X | 6.770 | 26.722 X |
| randcut6000b | 6000 | 6000 | 32 | 5456 | 5566 | 36000000 | 0.000 | 187.933 | 7.700 | 24.407 X | 7.103 | 26.458 X |
| randcut6000c | 6000 | 6000 | 36 | 5630 | 4844 | 36000000 | 0.000 | 169.182 | 7.039 | 24.035 X | 6.452 | 26.223 X |
| randcut6500a | 6500 | 6500 | 28 | 4967 | 4645 | 42250000 | 0.000 | 161.617 | 6.572 | 24.592 X | 6.081 | 26.579 X |
| randcut6500b | 6500 | 6500 | 52 | 6255 | 6311 | 42250000 | 0.000 | 255.481 | 10.249 | 24.927 X | 9.385 | 27.223 X |
| randcut6500c | 6500 | 6500 | 40 | 6187 | 5783 | 42250000 | 0.000 | 234.749 | 9.436 | 24.878 X | 8.640 | 27.169 X |
| randcut7000a | 7000 | 7000 | 44 | 6631 | 6669 | 49000000 | 0.000 | 306.135 | 12.083 | 25.336 X | 11.190 | 27.358 X |
| randcut7000b | 7000 | 7000 | 38 | 5956 | 6242 | 49000000 | 0.000 | 261.815 | 10.514 | 24.902 X | 9.814 | 26.679 X |
| randcut7000c | 7000 | 7000 | 32 | 6793 | 6494 | 49000000 | 0.000 | 278.242 | 12.096 | 23.003 X | 11.225 | 24.788 X |
| randcut7500a | 7500 | 7500 | 44 | 6993 | 6981 | 56250000 | 0.000 | 364.229 | 14.384 | 25.322 X | 13.425 | 27.132 X |
| randcut7500b | 7500 | 7500 | 32 | 6564 | 6565 | 56250000 | 0.000 | 334.090 | 12.850 | 25.999 X | 12.258 | 27.256 X |
| randcut7500c | 7500 | 7500 | 32 | 6651 | 6507 | 56250000 | 0.000 | 331.501 | 12.843 | 25.812 X | 12.247 | 27.069 X |
| randcut8000a | 8000 | 8000 | 28 | 7876 | 5639 | 64000000 | 0.000 | 363.668 | 14.174 | 25.657 X | 14.183 | 25.642 X |
| randcut8000b | 8000 | 8000 | 36 | 7967 | 6499 | 64000000 | 0.000 | 378.706 | 16.019 | 23.641 X | 15.863 | 23.874 X |
| randcut8000c | 8000 | 8000 | 34 | 7465 | 6935 | 64000000 | 0.000 | 417.253 | 16.281 | 25.628 X | 16.027 | 26.035 X |
| randcut8500a | 8500 | 8500 | 46 | 8023 | 8100 | 72250000 | 0.000 | 527.499 | 23.914 | 22.058 X | 19.392 | 27.202 X |
| randcut8500b | 8500 | 8500 | 42 | 8098 | 7748 | 72250000 | 0.000 | 523.007 | 22.965 | 22.774 X | 18.768 | 27.867 X |
| randcut8500c | 8500 | 8500 | 36 | 8208 | 6967 | 72250000 | 0.000 | 483.054 | 21.413 | 22.559 X | 17.411 | 27.744 X |
| randcut9000a | 9000 | 9000 | 44 | 8728 | 8547 | 81000000 | 0.000 | 613.783 | 41.284 | 14.867 X | 23.287 | 26.357 X |
| randcut9000b | 9000 | 9000 | 42 | 8712 | 9000 | 81000000 | 0.000 | 657.307 | 43.418 | 15.139 X | 24.206 | 27.155 X |
| randcut9000c | 9000 | 9000 | 36 | 8558 | 7075 | 81000000 | 0.000 | 540.588 | 32.943 | 16.410 X | 19.733 | 27.395 X |
| randcut9500a | 9500 | 9500 | 40 | 9256 | 6813 | 90250000 | 0.000 | 623.986 | 37.462 | 16.657 X | 22.143 | 28.180 X |
| randcut9500b | 9500 | 9500 | 36 | 8661 | 9349 | 90250000 | 0.000 | 743.840 | 49.040 | 15.168 X | 27.395 | 27.152 X |
| randcut9500c | 9500 | 9500 | 36 | 7931 | 8533 | 90250000 | 0.000 | 656.044 | 42.975 | 15.266 X | 24.012 | 27.322 X |
| randcut10000a | 10000 | 10000 | 36 | 8587 | 8382 | 100000000 | 0.000 | 723.389 | 48.010 | 15.067 X | 27.788 | 26.032 X |
| randcut10000b | 10000 | 10000 | 42 | 9298 | 9586 | 100000000 | 0.000 | 882.447 | 60.378 | 14.615 X | 32.539 | 27.120 X |
| randcut10000c | 10000 | 10000 | 38 | 8601 | 9618 | 100000000 | 0.000 | 810.420 | 53.605 | 15.118 X | 31.043 | 24.906 X |

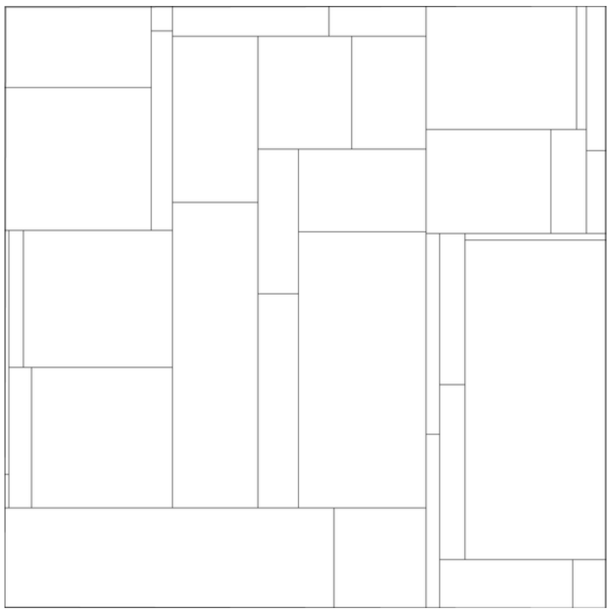


FIGURE 4.9: randcut6000c source.

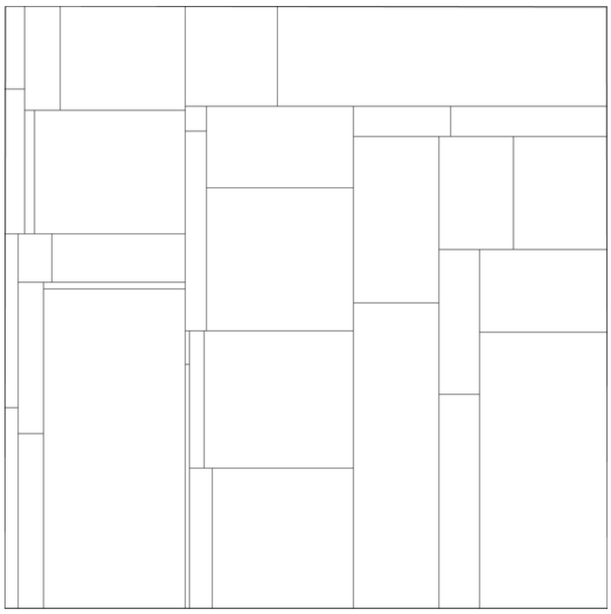


FIGURE 4.10: randcut6000c solution.

## 4.5 Considerations and Future Work

In this chapter we presented a parallel algorithm for solving the Unconstrained Two-Dimensional Guillotine Cutting Problem (2D-GCP) especially designed for running on a GPGPU platform. We proved the effectiveness of this method achieving, in the best case, a 30X speed-up factor upon the serial version, exploiting the native matrix-like structure of the problem and the fine grained computation required by the dynamic programming algorithm. We also provided two new sets of

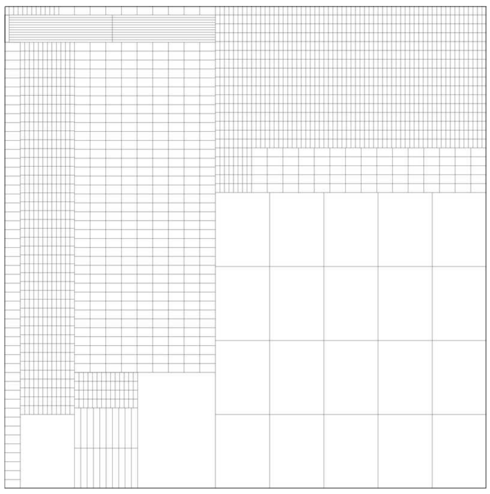


Figure 4.11: testcut8000 solution.

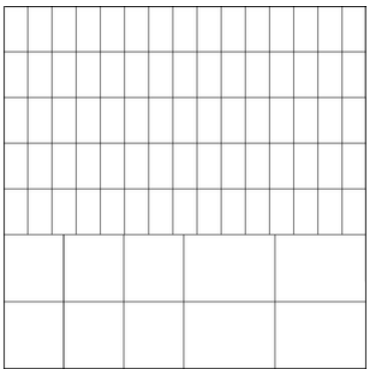


Figure 4.12: gcut13 solution.

test instances for this problem.

Our future aims are to extend the algorithm for solving the *staged* version of the problem and, eventually, GPU Computing to enhance other algorithms designed to approach similar packing problems (i.e. Bin Packing, Constrained Two-Dimensional Guillotine Cutting Problem, etc.).

# Chapter 5

# Vehicle Routing Problem

In this chapter we investigate the application of GPU computing to some of the most effective pricing strategies based on Dynamic Programming (q-route, through-q-route and ng-route relaxations) for Column Generation methods for the Vehicle Routing Problem. We propose the parallel versions of these algorithms in a massively parallel environment, discussing the implementation choices and evaluating the speed-up factors on literature test instances with respect to the serial version.

## 5.1 Introduction

The Vehicle Routing Problem (VRP) is among the most studied problems in combinatorial optimization, and retains unabated interest both because, though simple to state, it enjoys intriguing mathematical properties and because it can be quickly specified into problems of primary economic interest. The literature on the core problem variants and on the possible real-world variations got huge after the seminal paper which introduced it [60], and includes dedicated books [61], [62] and, more recently, also dedicated working groups of research associations [63].

The core problem can be quickly introduced as finding a least cost set of routes to service a number of customers from a central depot, given a cost matrix specifying the cost for going from any customer to any other one and from the depot to each customer. The problem can then be complicated at will, by adding constraints suggested by real world applications. A largely included constraint assumes that the routes are to be traveled by trucks in order to deliver or to collect goods from customers, thus the total amount of goods loaded on each truck cannot exceed its capacity, in weight or in volume. This gives rise to the Capacitated VRP variant (CVRP). Alternatively, each customer can ask either for a delivery or for a collection of goods, yielding the Pickup and Delivery variant(PDVRP). In case all deliveries of each route are to be made first, then all collections, we have the CVRP with backhauls (VRPB). A further quite common constraint considers feasible time windows for the visits at the customers (TWVRP). Moreover, in small area settings each truck could go back to the depot to reload (multitrip VRP, MTVRP), while in bigger areas it is common the use of more depots by the vehicles of the fleet (multidepot VRP, MDVRP). The vehicles of the fleet can be all equal or different among themselves (heterogeneous fleet CVRP, HVRP), could not be requested to return to the depot they started from (open CVRP, OVRP), could be requested to repeat the same routes with a given periodicity over the planning horizon (periodic VRP, PVRP), etc.

Furthermore, all listed constraints, and many more coming from operational practice, can be freely combined to model actual use cases. For example, a recent work on city logistics operational optimization ([64]) models its problem as a CVRP with time windows, multi-trip, heterogeneous fleet and pickup and delivery.

Given its theoretical and practical relevance, the VRP witnessed a wealth of diverse approaches for its solution and still fosters a lively research community studying either exact or heuristic methods or, more recently, both. A detailed survey is clearly out of scope for this paper, in the following we will recall just a few contributions.

Heuristic approaches have a seminal work in [65] and went through tailored heuristics, such as [66], then metaheuristics, such as tabu search ([67]), simulated annealing ([68]), ant colony [69], genetic and in general evolutionary algorithms [70], variable neighborhood search [71] and PSO [72], just to name a few.

Exact approaches are of more direct interest for this paper. Again, different approaches have been used, ranging from dynamic programming [73] to branch and bound [74], from branch and cut [75] to column generation [76]. In all cases, a central feature is the ability to compute tight lower bounds. Again, different approaches for computing bounds, recently, bounds based on nonelementary paths, such as q-routes [74] and most notably ng-routes [77, 78]appear to be particularly effective for the Capacitated VRP and VRP with Time Windows.

This chapter reports on the results obtained by implementing the q-routes, through-q-routes and ng-routes relaxations computation on a GPU parallel architecture. GPU are enjoying increasing interest among the optimization community given the possibility to significantly speedup tasks at the core of any approach of interest, thus to ultimately achieve substantial efficiency improvements [79]. Applications to combinatorial optimization problems have so far been reported for the knapsack problem [58] and for the Two-Dimensional Guillotine Cutting Problem [80]. This is the first work porting state of-the-art vehicle routing optimization components on GPU, specifically proposing a GPU implementation of the ng-relaxation.

The implementation on GPU of an optimization algorithm is a complex task that involves the study of tailored data structures and corresponding routines. This chapter reports in detail the choices we made to achieve the most efficient parallel implementation of the q-routes, through-q-routes and ng-routes routines and substantiates this with computational results on standard problem benchmarks from the literature.

## 5.2 Problem Description and Mathematical Formulations

The CVRP, in its basic version, consists in finding the least-cost set of routes to be travelled by $m$ homogeneous vehicles of identical capacity $Q$, in order to service each of $n$ customers, whose index set is $V_1$. All routes start and return to a common depot, conventionally indexed by 0. Let $V = V_1 \cup \{0\}$. Input data consist of the requests $q_i$, $i = 1, \ldots, n$ and of the travelling costs $c_{ij}$, $i = 0, \ldots, n$, $j = 0, \ldots, n$, between each pair of customers and between each customer and the depot.

The problem can thus be defined on a complete weighted graph $G = V, A, C$, where $A = [(i, j)], i, j \in V$, and $C = [c_{ij}], i, j \in V$ is the corresponding possibly asymmetric cost matrix. In real-world application, $G$ is typically an overlay graph superimposed on an actual road network, and nodes in $V$ correspond to geocoded facilities while arcs in $A$ correspond to least-cost paths, computed according to the metric to minimize (distance, time, ... ).

The problem can be formulated in different ways, we refer the reader to [61] for a thorough overview. The following two subsections introduce the formulations and the notation we use in the rest of the paper.

### 5.2.1 Two Index Formulation

The two index formulation associates a decision variable $x_{ij} \in \{0,1\}$ to each arc $(i,j) \in A$, specifying whether or not in the optimal solution there is a vehicle travelling directly from node $i \in V$ to node $j \in V$.

Different variants of this formulation are possible, the most compact ones require to compute for each subset of nodes $S \subseteq V$ the minimum number of vehicles needed to service set S, which is denoted by $r(S)$.

The formulation is as follows.

$$
\begin{aligned}
z_{CVRP} = \min \quad & \sum_{i\in V}\sum_{j\in V} c_{ij}x_{ij} && && (5.1)\\
s.t. \quad & \sum_{i\in V} x_{ij} = 1 && j \in V_1 && (5.2)\\
& \sum_{j\in V} x_{ij} = 1 && i \in V_1 && (5.3)\\
& \sum_{j\in V} x_{0j} = m && && (5.4)\\
& \sum_{i\notin S}\sum_{j\in S} x_{ij} \geq r(S) && S \subseteq V_1, S \neq \emptyset && (5.5)\\
& x_{ij} \in \{0,1\} && i,j \in V && (5.6)
\end{aligned}
$$

Constraints 5.2 and 5.3 impose that exactly one vehicle arrives and leaves each customer, constraint 5.4 specify the number of available vehicles (not all of which need to be used) and constraints 5.5, the so called *capacity−cut* constraints impose both the connectivity of the solution and the vehicle capacity requirements (see [61]).

### 5.2.2 Set Partitioning Formulation

The set partitioning formulation, originally proposed by [81], associates a decision variable $x_\ell$ to each feasible vehicle route, that is, to each route that can be travelled by a vehicle, leaving the depot, servicing a subset of customers that collectively do not exceed the vehicle capacity and finally returning to the depot.

Let $\mathscr{R}$ be the index set of all feasible routes, let $c_\ell$ be the cost of route $\ell 1 in \mathscr{R}$, and let $a_{i,\ell}$ be a binary coefficient, which is equal to 1 iff node $i \in V$ belongs to route $\ell \in (R)$.

The formulation is as follows.

$$z_{SP} = \min \sum_{\ell \in \mathscr{R}} c_\ell x_\ell \tag{5.7}$$

$$s.t. \sum_{\ell \in \mathscr{R}} a_{i\ell} x_\ell = 1 \qquad i \in V_1 \tag{5.8}$$

$$\sum_{\ell \in \mathscr{R}} x_\ell = m \tag{5.9}$$

$$x_\ell \in \{0,1\} \qquad \ell \in \mathscr{R} \tag{5.10}$$

Constraints 5.8 ensure that each customer is serviced by exactly one feasible route and constraint 5.9 impose the fleet cardinality. It is noteworthy that, in case the cost matrix satisfied the triangle inequality, equalities 5.8 could be turned into *greater or equal than* inequalities, thus turning the problem into an extended set covering problem, which is computationally easier to deal with.

## 5.3 Dynamic Programming Relaxations for the Pricing Problem

In 5.1 we have briefly described some techniques to find a solution, heuristic or exact, to the VRP. Mainly regarding the exact algorithms, is necessary to solve the pricing problem for selecting the most interesting columns for the Column Generation (CG) algorithm. The pricing problem is also an NP-Hard problem, the Elementary Shortest Path Problem with Resource Constrain (ESPPRC). This problem is formally defined: let $\mathbf{P}$ be the set of paths of $G$ s.t. each path $P \in \mathbf{P}$ starts from 0, visits a set of vertices $V_P \subseteq V$, delivers $q_P$ units of product and ends at vertex $\sigma_P \in V_P$ without loops. The ESPPRC can be solved with Dynamic Programming recursions expressed as follows: we define a state-space graph $\mathbf{X} = \{(X, i) : X \subseteq V, i \in V'\}$ and functions $f(X, i), \forall (X, i) \in \mathbf{X}$, where $f(X, i)$ is the cost of the least-cost path $P$ that visits the set of customers $X$, ends at the customer $i \in X$, and such that $\sum_{j \in X} q_j \leq Q$. As we can see, the exact DP algorithm can't be applied because of the dimension of the state-space graph $\mathbf{X}$. Christofides et al. [82] proposed the State-Space Relaxation that is a procedure whereby the state-space associated with DP recurrence is relaxed to compute valid lower bounds to the original problem. The next three relaxations that we will introduce in the next sections of this chapter are relaxations for the ESPPRC problem based on this principle.

In [74], [77], [83], [84], [85] , were proposed effective and reliable relaxations, based on Dynamic Programming recurrences. These recurrences, as well as all dynamic programming algorithms, trade space for time, enumerating all the interesting solutions for the relaxed problem. In these cases the elementary constraint is relaxed and the aim is to find interesting almost elementary paths that can be the base for the creation of feasible solution or a good start point for the computation of valid and tight lower bounds. In the next sections we will describe three of these methods (q-route, through-q-route and ng-route) and we will analyze the intrinsic characteristics of each one.

### 5.3.1 q-Route Relaxation

The q-route relaxation described in Christofides and al. [74] , is aimed to find routes without loops of two vertices. Defining $f(q, i)$ the cost of the least cost path $P = (0, 1_1, \ldots, i_k), i_k = i$, (not necessarily simple) from the depot 0 to the customer $i$ with total load $q = \sum_{h=1}^{k} q_{ij}$. Such a path is called q-path. A q-path with the additional edge $0, i$ is called q-route and has cost $f(q, i) + d_{0i}$. We can impose that the path should not contains loops formed by three consecutive vertices can be described as follows: let $\pi(q, i)$ be the vertex just prior to $i$ on the path corresponding to $f(q, i)$. Let $\phi(q, i)$ be the cost of the least cost path ending at vertex $i$ with the constraint that the vertex $\gamma(q, i)$ preceding $i$ is not equal to

$\pi(q,i)$. This recurrence can be formalized as follows: for a given value of $q$, let $h(j,i)$ be the cost of the least path from 0 to $i$ with $j$ just prior to $i$ and without loops. Then:

$$h(j,i) = \begin{cases} f(q-q_i,j)+d_{ji}, & \text{if } \pi(q-q_i,j) \neq i \\ \phi(q-q_i,j)+d_{ji}, & \text{otherwise} \end{cases} \tag{5.11}$$

Given the function $h$, function $f$ and $\phi$ can be computed for the given $q$ as follows:

$$\begin{cases} f(q,i) = & \min_{j\neq i}\{h(j,i)\} \\ \pi(q,i) = & j^* \end{cases} \tag{5.12}$$

where $j^*$ is the index of $j$, the predecessor, corresponding to the above minimum;

$$\begin{cases} \phi(q,i) = & \min_{k\neq\pi(q,i),k\neq i}\{h(k,i)\} \\ \gamma(q,i) = & k^* \end{cases} \tag{5.13}$$

where $k^*$ is the value of $k$ corresponding to the above minimum. The initialization of $f,\phi,\pi$ and $\gamma$ is $f(q_i)=\phi(q,i)=\inf$, for $q$ such that $q\neq q_i$ and:

$$\text{for } q \text{ such that } q=q_i \begin{cases} f(q,i) = & d_{0i} \\ \pi(q,i) = & 0 \\ \phi(q,i) = & \infty \end{cases} \tag{5.14}$$

Informally, we can say that the method builds a minimum non-elementary path, delivering $q$ quantity of goods with a dynamic programming recursion that adds an edge from $j$ only to the nodes that don't have $j$ itself as direct predecessor, finding a path without loops of two nodes.

The algorithm can be described as follows:

**Algorithm** *Q PATHS*$(N,Q,\mathbf{q_i},\mathbf{d})$

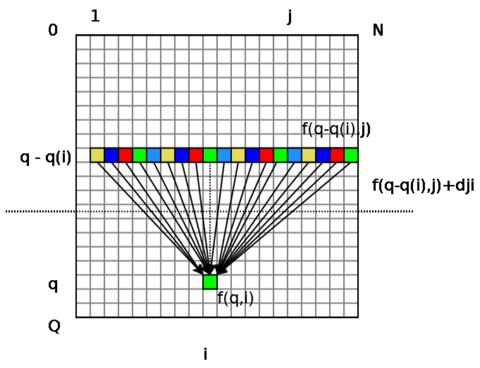
```
1.  // Data Structures
2.  $\mathbf{f}$, $\boldsymbol{\phi}$, $\boldsymbol{\pi}$, $\boldsymbol{\gamma}$
```

```
3.  // Initialization
4.  for $q = 0$ to $Q$ do
5.      for $i = 1$ to $N$ do
6.          if $q_i(i) == q$
7.             then $f(q,i) = d(0,i)$, $\pi(q,i) = 0$
8.                  $\phi(q,i) = \infty$, $\gamma(q,i) = \infty$
9.             else $f(q,i) = \infty$, $\pi(q,i) = -1$
10.                 $\phi(q,i) = \infty$, $\gamma(q,i) = \infty$
11. // q-Paths
12. for $q = 0$ to $Q$ do
13.     for $i = 1$ to $N$ do
14.         for $j = 1$ to $N$ do
15.             if $\pi(q - q_i(i)) \neq i$
16.                then $h(j,i) = f(q - q(i), j) + d(j,i)$
17.                else $h(j,i) = g(q - q(i), j) + d(j,i)$
18.     //Minima calculation for each $i$
19.     $f(q,i) = \min_{j \neq i} \{h(j,i)\}$
20.     $\pi(q,i) = j^*$
21.     $\phi(q,i) = \min_{k \neq \pi(q,i), k \neq i} \{h(k,i)\}$
22.     $\gamma(q,i) = k^*$
23. return f, $\boldsymbol{\phi}$, $\boldsymbol{\pi}$, $\boldsymbol{\gamma}$
```

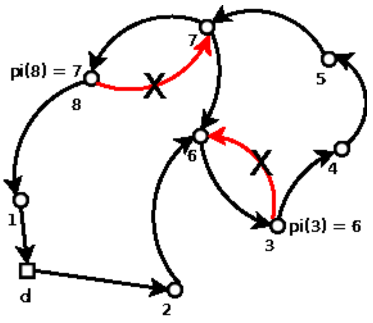
Using this method we can find shortest paths in respect of the request of each node $i$ and the vehicle's capacity Q, but the routes, also avoiding loops of two vertices, are not yet elementary (Figure 5.1b). The q-route relaxation can be computed in pseudo-polynomial time with a complexity of $\mathcal{O}(n^2Q)$. In figure 5.1a we can observe graphically the computation of a single $f(q,i)$ value. In the case of asymmetric VRP, where $d_{ij} \neq d_{ji}$, we compute the relaxation once for the $d$ matrix and once for its transpose $d^T$.

### 5.3.2 through-q-Route Relaxation

The through-q-route relaxation [74] is an enhancement for the q-route relaxation. In fact using the $f$ and the $g$ function we can find a better route mixing two paths retrieved by the q-path relaxation. Formally: Let $\phi(q,i)$ be the value of the least cost route, without loops, starting at the depot, passing through $i$ and finishing



(A) f(q,i) q-path computation



(B) q-path avoiding 2 vertices loop

FIGURE 5.1: f(q,i) q-path computation and 2-vertices loops avoiding.

back at the depot with a total load of $q$. This kind of route is a through-q-route. The $\psi(q,i)$ values are computed as follows:

$$\psi(q,i) = \min_{q_i \le \bar{q} \le (q+q_i)/2} \begin{cases} f(\bar{q},i) + f(q+q_i-\bar{q},i), & \text{if } \pi(\bar{q},i) \neq \pi(q+q_i-\bar{q},i) \\ \min \begin{cases} f(\bar{q},i) + \phi(q+q_i-\bar{q},i), \\ \phi(\bar{q},i) + f(q+q_i-\bar{q},i) \end{cases} & \text{otherwise} \end{cases} \tag{5.15}$$

The algorithm can be described as follows:

```
Algorithm THROUGH-Q ROUTES($N, Q, \mathbf{q_i}, \mathbf{f}, \boldsymbol{\phi}, \boldsymbol{\pi}, \boldsymbol{\gamma}$)
1.  // Data Structures
2.  $\boldsymbol{\psi}$
3.  // Initialization
4.  for $q = 0$ to $Q$ do
5.      for $i = 1$ to $N$ do
6.          $\psi(q,i) = \infty$
7.  // through q-routes
8.  min1, min2 // Minima
9.  for $i = 1$ to $N$ do
10.     $min1 = \infty$, $min2 = \infty$
11.     for $q = 0$ to $Q$ do
12.         for $q_i(i) \le \bar{q} \le (q + q_i(i))/2$ do
```

```
13.              $back = (q + q_i(i))/2 - \bar{q}$
14.              if $\pi(\bar{q}, i) \neq \pi(back, i) \wedge f(\bar{q}, i) + f(back, i) \leq min1$
15.                then $min = f(\bar{q}, i) + f(back, i)$
16.                else if $f(\bar{q}, i) + \phi(back, i) \leq \phi(\bar{q}, i) + f(back, i)$
17.                       then $min2 = f(\bar{q}, i) + \phi(back, i)$
18.                            if $min2 \leq min1$
19.                              then min1 = min2
20.                       else $min2 = \phi(\bar{q}, i) + f(back, i)$
21.                            if $min2 \leq min1$
22.                              then min1 = min2
23.          // $\psi$ Update
24.          $\psi(q, i) = min1$
25. return $\boldsymbol{\psi}$
```

More informally, this recurrence selects, among all the paths for a given $q$, the best combination of paths that starts and end to the deposit. Once computed the q-route relaxation, the through-q-route function $\psi(q, i)$ can be computed in pseudo-polynomial time with a complexity of $\mathcal{O}(n^2Q^2)$.

In the asymmetric case, we will use the $f$ and $\phi$ functions with the $\pi$ and the $\gamma$ computed by the asymmetric q-path described above. We will call $f^{fw}$ and $\phi^{fw}$ the functions obtained from q-path computation using the $d$ matrix, along with the predecessors matrices $\pi^{fw}$ and $\gamma^{fw}$. In the same fashion we will call $f^{bw}$, $\phi^{bw}$, $\pi^{bw}$ and $\gamma^{bw}$ the one computed with the $d^T$ matrix. In this case, we can extend the algorithm as follows:

**Algorithm** *ASY THROUGH-Q ROUTES*$(N, Q, \mathbf{q_i}, \mathbf{f^{fw}}, \boldsymbol{\phi}^{\mathbf{fw}}, \boldsymbol{\pi}^{\mathbf{fw}}, \boldsymbol{\gamma}^{\mathbf{fw}}, \mathbf{f^{bw}}, \boldsymbol{\phi}^{\mathbf{bw}}, \boldsymbol{\pi}^{\mathbf{bw}}, \boldsymbol{\gamma}^{\mathbf{bw}})$

```
1. // Data Structures
2. $\boldsymbol{\psi}$
3. // Initialization
4. for $q = 0$ to $Q$ do
5.     for $i = 1$ to $N$ do
6.         $\psi(q, i) = \infty$
7. // through q-routes
8. min1, min2 // Minima
```

```
9.  for $i = 1$ to $N$ do
10.     $min1 = \infty$, $min2 = \infty$
11.     for $q = 0$ to $Q$ do
12.         for $q_i(i) \leq \bar{q} \leq (q + q_i(i))/2$ do
13.             $back = (q + q_i(i))/2 - \bar{q}$
14.             if $\pi^{fw}(\bar{q}, i) \neq \pi^{bw}(back, i) \wedge f^{fw}(\bar{q}, i) + f^{bw}(back, i) \leq min1$
15.                then $min = f^{fw}(\bar{q}, i) + f^{bw}(back, i)$
16.                else  if $f^{fw}(\bar{q}, i) + \phi^{bw}(back, i) \leq \phi^{fw}(\bar{q}, i) + f^{bw}(back, i)$
17.                        then $min2 = f^{fw}(\bar{q}, i) + \phi^{bw}(back, i)$
18.                             if $min2 \leq min1$
19.                                then min1 = min2
20.                        else  $min2 = \phi^{fw}(\bar{q}, i) + f^{bw}(back, i)$
21.                             if $min2 \leq min1$
22.                                then min1 = min2
23.         // $\psi$ Update
24.         $\psi(q, i) = min1$
25. return $\psi$
```

### 5.3.3 ng-Route Relaxation

Righini and Salani [83–85] proposed a DP relaxation based on the construction of elementary paths in an decreasing state space, Baldacci et al. [77] proposed a more effective relaxation for the pricing problem, generalizing the Righini's idea, the ng-route relaxation. The main problem afflicting the other methods described above is that allow cycles longer than two vertices. Procedures to avoid bigger loops are computationally expensive and can address loops of three or four vertices only. The ng-route relaxation partially solves this problem introducing a supplementary information for the DP algorithm, allowing the recurrence to 'remember' an arbitrary number of nodes during the state-expansion phase and avoiding the creation of loops with a significant cardinality of nodes.

The algorithm has specific rules according to the different kind of Vehicle Routing problem in which is applied (VRPTW, CVRP..). We take in consideration only the one afferent to the Capacitated Vehicle Routing Problem (CVRP). The ng-route relaxation can be described as follows: Let $N_i \subseteq V$ be a set of selected customers for vertex $i$ (according to some criterion), such that $i \in N_i$ and $|N_i| \leq \Delta(N_i)$, where $\Delta(N_i)$ is the cardinality of the selected neighbors for $i$ plus $i$ itself. The sets $N_i$ allow us to associate with each path $P = (0, i_1, \dots, i_k)$ the subset $\Pi(P) \subseteq V(P)$ containing customer $i_k$ and every customer $i_r, r = 1, \dots, k-1$ of $P$ that belongs to all set $N_{i_{r+1}}, \dots, N_{i_k}$ associated with the customer $i_{r+1}, \dots, i_k$ visited after $i_r$. The set $\Pi(P)$ is defined as:

$$\Pi(P) = \left\{ i_r : i_r \in \bigcap_{s=r+1}^{k} N_{i_s}, r = 1, \dots, k-1 \right\} \bigcup \{i_k\}. \tag{5.16}$$

A ng-path $(q, i, NG)$, is a non-necessarily elementary path $P = (0, i_1, \dots, i_{k-1}, i_k = i)$ starting from the depot, visiting a subset of customers (even more than once) such that $NG = \Pi(P)$, ending at customer $i$ such that $i \notin \Pi(P')$ where $P' = (0, i_1, \dots, i_{k-1})$. We indicate with $f(q, i, NG)$ the cost of the least cost ng-path $(q, i, NG)$. We define an ng-route an ng-path $(q, i, NG)$ plus the edge from $i$ to the depot and the cost of a ng-route $f(q, 0, NG) = f(q, i, NG) + d_{i0}$. Functions $f(q, i, NG)$ can be computed on the graph defined as $H = (\Phi, \Psi)$ where:

$$\Phi = \left\{ (i, q, NG) : q_i \leq q \leq Q, \ \forall NG \subseteq N_i \ s.t. \ i \in NG \ \wedge \sum_{j \in NG} q_j \leq q, \forall i \in V' \right\}, \tag{5.17}$$

$$\Psi = \left\{ ((j, q', NG'), (i, q, NG)) : \forall (j, q', NG') \in \Psi^{-1}(i, q, NG), \forall (i, q, NG) \in \Phi \right\}, \tag{5.18}$$

where

$$\Psi^{-1}(i, q, NG) = \left\{ (j, q - q_i, NG') : \forall NG' \subseteq N_j \ s.t. \ j \in NG' \ and \ NG' \cap N_i = NG / \{i\}, \ \forall j \in \Gamma_i^{-1} \right\} \tag{5.19}$$

The function $f(i, q, NG)$ can be computed using the DP recursion:

$$f(i, q, NG) = \min_{(j, q', NG') \in \Psi^{-1}(i, q, NG)} \left\{ f(j, q', NG') + d_{ji} \right\}, \forall (i, q, NG) \in \Phi. \tag{5.20}$$

Is necessary to notice that the $\Delta$ parameter is critical for this relaxation: bigger is the cardinality of the neighborhood set, better is the bound obtained. The set's cardinality brings, inevitably to a combinatorial explosion of the recursion's states. Martinelli, Pecin and Poggi [78] brilliantly resolve this problem mixing the Decremental State Space relaxation proposed by Righini with the computation of the ng-route relaxation, using the exact relaxation only when necessary, allowing to an heuristic procedure based on the q-route relaxation to find the most promising routes to insert in the CG algorithm.

```
Algorithm NG-PATHS($N, Q, \mathbf{d}, \mathbf{q_i}, \mathbf{N_i}$)
1.  // Data Structures
2.  f, $\boldsymbol{\pi_n}$, $\boldsymbol{\pi_{ng}}$
3.  // Initialization
4.  for $q = 0$ to $Q$ do
5.      for $i = 1$ to $N$ do
6.          for $ng = 0$ to $|NG_{list}(i, q)|$ do
7.              if $q_i(i) == q$
8.                 then $NG = i, NG_{list}(q, i).Add(NG)$
9.                      $f(q, i, NG) = d(0, i)$, $\pi_n(q, i, NG) = 0$, $\pi_{ng}(q, i, NG) = 0$
10.                else $f(q, i, NG) = \infty$, $\pi_n(q, i, NG) = -1$, $\pi_{ng}(q, i, NG) = -1$
11. // ng-Paths
12. for $q = 0$ to $Q$ do
13.     for $i = 1$ to $N$ do
14.         for $j = 1$ to $N$ do
15.             for $ng = 0$ to $|NG_{list}(q - q_i(i), j)|$do
16.                 $NG = NG_{list}(q - q_i(i), j, ng)$
17.                 if $i \notin N_i \cup NG$
18.                    then $NG_{new} = N_i \cap NG \cup i$
19.                         if $f(q - q_i(i), j, NG) + d_{ji} \leq f(q, i, NG_{new})$
20.                            then $f(q, i, NG_{new}) = f(q - q_i(i), j, NG) + d_{ji}$
21.                                 Add $NG_{new}$ to $NG_{list}(i, q)$
22.                                 $\pi_n(q, i, NG_{new}) = j$
23.                                 $\pi_{ng}(q, i, NG_{new}) = NG_{list}(q - q_i(i), j, ng)$
24. return f, $\boldsymbol{\pi_n}$, $\boldsymbol{\pi_{ng}}$
```

In the algorithm we have introduced $\pi_n$ and $\pi_{ng}$ that keep track of the predecessor node $j \in \Gamma_i^{-1}$ and the predecessor path $\Pi(P)$, $P = \{0, i_1, \ldots, i_{k-1}\}$, respectively.

We also introduce the dominance among the labels in the recursion. In fact we can obtain the same label $(i, q, NG)$ expanding from two different predecessor state $(j, q - q_j, NG_j)$ and $(k, q - q_k, NG_k)$. Obviously, the dominant label is the one with the lower function value $f$.

The asymmetric extension for this relaxation is trivial: we can compute the $f^{fw}(i, q, NG)$ using the $d$ matrix and the $f^{bw}(i, q, NG)$ function using its transpose $d^T$.

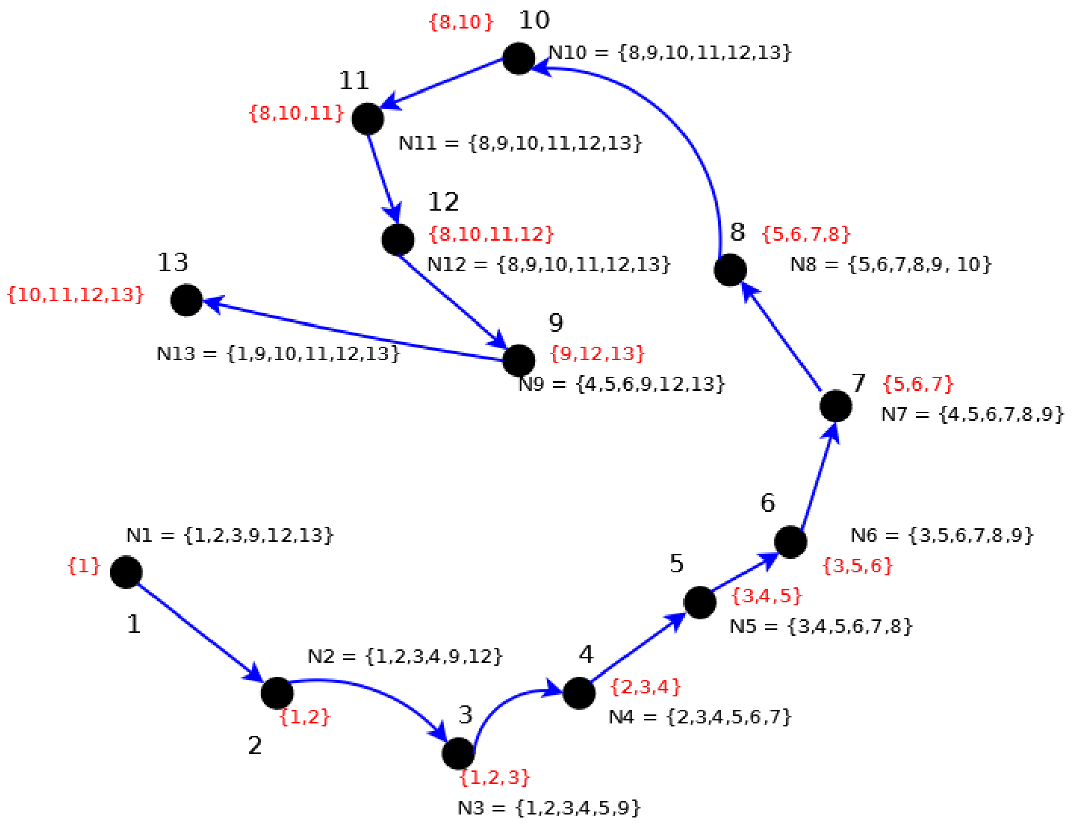


FIGURE 5.2: ng-Path example.

## 5.4 Parallel Relaxations on GPU

In this section we describe the parallel algorithms designed to run these relaxations on a GPU. Dynamic Programming algorithms ported on this kind of devices have given good results in many application: Boyer a et al. [58] proved that on a consumer GPU is possible to obtain a 20X speed-up factor for solving the Knapsack

Problem. Harish et al. [86] and Buluç et al. [87] provided an extensive spectrum of graph problems (Deep First Search, etc..) having advantages from the utilization of a GPU. Ortega-Arranz et al. [88] and Kumar et al. [89] proved that also algorithms like the Bellman-Ford and Dijkstra ones for solving the Single Source Shortest Path Problem (SSSPP) can be enhanced by the use of a many-core processor.

The GPGPU has given remarkable results also for the All Pairs Shortest Path Problem (APSPP), solved by means of the Floyd-Warshall algorithm, Katz et al. [56] and Lund et al. [57]. Maniezzo et al. [80] proved the effectiveness of many-cores platform also for supply chain's problems. As we can see, all the cited methods are based on the Dynamic Programming paradigm; indeed, the fine calculation granularity and the matrix-like data structures characterizing often these algorithms, fit particularly well on the many-cores architecture, where every thread execute a, relatively simple, computational kernel. We decided to use the CUDA parallel programming model because is, actually, the best trade off among portability, usability and reliability. Nvidia, also, provides a good environment for debugging and profiling applications (Nsight debugger for Microsoft Visual Studio, etc..) with effective functionalities.

## 5.4.1 GPU q-Route

The first relaxation that we consider is the q-route relaxation. In the next subsections we will expose the parallelism inside the method and we will describe the proposed algorithm for the GPU.

#### 5.4.1.1 Exposing Parallelism

The equations 5.11 and 5.12 describe the expansion rules for the creation of a new state $f(q, i)$. As we can see, this is a *backward* recursion, because we create the new state from the previous stages. This peculiarity enables a very interesting effect inside the recursion: all the states $f(q, i) \in q$ stage, can be calculated independently using all the $f(q - q_i, j), j \in \Gamma_i^{-1}$, $i \in V, i, j = 1, 2, \ldots, N$ states of the $q - q_i$ stage. In fact, we can evaluate at the same time all the $f(q, i)$ states inside the $q$ stage, as depicted in figure 5.3.

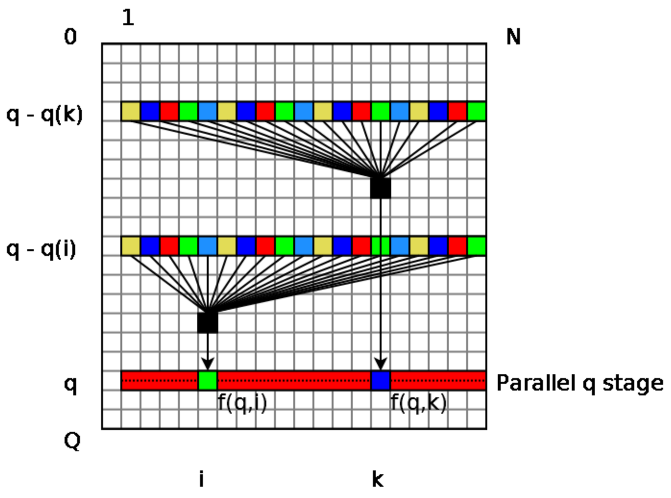


FIGURE 5.3: Stage q parallel computation.

#### 5.4.1.2 Algorithm Description

In this section we will provide the pseudo-code for the parallel algorithm and the computation kernel for the q-path algorithm. According to the CUDA programming model, we can assign a threads block for each state $f(q, i)$ and $T$ threads for each block. We decided to assign in this manner the workload for allowing us to compute in parallel not only the state of the recursion (intra-grid parallelism), but also the *min* operation required to compute it. In fact, we will use the threads of each block to evaluate in parallel the reductions (min operation) described in the equations 5.12 and 5.13 (intra-block parallelism) using the method suggested in [59]. All the data structures have linear access to each element, it means that all matrices are stored in a row-major fashion. For the legibility of the algorithm, we decided to use two indexes for the matrices anyway.

```
Algorithm GPU Q-PATHS($N, Q, \mathbf{q_i}, \mathbf{d}$)
1.  // Data Structures
2.  $\mathbf{f}$, $\boldsymbol{\phi}$, $\boldsymbol{\pi}$, $\boldsymbol{\gamma}$
3.  // Kernel Setup
4.  BLOCKS $B = N - 1$, THREADS $T$, SHARED-MEM $Sh[2 * (N - 1)]$
5.  // Main Loop
6.  for $q = 0$ to $Q$ do
7.      Q-PATHS-KERNEL<<< $B, T, Sh$ >>>($Q$, $q$, $\boldsymbol{q_i}$, $\boldsymbol{d}$, $\boldsymbol{f}$, $\boldsymbol{\phi}$, $\boldsymbol{\pi}$, $\boldsymbol{\gamma}$)
8.  return $\mathbf{f}$, $\boldsymbol{\pi}$, $\boldsymbol{\phi}$, $\boldsymbol{\gamma}$
```

**Algorithm** *Q-PATHS-KERNEL*$(Q, N, q, \mathbf{q_i}, \mathbf{d}, \mathbf{f}, \boldsymbol{\phi}, \boldsymbol{\pi}, \boldsymbol{\gamma})$

```
1.  min1, min2 // Variables Initialization
2.  pred1,pred2
3.  $\boldsymbol{h_{sh}}[N-1]$, $\boldsymbol{\pi_{sh}}[N-1]$
4.  $thdidx = threadIdx.x$, $Nthds = blockDim.x$, $nodeidx = blockIdx.x$
5.  $slack = N\%Nthds$, $times = N/Nthds$
6.  if $thdidx \leq slack$
7.     then $times++$
8.  // Shared Memory Initialization
9.  for $t = 0$ to $times$ do
10.       $h_{sh}[thdidx + t * Nthds] = \infty$
11.       $\pi_{sh}[thdidx + t * Nthds] = -1$
12. syncthreads()
13. // Partial Minima Computation
14. for $t = 0$ to $times$ do
15.       $j = thdidx + t * Nthds$
16.       if $\pi(q - q_i(nodeidx), j) \neq nodeidx$
17.          then $h_{sh}[j] = f(q - q_i(nodeidx), j) + d(j, i)$
18.               $\pi_{sh}[j] = j$
19.          else $h_{sh}[j] = \phi(q - q_i(nodeidx), j) + d(j, i)$
20.               $\pi_{sh}[j] = j$
21. syncthreads()
22. // First reduction to get $f(q, i)$ value
23. GPU REDUCTION($h_{sh}$, $\pi_{sh}$)
24. $min1 = h_{sh}[0]$, $pred1 = \pi_{sh}[0]$, $h_{sh}[0] = \infty$, $\pi_{sh}[0] = -1$
25. // Second reduction to get $\phi(q, i)$ value
26. GPU REDUCTION($h_{sh}$, $\pi_{sh}$)
27. $min2 = h_{sh}[0]$, $pred2 = \pi_{sh}[0]$
28. // Matrices update
29. if $nodeidx \neq thdidx$
30.    then $f(q, i) = min1$, $\pi(q, i) = pred1$
31.         $\phi(q, i) = min2$, $\gamma(q, i) = pred2$
```

**Algorithm** *GPU REDUCTION*$(\mathbf{h_{sh}}, \boldsymbol{\pi_{sh}}, times)$

```
1.  // Keeping only the most promising value for each thread
2.  for $t = 0$ to $times$ do
```

```
3.       if $h_{sh}[thdidx] > h_{sh}[thdidx + t * Nthds]$
4.         then $f_swap = h_{sh}[thdidx]$
5.              $h_{sh}[thdidx] = h_{sh}[thdidx + t * Nthds]$
6.              $h_{sh}[thdidx + t * Nthds] = f_swap$
7.              // Update Predecessor
8.              $\pi_swap = \pi_{sh}[thdidx]$
9.              $\pi_{sh}[thdidx] = \pi_{sh}[thdidx + t * Nthds]$
10.             $\pi_{sh}[thdidx + t * Nthds] = \pi_swap$
11. syncthreads()
12. // Reduction
13. for $s = Nthds/2$ to $0,s/ = 2$ do
14.      if $thdidx < s$
15.        then if $h_{sh}[thdidx + s] < h_{sh}[thdidx]$
16.               then $f_swap = h_{sh}[thdidx]$
17.                    $h_{sh}[thdidx] = h_{sh}[thdidx + s]$
18.                    $h_{sh}[thdidx + s] = f_swap$
19.                    // Update Predecessor
20.                    $\pi_swap = \pi_{sh}[thdidx]$
21.                    $\pi_{sh}[thdidx] = \pi_{sh}[thdidx + s]$
22.                    $\pi_{sh}[thdidx + s] = \pi_swap$
23.             syncthreads()
```

The lines 6-8 of the main procedure, *GPU Q-PATHS*, is the main loop of the algorithm. Indeed, inside the for cycle we call, iteratively, for each stage $q$ of the Dynamic Programming recurrence the computation kernel for the GPU. We also define the dimension of the shared memory for each block of the grid. The dimension is designed to store all the $\left|\Gamma_i^{-1}\right|$ entries for the $h(i)$ vector defined in the equation 5.11 and the predecessor node for each entry (in our case, the number of predecessor is equal to $N - 1$ supposing that the digraph $G$ is complete). Once initialized the shared memory (lines 9-11 of the *Q-PATHS-KERNEL* procedure), we calculate for each predecessor its function value and we store it in the $h_{sh}$ array together with the node's index in $\pi_{sh}$ (lines 14-21) according with the 5.11 equation.

In lines 23 and 26 we calculate the minima values for the $f$ and the $\phi$ functions. In order to compute these values, we call twice the *GPU REDUCTION* procedure. This procedure, a device function in the implementation, is based on the one described in [59], we modified it in order to keep trace of the values inside the shared memory, swapping instead of overwriting the values themselves. In lines 2-10 we pre-calculate the significant values for each thread, keeping only the most promising, and then we perform the reduction to find the minimum (lines 13-23). Every thread compute one or more values thanks to the indices that we assigned in the lines 4-7 of the main kernel, calculating the occurrence of the thread index inside the total number of the problem nodes. Finally, in lines 29-31, we update the data structures with the new function values. In line 24 of the main kernel, we reset the result of the first reduction and we store the first minimum and its predecessor.

### 5.4.2 GPU through-q-route

In this section we will describe the parallel algorithm for the through-q-route relaxation. This method, as described in 5.3.2, is based on the function values obtained from the q-paths relaxation. In order to minimize the memory transaction between the CPU and the GPU, we will compute the q-path and, keeping the results on the GPU memory, we will perform the GPU algorithm for the through-q-routes.

#### 5.4.2.1 Exposing Parallelism

According to the equation 5.15, we can see the nature of the computation is strictly combinatorial and each $\psi(q, i)$ function value is independent from the others. We decided to compute in parallel all the function values for each $i$ node of the graph (each column of the matrix) which is the dimension, the quantity dimension $q$, often bigger and computationally more expensive. In the next paragraph we will provide the pseudo-code for the GPU kernel and we will give a brief description of the algorithm.

#### 5.4.2.2 Algorithm Description

In this section we will describe the GPU kernel designed to compute the through-q-route relaxation on a GPU.

**Algorithm** *GPU THROUGH-Q ROUTES*$(N, Q, \mathbf{f}, \mathbf{q_i}, \boldsymbol{\phi}, \boldsymbol{\pi}, \boldsymbol{\gamma})$

```
1.  // Data Structures
2.  $\boldsymbol{\psi}$
3.  // Kernel Setup
4.  BLOCKS B = Q, THREADS T, SHARED-MEM Sh[T]
5.  // Main Loop
6.  for $i = 1$ to $N$ do
7.      THROUGH-Q-ROUTES-KERNEL<<< $B, T, Sh$ >>>($Q$, $i$, $\boldsymbol{q_i}$, $\boldsymbol{d}$, $\boldsymbol{f}$,
        $\boldsymbol{\phi}$, $\boldsymbol{\pi}$, $\boldsymbol{\gamma}$, $\boldsymbol{\psi}$)
8.  return $\boldsymbol{\psi}$
```

**Algorithm** *THROUGH-Q-ROUTES-KERNEL*$(Q, N, i, \mathbf{q_i}, \mathbf{f}, \boldsymbol{\phi}, \boldsymbol{\pi}, \boldsymbol{\gamma}, \boldsymbol{\psi})$

```
1.  $\boldsymbol{sh}[T]$ // Shared memory
2.  $NThds = blockDim.x$, $thdidx = threadIdx.x$
3.  $q = blockIdx.x$, $startidx = q_i(i)$, $endidx = (q + q_i(i))/2$, $diff = endidx -$
    $startidx$
4.  // Shared Memory Initialization
5.  $sh[thdidx] = \infty$
6.  $times = diff/NThds$, $slack = diff\%NThds$
7.  if $thdidx < slack$
8.    then $times++$
9.  syncthreads()
10. for $t = 0$ to $times$ do
11.     $\bar{q} = thdidx + startidx + (t * NThds)$
12.     if $\pi(\bar{q}, i) \neq \pi(q + q_i(i) - \bar{q}, i)$
13.       then $f_{new} = f(\bar{q}, i) + f(q + q_i(i) - \bar{q}, i)$
14.            if $sh[thdidx] > f_{new}$
15.              then $sh[thdidx] = f_{new}$
16.       else $\phi_a = f(\bar{q}, i) + \phi(q + q_i(i) - \bar{q}, i)$
17.            $\phi_b = \phi(\bar{q}, i) + f(q + q_i(i) - \bar{q}, i)$
18.            $\phi_{new}$=0, $(\phi_a < \phi_b)?\phi_{new} = \phi_a : \phi_{new} = \phi_b$
19.            if $sh[thdidx] > \phi_{new}$
20.              then $sh[thdidx] = \phi_{new}$
21. syncthreads()
22. // Parallel reduction
```

```
23. for $s = Nthds/2$ to $0, s/ = 2$ do
24.     if $thdidx < s$
25.       then if $h_{sh}[thdidx + s] < h_{sh}[thdidx]$
26.               then $h_{sh}[thdidx] = h_{sh}[thdidx + s]$
27.            syncthreads()
28. syncthreads()
29. // Data Update
30. if $thdidx == 0$
31.    then $\psi(q, i) = sh[0]$
```

In line 3 we define the indices of the data for the $q$ block. The $diff$ variable describes the range of indices for the block. In line 5 we initialize the shared memory and in lines 6-8, as for the q-paths kernel, we define the indices for each thread of the block. In lines 10- 20 we compute the partial results and we store them in the shared memory, according to the equation 5.15; in line 11 we define the indices $\bar{q}$ for each thread. Once computed the partial results, we can perform a standard parallel reduction (lines 23-28) in the shared memory as described in [59]. In this case we don't need to keep all the values inside the shared memory, in fact we need only the minimum for each state. Finally, the thread 0 updates the $\psi(q, i)$ function value (lines 30-31).

### 5.4.3 GPU ng-Route

Unlike the other relaxations, the ng-route is more computationally expensive but numerically is more effective than the previous two. The main problems afflicting this method is an efficient management of the NG sets and the dominance among them. In fact, the NG set and its cardinality for each $f(q, i)$ stage is dynamic. Dynamic data structures in a GPGPU environment are not desirable, indeed searches and the management inside these structures are performance killers. In the next sections we will describe our strategies for addressing these problems and then a parallel algorithm for the GPU.

#### 5.4.3.1 Exposing Parallelism

The first problem that emerges for the porting of this relaxation on a many-cores platform is the dynamic nature of the NG 'dimension' of the recurrence. A static data structure is almost mandatory for exploiting the computation capabilities of these devices. We addressed this problem in this way: As we can see, from the equations 5.17 and 5.19, the $NG$ set for each stage $(i, q)$ is completely contained in the $N_i$ set chosen for the $i$ node. Plus, in each $NG$ set the relative $i$ node is contained. From these assumptions we can state that the complete enumeration for all the possible NG sets in the $(i, q)$ stage is the $NG_i^{\mathcal{P}} \in \mathcal{P}(N_i)$ set, that is a subset of the power set of $N_i$ composed only by the subset with $i$.

The cardinality of the power set of $N_i$ is $2^{\Delta(N_i)}$ but for the $NG_{dim}$ the cardinality is $2^{\Delta(N_i)-1}$, because we are taking into account only the sets with $i$. For reasonable $\Delta(N_i)$ (10-14) we can easily enumerate all the possible sets for the $NG$ dimension, allowing us to make static this dimension too. As we illustrate in figure 5.4 it's possible to describe all the states and stages of the recursion like a 3-dimensional cube: vehicle's capacity $Q$, customers/nodes $i$ and sets' indices $NG$. In the third dimension $NG$ we consider only the indices of these sets, as we will describe in the following sections, we can index these sets and use these indices to define the dimension. To our purposes, in order to expose the parallelism of the method, we can reformulate the equation 5.20 in its *forward* form:

$$f(k, q + q_i, NG) = \min_{(i,q,NG') \in \Psi(k,q+q_i,NG)} \{f(i, q, NG) + d_{ik}\}, \forall (k, q + q_i, NG) \in \Phi. \tag{5.21}$$

Exploiting the equation 5.21 we can easily observe that we can compute independently all the states $f(k, q + q_k, NG)$ from all the states $(i, q, NG')$ in the $q$ set of stages.

#### 5.4.3.2 Dominance Management

To easily address the management of the dominances among the states during the expansion, we decided to pre-calculate the transitions among the $NG$ sets of each node. We basically create a *transition map* that taking in input the $NG$ set of the starting node, the index of the starting node $j$ and the end node $i$, gives in output the index of the new $NG$ set among the enumerated ones of $i$. We indexed the $NG$ sets for each node $i$ exploiting a bitmap. As we described before, we enumerate all the possible $NG$ sets for each node obtained from the relative $N_i$ using its power set. We define the *mask* for the $NG$ set as: $mask = \{b_0, b_1, \dots, b_h\}, h = 2^{\Delta(N_i)-1}, b \in \{0,1\}$ and the mask for each $NG$ is defined:

$$mask(h) = \begin{cases} 1 & \text{if } NG(k) \in Ni, k = 0, \dots, |NG| \\ 0 & \text{otherwise} \end{cases} \tag{5.22}$$

Based on this mask, we can define the *index* for the $NG$:

$$NG_{index} = \sum_{h=0}^{\Delta(N_i)-1} b * 2^h \tag{5.23}$$

For example: Given $NG = \{7,2,4,6\} \in NG_7^{\mathcal{P}}$, $N_7 = \{7,2,4,6,8,10\}$, $\Delta(N_7) = 6$. We will have: $mask_{NG} = \{1,1,1,1,0,0\}$ and the index: $NG_{index} = 1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 1 * 2^3 + 0 * 2^4 + 0 * 2^5 = 1 + 2 + 4 + 8 + 0 + 0 = 15$.

We give, for each combination of the 1 and 0 of the $NG$ map, an univocal index. This univocal index gives us the possibility to know in advance which will be the index for the new $NG$ set created by the expansion to another state. Given the description for the index of each $NG$ set we can define the transition map among the $NG$ sets of each node: given the $NG_{index}$, the starting node $j$ and the $N_i$ set of the destination node $i$, the mapping function returns the index $NG'_{index}$ of the $NG' \in NG_i^{\mathcal{P}}$.

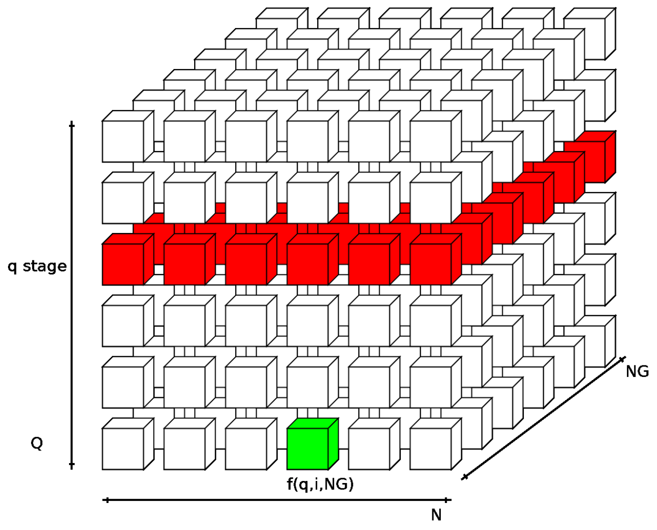


FIGURE 5.4: NG path 3-D states space.

#### 5.4.3.3 Active Sets

Exploiting the equation 5.16 we can pre-calculate the $NG \in NG_i^{\mathcal{P}}$ sets active during the recursion. In fact not all the $NG$ enumerated are effectively used by the method. For each stage $q$, we can retrieve all the $NG$ sets involved in the computation simply running the recursion, without computing $f$, once before the main part of the algorithm in which the ng-relaxation is used, exploiting the transition map defined in the previous paragraph. This property allows us to apply a modified version of what is called 'threads compaction', described in [86] and analyzed in [90]. This method consists in creating a mask for allowing to the GPU to spawn only the threads useful for the computation. Using this technique and exploiting the property described before, we can take in consideration only the states effectively useful for the relaxation and calibrate the device's resources on these, avoiding the overhead induced by not working threads. The active sets for each stage together with the indexing for the $NG$ sets also enhance the serial version of the method, indeed we avoided the overhead for the dominance and we reduced drastically the computation at each stage. We propose a modified version of the serial algorithm in the next paragraph.

#### 5.4.3.4 Algorithm Description

In this section we describe first the enhancement for the serial algorithm using the previous consideration, then we will propose a parallel kernel for the GPU.

**Algorithm** *NG-PATHS 2*$(N, Q, \mathbf{d}, \mathbf{q_i}, \mathbf{ActiveSets}, \mathbf{TransMap})$

```
1.  // Data Structures
2.  f, $\boldsymbol{\pi_n}$, $\boldsymbol{\pi_{ng}}$
3.  // Initialization
4.  for $q = 0$ to $Q$ do
5.      for $i = 1$ to $N$ do
6.          for $ng = 0$ to $|NG_{list}(i, q)|$ do
7.              if $q_i(i) == q$
8.                  then $f(q, i, 0) = d(0, i)$, $\pi_n(q, i, 0) = 0$, $\pi_{ng}(q, i, 0) = 0$
9.                  else $f(q, i, 0) = \infty$, $\pi_n(q, i, 0) = -1$, $\pi_{ng}(q, i, 0) = -1$
10. // ng-Paths
```

```
11. for $q = 0$ to $Q$ do
12.     for $i = 1$ to $N$ do
13.         for $j = 1$ to $N$ do
14.             $dist_{ij} = dist(i, j)$
15.             for $h = 0$ to $|ActiveSets(q - q_i(i), j)|$ do
16.                 $NG_{index} = ActiveSets(q - q_i, j, h)$
17.                 $NG'_{index} = TransMap(i, j, NG_{index})$
18.                 if $f(q, i, NG'_{index}) > f(q - q_i(i), j, NG_{index}) + dist_{ij}$
19.                    then $f(q, i, NG'_{index}) = f(q - q_i(i), j, NG_{index}) + dist_{ij}$
20.                         $\pi_n(q, i, NG'_{index}) = j$
21.                         $\pi_{ng}(q, i, NG'_{index}) = NG_{index}$
22. return f, $\boldsymbol{\pi_n}$ , $\boldsymbol{\pi_{ng}}$
```

In lines 15 and 16 we introduced the *ActiveSets* and *TransMap* data structures giving us the NG sets indices to update.

**Algorithm** *GPU NG-PATHS*$(N, Q, \mathbf{d}, \mathbf{q_i}, \mathbf{ActiveSets}, \mathbf{TransMap})$

```
1. // Data Structures
2. f, $\boldsymbol{\pi_n, \pi_{ng}, SLabels_N, SLabels_{NG}}$, ActThds
3. // The variables f, $\boldsymbol{\pi_n}$, $\boldsymbol{\pi_{ng}}$ are initialized in the same fashion of the serial
   algorithm
4. // Number of Active Threads for each set of stages $q$
5. for $q = 0$ to $Q$ do
6.     for $i = 1$ to $N$ do
7.         $ActThds(q) += |ActSets(q, i)|$ // Initialize StartLabels
8. for $q = 0$ to $Q$ do
9.     for $i = 1$ to $N$ do
10.        for $h = 0$ to $|ActSets(q, i)|$ do
11.            $NG = ActSets(q, i, h)$
12.            $SLabels_N(q)$.Add(i)
13.            $SLabels_{NG}(q)$.Add(NG)
14. //Main Loop
15. for $q = 0$ to $Q$ do
16.     // Kernel Setup
17.     $THDS = T$
```

```
18.        $BLK.y = N-1$, $BLK.x = |ActThds(q)|/THDS$
19.        NG-PATHS-KERNEL<<< $BLKS, THDS$ >>>
20.        $(q, \boldsymbol{q_i}, \boldsymbol{Map}, \boldsymbol{d}, \boldsymbol{f}, \boldsymbol{\pi_n}, \boldsymbol{\pi_{ng}}, \boldsymbol{ActSets}, \boldsymbol{SLabels_{Nodes}}, \boldsymbol{SLabels_{NG}})$
21.  return $\boldsymbol{f}, \boldsymbol{\pi_n}, \boldsymbol{\pi_{ng}}$
```

```
Algorithm NG-PATHS-KERNEL($q, \mathbf{q_i}, \mathbf{Map}, \mathbf{d}, \mathbf{f}, \boldsymbol{\pi}_n, \boldsymbol{\pi}_{ng}, \mathbf{ActSets}, \mathbf{SLabels_N}, \mathbf{SLabels_{NG}}$)
1.  $idx = blockIdx.x * blockDim.x + threadIdx.x$
2.  $i = blockIdx.y$
3.  $j = SLabels_N(q, idx)$, $NG_{index} = SLabels_{NG}(q, idx)$
4.  $dist = d(j, i)$
5.  $NG'_{index} = Map(i, j, NG_{index})$
6.  if $f(q + q_i(i), i, NG'_{index}) > f(q, j, NG_{index}) + dist$
7.     then $f(q + q_i(i), i, NG'_{index}) = f(q, j, NG_{index}) + dist$
8.          $\pi_n(q + q_i(i), i, NG'_{index}) = j$
9.          $\pi_{ng}(q + q_i(i), i, NG'_{index}) = NG$
```

Inside the main procedure, *GPU NG-PATHS*, in lines 5-7 we count the number of active labels for each stage $q$. Using this value, inside the main loop of the procedure, we spawn the necessary number of thread for each iteration of the loop (line 19).

In lines 8-13 we initialize the $StartLabels_{Nodes}$ and $StartLabels_{NG}$ structures, containing, for each $q$ the indices of the active NG for each node. The main loop is described in lines 15-20. As for the serial version of the algorithm, we return the $f$ array with the function values with the array for the predecessors node, $\pi_n$ and the array for the predecessor path $\pi_{ng}$ (line 21).

Inside the GPU kernel, *NG-PATHS-KERNEL*, in lines 1-3 we define the indices of the label, in line 5, using the transition map, we find the index if the new *NG* set for the expanded label and in lines 6-9 we update, if necessary, the new label. The operation in these lines are implemented using the *AtomicMin()* CUDA primitive to manage the concurrent update of a single variable by more threads simultaneously, in order to avoid race conditions and inconsistent results.

### 5.4.4 Asymmetric Relaxations

The asymmetric case for the VRP is characterized by $d_{ij} \neq d_{ji}$. In the case of q-path and ng-paths, we are forced to calculate twice the relaxation, once using the $d$ matrix (forward) and once using the $d^T$ transpose matrix (backward). For the through-q-route relaxation, we have to use the output from the asymmetric computation of the q-path, as described in *ASY THROUGH-Q ROUTES* algorithm. We propose an effective parallel approach to solve this version of the VRP exploiting all the parallel features of a GPU.

A *Stream* is a sequence of operations that execute in issue-order on the GPU. More intuitively, we can say that a GPU can execute concurrently multiple kernels and memory transactions, overlaying the operations (e.g. transferring data for the kernel 2 from the HOST to the GPU while executing the kernel 1). This is possible due to different engines managing the execution and memory transfer operations. Depending on the number of the simultaneous streams supported by the GPU, (typically 4) we can hide the memory transaction operation with the computation of a kernel and then compute the data loaded with another kernel. This feature allows us to compute concurrently the two relaxations (forward and backward) on the same GPU, introducing another level of parallelism (among kernels). In our case we don't use the streams to hide the memory transactions between the CPU and the GPU, but to execute the same kernel with different data on the same GPU, in a typical SIMD approach.

In the following we propose the pseudo-code for these algorithms. The kernels are the same described in the previous paragraph. The main difference is the use of the cudaMemcpyAsync() primitive that is a page-locked memory (pinned memory) for the characteristics of which we refer to the official documentation of CUDA.

```
Algorithm GPU-Q-PATHS ASY($N, Q, \mathbf{q_i}, \mathbf{d}, \mathbf{d^T}$)
1.  // Data Structures
2.  $\boldsymbol{f^{fw}}$, $\boldsymbol{\phi^{fw}}$, $\boldsymbol{\pi^{fw}}$, $\boldsymbol{\gamma^{fw}}$
3.  $\boldsymbol{f^{bw}}$, $\boldsymbol{\phi^{bw}}$, $\boldsymbol{\pi^{bw}}$, $\boldsymbol{\gamma^{bw}}$
4.  // Stream Initialization
5.  Stream $FW$, Stream $BW$
6.  // Kernel Setup
7.  BLOCKS $B = N - 1$, THREADS $T$, SHARED-MEM $Sh[2 * (N - 1)]$
8.  // Main Loop
9.  for $q = 0$ to $Q$ do
```

```
10.     Q-PATHS-KERNEL<<< $B, T, Sh, FW$ >>>($Q$, $q$, $\boldsymbol{q_i}$, $\boldsymbol{d}$, $\boldsymbol{f^{fw}}$, $\boldsymbol{\phi^{fw}}$,
        $\boldsymbol{\pi^{fw}}$, $\boldsymbol{\gamma^{fw}}$)
11.     Q-PATHS-KERNEL<<< $B, T, Sh, BW$ >>>($Q$, $q$, $\boldsymbol{q_i}$, $\boldsymbol{d^T}$, $\boldsymbol{f^{bw}}$, $\boldsymbol{\phi^{bw}}$,
        $\boldsymbol{\pi^{bw}}$, $\boldsymbol{\gamma^{bw}}$)
12. return $\boldsymbol{f^{fw}}$, $\boldsymbol{\pi^{fw}}$, $\boldsymbol{\phi^{fw}}$, $\boldsymbol{\gamma^{fw}}$, $\boldsymbol{f^{bw}}$, $\boldsymbol{\pi^{bw}}$, $\boldsymbol{\phi^{bw}}$, $\boldsymbol{\gamma^{bw}}$
```

```
Algorithm GPU NG-PATHS ASY($N, Q, \mathbf{d}, \mathbf{d^T}, \mathbf{q_i}, \mathbf{ActSets^{fw}}, \mathbf{ActSets^{bw}}, \mathbf{Map^{fw}}, \mathbf{Map^{bw}}$)
1.  // Data Structures
2.  $\boldsymbol{f^{fw}}$, $\boldsymbol{\pi_n^{fw}}$, $\boldsymbol{\pi_{ng}^{fw}}$, $\boldsymbol{SLabels_N^{fw}}$, $\boldsymbol{SLabels_{NG}^{fw}}$, $\boldsymbol{ActThds^{fw}}$
3.  $\boldsymbol{f^{bw}}$, $\boldsymbol{\pi_n^{bw}}$, $\boldsymbol{\pi_{ng}^{bw}}$, $\boldsymbol{SLabels_N^{bw}}$, $\boldsymbol{SLabels_{NG}^{bw}}$, $\boldsymbol{ActThds^{bw}}$
4.  // The variables $\mathbf{f}$, $\boldsymbol{\pi_n}$, $\boldsymbol{\pi_{ng}}$ are initialized in the same fashion of the serial algorithm
5.  // Number of Active Threads for each set of stages $q$
6.  for $q = 0$ to $Q$ do
7.      for $i = 1$ to $N$ do
8.          $ActThds^{fw}(q) += |ActSets^{fw}(q, i)|$
9.          $ActThds^{bw}(q) += |ActSets^{bw}(q, i)|$
10. // Initialize StartLabels
11. for $q = 0$ to $Q$ do
12.     for $i = 1$ to $N$ do
13.         for $h = 0$ to $|ActSets^{fw}(q, i)|$ do
14.             $NG = ActSets^{fw}(q, i, h)$
15.             $SLabels_N^{fw}(q)$.Add(i)
16.             $SLabels_{NG}^{fw}(q)$.Add(NG)
17.         for $h = 0$ to $|ActSets^{bw}(q, i)|$ do
18.             $NG = ActSets^{bw}(q, i, h)$
19.             $SLabels_N^{bw}(q)$.Add(i)
20.             $SLabels_{NG}^{bw}(q)$.Add(NG)
21. // Stream Initialization
22. Stream FW, Stream BW
23. //Main Loop
24. for $q = 0$ to $Q$ do
25.     // Kernel Setup
26.     $THDS = T$
27.     $BLK.y = N - 1$, $BLK.x = |ActThds^{fw}(q)|/THDS$
28.     NG-PATHS-KERNEL<<< $BLKS, THDS, 0, FW$ >>>
29.     ($N$, $Q$, $q$,$\boldsymbol{q_i}$, $\boldsymbol{Map_{fw}}$, $\boldsymbol{d}$, $\boldsymbol{f^{fw}}$, $\boldsymbol{\pi_n^{fw}}$, $\boldsymbol{\pi_{ng}^{fw}}$, $\boldsymbol{ActSets^{fw}}$,
30.     $\boldsymbol{SLabels_N^{fw}}$, $\boldsymbol{SLabels_{NG}^{fw}}$)
31.     $BLK.y = N - 1$, $BLK.x = |ActThds^{bw}(q)|/THDS$
32.     NG-PATHS-KERNEL<<< $BLKS, THDS, 0, BW$ >>>
33.     ($N$, $Q$, $q$, $\boldsymbol{q_i}$, $\boldsymbol{Map^{bw}}$, $\boldsymbol{d^T}$, $\boldsymbol{f^{bw}}$, $\boldsymbol{\pi_n^{bw}}$, $\boldsymbol{\pi_{ng}^{bw}}$, $\boldsymbol{ActSets^{bw}}$,
34.     $\boldsymbol{SLabels_N^{bw}}$, $\boldsymbol{SLabels_{NG}^{bw}}$)
```

```
35. return$\boldsymbol{f^{fw}}$, $\boldsymbol{f^{bw}}$, $\boldsymbol{\pi_n^{fw}}$, $\boldsymbol{\pi_{ng}^{fw}}$, $\boldsymbol{f_{bw}}$, $\boldsymbol{\pi_n^{bw}}$, $\boldsymbol{\pi_{ng}^{bw}}$
```

The through-q-route algorithm only changes the data structures in input:

**Algorithm** *GPU THROUGH-Q-ROUTES ASY*$(N, Q, \mathbf{q_i}, \mathbf{f^{fw}}, \boldsymbol{\phi^{fw}}, \boldsymbol{\pi^{fw}}, \boldsymbol{\gamma^{fw}}, \mathbf{f^{bw}}, \boldsymbol{\phi^{bw}}, \boldsymbol{\pi^{bw}}, \boldsymbol{\gamma^{bw}})$

```
1.  // Data Structures
2.  $\boldsymbol{\psi}$
3.  // Kernel Setup
4.  BLOCKS $B = Q$, THREADS $T$, SHARED-MEM $Sh[T]$
5.  // Main Loop
6.  for $i = 1$ to $N$ do
7.      THROUGH-Q-ROUTES-KERNEL ASY<<< $B, T, Sh$ >>>($Q$, $i$, $\boldsymbol{q_i}$, $\boldsymbol{f^{fw}}$, $\boldsymbol{\phi^{fw}}$, $\boldsymbol{\pi^{fw}}$,
        $\boldsymbol{\gamma^{fw}}$, $\boldsymbol{f^{bw}}$, $\boldsymbol{\phi^{bw}}$, $\boldsymbol{\pi^{bw}}$, $\boldsymbol{\gamma^{bw}}$, $\boldsymbol{\psi}$)
8.  return $\boldsymbol{\psi}$
```

**Algorithm** *THROUGH-Q-ROUTES-KERNEL ASY*$(Q, N, i, \mathbf{q_i}, \mathbf{f^{fw}}, \boldsymbol{\phi^{fw}}, \boldsymbol{\pi^{fw}}, \boldsymbol{\gamma^{fw}}, \mathbf{f^{bw}}, \boldsymbol{\phi^{bw}}, \boldsymbol{\pi^{bw}}, \boldsymbol{\gamma^{bw}}, \boldsymbol{\psi})$

```
1.  $\boldsymbol{sh}[T]$ // Shared memory
2.  $NThds = blockDim.x$, $thdidx = threadIdx.x$
3.  $q = blockIdx.x$, $startidx = q_i(i)$, $endidx = (q + q_i(i))/2$, $diff = endidx - startidx$
4.  // Shared Memory Initialization
5.  $sh[thdidx] = \infty$
6.  $times = diff/NThds$, $slack = diff\%NThds$
7.  if $thdidx < slack$
8.    then $times++$ syncthreads()
9.  for $t = 0$ to $times$ do
10.     $\bar{q} = thdidx + startidx + (t * NThds)$
11.     if $\pi^{fw}(\bar{q}, i) \neq \pi^{bw}(q + q_i(i) - \bar{q}, i)$
12.       then $f_{new} = f^{fw}(\bar{q}, i) + f^{bw}(q + q_i(i) - \bar{q}, i)$
13.            if $sh[thdidx] > f_{new}$
14.              then $sh[thdidx] = f_{new}$
15.       else $\phi_a = f^{fw}(\bar{q}, i) + \phi^{bw}(q + q_i(i) - \bar{q}, i)$
16.            $\phi_b = \phi^{fw}(\bar{q}, i) + f^{bw}(q + q_i(i) - \bar{q}, i)$
17.            $\phi_{new} = 0$, $(\phi_a < \phi_b)?\phi_{new} = \phi_a : \phi_{new} = \phi_b$
18.            if $sh[thdidx] > \phi_{new}$
19.              then $sh[thdidx] = \phi_{new}$
20. syncthreads()
21. // Parallel reduction
22. for $s = Nthds/2$ to $0$,$s/ = 2$ do
23.     if $thdidx < s$
```

```
24.          then if $h_{sh}[thdidx + s] < h_{sh}[thdidx]$
25.                 then $h_{sh}[thdidx] = h_{sh}[thdidx + s]$
26.               syncthreads()
27. syncthreads()
28. // Data Update
29. if $thdidx == 0$
30.    then $\psi(q, i) = sh[0]$
```

## 5.5 Computational Results

In this section we report the experimental results of our algorithms. Each table reports the execution time for a single run of the methods. All the times for the GPU are calculated taking in account the load and store time for the data between the CPU and the GPU. We choose to take into account these times because the relaxations are repeatedly called inside a CG algorithm and the load and store times have a significant impact on the performances.

The test machine is a workstation equipped with an Intel i7 4820K @3.9 GHz with 32 Gigabytes of RAM and a Nvidia GTX TITAN with 2688 Cuda Cores @837 MHz with 6 Gigabytes of GDDR5 RAM provided by SINTEF [91]. The data-sets are the ones of the VRPLIB [92] and the bigger instances are the ones provided by [93]. We report the speed-up factor between the serial and the parallel versions of the methods. The speed up factor is the ratio between the serial and the parallel algorithms: $SpeedUp = Time_{Serial}/Time_{Parallel}$.

Table 5.1: Computational results for the symmetric q-path and through-q-route relaxations.

| Instances | | | CPU Algorithm Times | | | GPU Algorithm Times | | | Speed-Up | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Nodes | Capacity | q-path | t-q-route | q + through-q | q-path | t-q-route | q + through-q | q-paths | t-q-route | q + through-q |
| V560-1200 | 560 | 1200 | 1.045 | 2.325 | 3.370 | 0.140 | 0.210 | 0.350 | 7.455 X | 11.055 X | 9.615 X |
| V600-900 | 600 | 900 | 0.874 | 1.311 | 2.185 | 0.101 | 0.128 | 0.228 | 8.692 X | 10.257 X | 9.568 X |
| V640-1400 | 640 | 1400 | 1.576 | 4.695 | 6.271 | 0.177 | 0.284 | 0.461 | 8.897 X | 16.527 X | 13.596 X |
| V720-1500 | 720 | 1500 | 2.215 | 6.677 | 8.892 | 0.251 | 0.363 | 0.614 | 8.831 X | 18.369 X | 14.475 X |
| V760-900 | 760 | 900 | 1.857 | 2.340 | 4.197 | 0.162 | 0.149 | 0.311 | 11.483 X | 15.717 X | 13.513 X |
| V800-1700 | 800 | 1700 | 4.805 | 11.497 | 16.302 | 0.362 | 0.511 | 0.873 | 5.126 X | 22.489 X | 18.663 X |
| V840-900 | 840 | 900 | 3.027 | 2.932 | 5.959 | 0.209 | 0.165 | 0.373 | 14.514 X | 17.807 X | 15.967 X |
| V880-1800 | 880 | 1800 | 7.004 | 15.241 | 22.245 | 0.486 | 0.627 | 1.113 | 14.424 X | 24.289 X | 19.985 X |
| V960-2000 | 960 | 2000 | 10.046 | 23.353 | 33.399 | 0.628 | 0.837 | 1.465 | 15.995 X | 27.910 X | 22.802 X |
| V1040-2100 | 1040 | 2100 | 13.166 | 29.640 | 42.806 | 0.869 | 0.994 | 1.862 | 15.158 X | 29.831 X | 22.987 X |
| V1120-2300 | 1120 | 2300 | 16.832 | 40.154 | 56.986 | 1.061 | 1.276 | 2.336 | 15.869 X | 31.479 X | 24.392 X |
| V1200-2500 | 1200 | 2500 | 21.092 | 52.026 | 73.118 | 1.328 | 1.607 | 2.935 | 15.878 X | 32.385 X | 24.914 X |

Table 5.2: Computational results for the asymmetric q-path and through-q-route relaxations.

| Instances | | | CPU Algorithm Times | | | GPU Algorithm Times | | | SpeedUp | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Nodes | Capacity | q-path | t-q-route | q + through-q | q-path | t-q-route | q + through-q | q-paths | t-q-route | q + through-q |
| A034-02f | 34 | 1000 | 0.008 | 0.031 | 0.039 | 0.024 | 0.007 | 0.031 | 0.334 X | 4.722 X | 1.278 X |
| A036-03f | 36 | 1000 | 0.015 | 0.031 | 0.031 | 0.024 | 0.007 | 0.031 | 0.619 X | 4.557 X | 0.999 X |
| A039-03f | 39 | 1000 | 0.015 | 0.031 | 0.046 | 0.026 | 0.007 | 0.033 | 0.579 X | 4.213 X | 1.382 X |
| A045-03f | 45 | 1000 | 0.015 | 0.047 | 0.062 | 0.026 | 0.009 | 0.034 | 0.578 X | 5.505 X | 1.799 X |
| A048-03f | 48 | 1000 | 0.016 | 0.047 | 0.063 | 0.025 | 0.009 | 0.035 | 0.634 X | 4.986 X | 1.817 X |
| A056-03f | 56 | 1000 | 0.015 | 0.063 | 0.078 | 0.026 | 0.011 | 0.037 | 0.584 X | 5.777 X | 2.131 X |
| A065-03f | 65 | 1000 | 0.016 | 0.093 | 0.109 | 0.026 | 0.013 | 0.039 | 0.606 X | 7.388 X | 2.796 X |
| A071-03f | 71 | 1000 | 0.031 | 0.078 | 0.109 | 0.026 | 0.014 | 0.040 | 1.184 X | 5.642 X | 2.724 X |
| Balman859-1000 | 859 | 1000 | 6.100 | 3.026 | 9.126 | 0.393 | 0.122 | 0.515 | 15.514 X | 24.781 X | 17.710 X |
| Balman859-2000 | 859 | 2000 | 13.385 | 17.113 | 30.498 | 0.791 | 0.502 | 1.292 | 16.929 X | 34.102 X | 23.597 X |

Table 5.3: Computational results for the symmetric ng-path relaxation.

| Instances | | | CPU Algorithm Times | | | | | GPU Algorithm Times | | | | | SpeedUp | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Nodes | Capacity | NG:8 | NG:10 | NG:12 | NG:13 | NG:14 | NG:8 | NG:10 | NG:12 | NG:13 | NG:14 | NG:8 | NG:10 | NG:12 | NG:13 | NG:14 |
| A-n62-k8 | 62 | 100 | 0.047 | 0.093 | 0.281 | 0.421 | 0.717 | 0.004 | 0.007 | 0.021 | 0.047 | 0.080 | 12.796 X | 12.510 X | 13.646 X | 8.875 X | 8.947 X |
| A-n63-k10 | 63 | 100 | 0.047 | 0.094 | 0.234 | 0.359 | 0.546 | 0.003 | 0.007 | 0.020 | 0.046 | 0.070 | 14.030 X | 12.974 X | 11.958 X | 7.775 X | 7.813 X |
| A-n64-k9 | 64 | 100 | 0.047 | 0.093 | 0.249 | 0.375 | 0.515 | 0.003 | 0.007 | 0.021 | 0.034 | 0.066 | 14.766 X | 13.358 X | 11.906 X | 10.969 X | 7.830 X |
| A-n80-k10 | 80 | 100 | 0.078 | 0.187 | 0.484 | 0.733 | 1.061 | 0.005 | 0.012 | 0.036 | 0.067 | 0.113 | 14.888 X | 15.004 X | 13.425 X | 10.927 X | 9.356 X |
| B-n50-k8 | 50 | 100 | 0.031 | 0.094 | 0.296 | 0.484 | 0.827 | 0.003 | 0.007 | 0.024 | 0.042 | 0.095 | 10.431 X | 12.835 X | 12.530 X | 11.433 X | 8.715 X |
| B-n68-k9 | 68 | 100 | 0.094 | 0.234 | 0.639 | 0.842 | 1.373 | 0.006 | 0.021 | 0.068 | 0.088 | 0.167 | 14.925 X | 10.928 X | 9.441 X | 9.530 X | 8.220 X |
| B-n78-k10 | 78 | 100 | 0.110 | 0.343 | 0.998 | 1.482 | 2.511 | 0.010 | 0.032 | 0.109 | 0.199 | 0.290 | 11.542 X | 10.754 X | 9.192 X | 7.442 X | 8.659 X |
| E-n51-k5 | 51 | 160 | 0.063 | 0.125 | 0.359 | 0.593 | 0.920 | 0.004 | 0.008 | 0.025 | 0.062 | 0.084 | 16.475 X | 15.543 X | 14.089 X | 9.607 X | 10.916 X |
| E-n76-k7 | 76 | 220 | 0.156 | 0.421 | 1.279 | 2.043 | 3.260 | 0.010 | 0.026 | 0.081 | 0.194 | 0.305 | 15.388 X | 16.476 X | 15.775 X | 10.533 X | 10.680 X |
| E-n76-k8 | 76 | 180 | 0.125 | 0.296 | 0.921 | 1.420 | 2.215 | 0.008 | 0.019 | 0.071 | 0.139 | 0.222 | 16.346 X | 15.811 X | 12.894 X | 10.228 X | 9.975 X |
| E-n76-k10 | 76 | 140 | 0.078 | 0.187 | 0.530 | 0.811 | 1.216 | 0.005 | 0.012 | 0.036 | 0.067 | 0.124 | 14.684 X | 15.352 X | 14.536 X | 12.174 X | 9.773 X |
| E-n76-k14 | 76 | 100 | 0.031 | 0.078 | 0.203 | 0.297 | 0.390 | 0.003 | 0.006 | 0.018 | 0.044 | 0.065 | 10.375 X | 12.597 X | 11.361 X | 6.688 X | 5.998 X |
| E-n101-k8 | 101 | 200 | 0.272 | 0.795 | 2.371 | 3.619 | 5.523 | 0.017 | 0.046 | 0.182 | 0.306 | 0.528 | 15.826 X | 17.271 X | 13.042 X | 11.824 X | 10.453 X |
| E-n101-k14 | 101 | 112 | 0.110 | 0.280 | 0.765 | 1.108 | 1.623 | 0.007 | 0.018 | 0.061 | 0.098 | 0.170 | 15.621 X | 15.862 X | 12.535 X | 11.344 X | 9.520 X |
| F-n135-k7 | 135 | 2210 | 9.516 | 30.342 | 91.853 | Out | Out | 0.480 | 1.634 | 5.694 | Out | Out | 19.811 X | 18.565 X | 16.131 X | Out | Out |
| M-n121-k7 | 121 | 200 | 0.671 | 2.169 | 9.345 | 19.407 | 38.876 | 0.045 | 0.246 | 1.203 | 2.710 | 5.731 | 14.871 X | 8.817 X | 7.767 X | 7.160 X | 6.783 X |
| M-n151-k12 | 151 | 200 | 0.577 | 1.591 | 4.181 | 6.692 | 10.358 | 0.034 | 0.099 | 0.323 | 0.556 | 0.978 | 16.731 X | 16.083 X | 12.952 X | 12.038 X | 10.587 X |
| M-n200-k16 | 200 | 200 | 0.983 | 2.901 | 7.301 | 10.905 | 15.928 | 0.064 | 0.198 | 0.611 | 0.956 | 1.524 | 15.292 X | 14.629 X | 11.946 X | 11.408 X | 10.450 X |
| M-n200-k17 | 200 | 200 | 0.998 | 2.902 | 7.332 | 10.888 | 15.943 | 0.064 | 0.197 | 0.617 | 0.966 | 1.527 | 15.501 X | 14.759 X | 11.887 X | 11.272 X | 10.443 X |
| P-n50-k8 | 50 | 120 | 0.031 | 0.047 | 0.109 | 0.171 | 0.250 | 0.002 | 0.005 | 0.012 | 0.024 | 0.044 | 14.299 X | 9.918 X | 8.804 X | 7.001 X | 5.685 X |
| P-n70-k10 | 70 | 135 | 0.047 | 0.125 | 0.359 | 0.530 | 0.749 | 0.004 | 0.009 | 0.035 | 0.048 | 0.089 | 12.401 X | 13.832 X | 10.362 X | 10.954 X | 8.439 X |

Table 5.4: Computational results for the asymmetric ng-path relaxation.

| Instances | | | CPU Algorithm Times | | | | | GPU Algorithm Times | | | | | SpeedUp | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | Nodes | Capacity | NG:8 | NG:10 | NG:12 | NG:13 | NG:14 | NG:8 | NG:10 | NG:12 | NG:13 | NG:14 | NG:8 | NG:10 | NG:12 | NG:13 | NG:14 |
| A034-02f | 34 | 1000 | 0.422 | 1.045 | 3.151 | 5.055 | 8.096 | 0.022 | 0.035 | 0.086 | 0.128 | 0.199 | 19.055 X | 29.825 X | 36.821 X | 39.555 X | 40.595 X |
| A036-03f | 36 | 1000 | 0.406 | 1.014 | 3.230 | 4.711 | 7.629 | 0.023 | 0.037 | 0.086 | 0.127 | 0.194 | 17.849 X | 27.397 X | 37.417 X | 37.189 X | 39.292 X |
| A039-03f | 39 | 1000 | 0.406 | 1.185 | 3.417 | 5.289 | 8.205 | 0.024 | 0.038 | 0.091 | 0.146 | 0.221 | 17.256 X | 31.399 X | 37.441 X | 36.216 X | 37.103 X |
| A045-03f | 45 | 1000 | 0.546 | 1.404 | 4.259 | 6.942 | 12.137 | 0.027 | 0.053 | 0.138 | 0.238 | 0.443 | 19.884 X | 26.447 X | 30.809 X | 29.206 X | 27.377 X |
| A048-03f | 48 | 1000 | 0.671 | 1.809 | 5.835 | 9.625 | 18.205 | 0.030 | 0.079 | 0.239 | 0.427 | 0.956 | 22.587 X | 22.900 X | 24.374 X | 22.565 X | 19.041 X |
| A056-03f | 56 | 1000 | 0.905 | 2.730 | 7.784 | 14.367 | 21.918 | 0.041 | 0.114 | 0.331 | 0.687 | 1.102 | 22.124 X | 23.856 X | 23.514 X | 20.910 X | 19.898 X |
| A065-03f | 65 | 1000 | 1.154 | 3.213 | 9.423 | 15.398 | 25.365 | 0.053 | 0.122 | 0.398 | 0.720 | 1.298 | 21.691 X | 26.385 X | 23.661 X | 21.391 X | 19.547 X |
| A071-03f | 71 | 1000 | 1.326 | 3.869 | 11.684 | 19.625 | 30.639 | 0.057 | 0.160 | 0.572 | 1.109 | 1.813 | 23.425 X | 24.127 X | 20.428 X | 17.703 X | 16.904 X |

In table 5.1 we report the speed-up factor relative to the q-paths and through-q-routes algorithms. We used the biggest instances for the CVRP in literature for two reasons:

1. the execution times for these algorithms on the VRPLIB is negligible because of the small instances dimensions ,
2. we want to highlight the method's scalability on very difficult instances.

In fact, for bigger instances the global speed-up factor obtained is really consistent, 24 X for the V1200-2500 instance. We decided to report the unified speed-up for both the methods because to compute the through-q-routes we need the results of q-paths. However, we can see that the through-q-routes is the algorithm achieving the best performances.

For the asymmetric instances, table 5.2, as before, the best results are obtained for big instances.

In table 5.3, we report the results for the NG relaxation for the symmetric CVRP. In this case we evaluate the scalability of the method among different dimensions for the $\Delta$ parameter (8,10,12,13,14), using the VRPLIB instances because of the data-structures dimensions reached during the computations. It's easy to notice that the performances of the method degrade with bigger neighborhood sets, because of the increasing of atomic operations among the labels and the bigger data transfer time from and to the GPU, but in most cases, remaining above the 10 X factor. In the asymmetric 5.4 case we can appreciate the maximum speed-up obtained, 40 X. In this case we can show all the computation capabilities of the device, exploiting all the parallelism levels available as described before.

## 5.6 Considerations and Future Work

In this chapter we highlighted the great advantages that a parallel algorithm can bring to this pricing strategies for the VRP. In fact, inside a CG method, the pricing problem is the most computationally expensive routine used to generate the columns to insert inside the master. The use of a GPU seems to be a very good alternative in terms of execution time, portability and affordability (the device used is an high end gaming GPU). Seems notable, moreover, the great performances obtained for the biggest instances, the hardest to solve and the ones taking relevant elaboration time. The future work planned is to explore the implementation of these methods with OpenCL, with the aim to make the methods portable on the most common parallel processors of different vendors, and, obviously, exploit these relaxations, or some their variants, in order to design most powerful and effective algorithms for solving instances from different classes of VRPs.

# Chapter 6

# Single Source Shortest Path Problem

## 6.1 Introduction

The ever more complex systems and eco-systems represented by urban centers and industrialized countries are growing fast and, nowadays, to exploit optimally the infrastructures composing these systems is more and more difficult. Passing through the public transport to the goods transportation and delivery, the problems related are increasingly strategic and dealing with these is the focus of many studies and researches from academia and private companies.

To provide high quality decision support tools is mandatory to preserve the resources from the environmental point of view and enhance life's quality in densely populated areas. By now, for instance, a great urban center has different kinds, or modes, of transportation, covering the city area, allowing people to move easily from a location to another. Also in highly industrialized countries, the transportation is characterized by multiple networks (railways, motorways, air-flights . . . ), that permit a fast displacement of people and goods. The layered networks composing these transportation infrastructures have different properties and characteristics, making the related mathematical models more complex and the consistent and effective resolution of optimization problem for these models a non-trivial challenge.

Routing is a widely explored research topic and has its origin in early fifties with the well known Dijkstra's algorithm [94], for finding the shortest path inside an oriented graph or the Bellman-Ford algorithm [95], computationally more expensive but more effective for other purposes. More generally, for routing we mean finding the 'best' path relative to one or more aspects of the journey: mileage, cost, fuel consumption, time, number of transportation modes used and others. In fact, we shift the focus from finding the shortest path in a geographical network, to optimize a route among different layers with respect to other factors. One aspect, mainly related to the people and goods transportation, is the arrival time to a certain destination. In a public transport or goods delivery scenario, earlier is the arrival time, better is the QoS.

Over the years many enhancement to the basic algorithms cited above has been proposed, achieving good results in a large spectrum of routing problems, but in cases where the query is strictly related to a temporal dimension, algorithm using bi-directional search, contraction hierarchies or an heuristic to compute a completion bound to the solution like A* [96] are not applicable because of the changes of the networks values (mainly the edges costs) or the topology in function of time. In this case we can only compute the routing problem's solution using an 'augmented' version of the Shortest Path algorithm that, up to certain dimensions, is computationally expensive.

In this chapter we propose two parallel algorithms implemented both on CPU (multi-core, shared memory platform) and GPU to solve the Earliest Arrival Problem in a Time-Dependent Multi-Modal Network reporting the performance obtained compared to the serial version.

## 6.2 Problem Definition

In this section we will define all problem's peculiarities starting from the definition of Earliest Arrival Problem, showing that can be solved using an augmented version of the Shortest Path algorithm. We will also define a Multi-Modal Network, the algorithm to manage the routing in this type of graph and, finally, we will add to the model the time dimension, describing what changes will it add to the model.

**Definition 6.1.** (Earliest Arrival Problem). Given a time-independent or time-dependent network, source and target points $s$ and $t$ in the network, we ask for a *route* in the network with the following properties:

1. The route must start at s,
2. the route ends at t,
3. the length (travel time) of all other routes satisfying the properties (1)–(2) must be bigger or at least equal.

In other words, from all the possible routes in the network from $s$ to $t$, we seek the route with the minimum cost for arriving to $t$. As mentioned before, the route's cost can be any aspect from the fuel consumption to the travel time or a mix of more of these criteria. In this paper we will cover only the optimization relative to one criteria.

Analyzing the definition 6.1, we can easily notice that, substituting to the optimization criteria any function of weight for the edges of the network, this problem becomes a Single Source Shortest Path Problem. We will define the SSSP Problem first, then we will propose the Multi-Modal and Time-Dependent version.

### 6.2.1 Single Source Shortest Path Problem

Shortest Path is a deeply investigated problem in the Combinatorial Optimization. This problem and its variations are subject of research from about five decades and can be found, often like subproblem, in a wide plethora of applications. In our case, shortest paths are the basis for the problem we are discussing.

**Definition 6.2.** (Single Source Shortest Path Problem). Given a weighted, directed graph $G = (V, E)$, a source node $s \in V$, a target node $t \in V$ and a weight function $w(e)$ for the edge $e = (v_a, v_b)$, $v_a \in V$ and $v_b \in V$ , we ask for a path $P = \{v_1, ..., v_k\}$, with the following properties:

1. The path begins at $s$, thus $v_1 = s$,
2. the path ends at $t$, thus $v_k = t$,

3. $P$ is minimal.

We define $Len(P) = \sum_{i=1}^{k-1} w(v_i, vi + 1)$, the length of the path $P$.

The SSSPP has some common declinations, depending on the number of sources and target considered:

- *Many-To-Many-Shortest Path Problem.* This is a generalization of the Shortest Path Problem. Instead of one node $s$ and $t$ we are given a set of source nodes $S \subseteq V$ and a set of target nodes $T \subseteq V$. We now ask for a shortest path $P_{s,v}$ for each pair $(s, t) \in S \times T$. In multi-modal routing the Earliest Arrival Problem will actually transform to this version of the problem.
- *One-To-All-Shortest Path Problem.* This is a special case of the Many-To-Many-Shortest Path Problem where $S$ is a singleton set consisting of one source node s and $T = V$ is the set of all nodes. Hence, we are asking for shortest paths $P_v$ to every node $v \in V$. Because the edge set of all resulting paths $T = \cup_{P_{s,v}}, v = 1, \ldots, |V|$ forms a tree, we might also say that we compute a *shortest path tree.*
- *All-Pairs-Shortest Path Problem.* This is a version of the Many-To-Many-Shortest Path Problem where both S and T are the complete node set V of the graph. Having the All-Pairs-Shortest Path Problem solved automatically includes solutions for all instances of the Shortest Path Problem in the graph. For this problem in most cases is used the Floyd-Warshall dynamic programming algorithm [97].

All these problems can be solved using the same algorithm, executing it multiple time often. In this paragraph we didn't take into account the time dimension, we will extend the model after the introduction of the Multi-Modal Networks.

We can consider the SSSPP a special case of a Multi-Modal network with only one mode. In this case the solution algorithm for the routing problem without time-dependencies is equivalent to the Dijkstra algorithm.

**Algorithm** *Unimodal Routing*$(s, t, \mathbf{V}, \mathbf{E}, w())$

```
1.  vals // Tentative Distances Values
2.  preds // Predecessors vector
3.  Queue Q = null // Priority Queue
4.  // Data Structures Initialization
5.  Q.Insert(s)
6.  for $i = 1$ to $|V|$ do
7.      if $i == s$
8.          then $vals[i] = 0$, $preds[i] = 0$
9.          else $vals[i] = \infty$, $preds[i] = \infty$
10. // Algorithm
11. while Q$\neq$ null do
12.     $n$ = Q.first()
13.     if $n == t$
14.         then break
15.         else $val_n = vals[n]$
16.     for each $succ \in \Gamma_n$ do
17.         $cost_e = w((n, succ))$
18.         if $vals[succ] > val_n + cost_e$
19.             then $vals[succ] = vals_n + cost_e$
20.                  $preds[succ] = n$
21.                  Q.Insert($succ$)
```

The proposed algorithm is a straightforward implementation of the Dijkstra one. In fact in line 3 we initialize a priority queue implemented with a heap, ordered by the tentative values, in line 5 we insert in the heap the start node $s$. In lines 6-9 we initialize the tentative values for each node and the predecessors to retrieve the shortest path.

In lines 11-21 we have the core algorithm inside a *do-while* cycle that ends once the queue is empty. In line 12 we extract the root from the heap, in lines 13-15 we check if we have reached the target $t$, otherwise we extract the cost of the label relative to the actual node $n$. The *foreach* statement, in lies 16-21, evaluate the new tentative values for each successor *succ* of $n$ and update them if the successor's label is greater, keeping trace of the predecessor, $n$, and inserting the successor node *succ* in the queue, line 21, that will reorder itself maintaining the heap properties.

### 6.2.2 Multi-Modal Networks

In this section, we define a Multi-Modal Network and the other basis on which relies the routing for this type of networks. First, we give de definition of Multi-Modal or Multi-Layer Network:

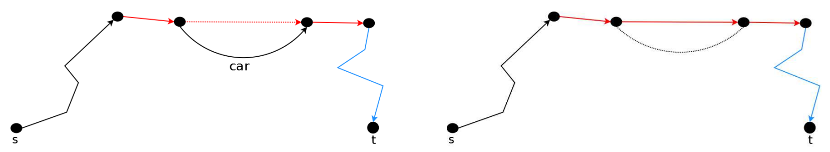
**Definition 6.3.** (Multi-Modal Network). Given a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{M})$ where:

1. $\boldsymbol{M} = \{M^1, M^2, \ldots, M^n\}$ is the set of modes, or layers, composing the network, where $M^i = (V^i, E^i)$ is the graph representing the $i$ mode or layer, $V^i$ is the set of vertices of $i$, $E^i$ is the set of edges of $i$ and $n = |\boldsymbol{M}|$,
2. $\boldsymbol{V} = \bigcup_{i=1}^{n} V^i$, with $V^i \cap V^j = \{0\}$, $i \neq j$,
3. $v^i \in V^i$ is a vertex for the mode $i$,
4. $e^i \in E^i$ is an edge connecting two nodes of the same mode $i$, $e = (v_h^i, v_k^i)$,
5. $e^t \in E^t$, is an edge connecting two nodes of different modes $i$ and $j$, $e^t = (v_h^i, v_k^j)$, with $i \neq j$ (*transition edge*),
6. $\boldsymbol{E} = \bigcup_{i=1}^{n} E^i \cup E^t$ is the set of edges of $\boldsymbol{G}$ where $E^i \cap E^j = \{0\}$.

The $\boldsymbol{G}$ graph is a *multi-modal* or *multi-layer* graph composed by $n$ modes or layers. For each $M^i$ graph we can define a cost function $w_i(e^i)$ for the mode's edges. For the transition edges, the $w_t(e_t)$ is equal to 0.

For instance, the network of public transport in an urban area (roads, bus lines, tram lines, trains, boats, etc. . . ), can be modelled with a multi-modal network. Another example can be the different types of transportation for a delivery service (postal service for instance), that has different types of networks to deliver the goods (ship, aeroplane, etc. . . ).

The routing problem for this type of networks can be seen as Shortest Path Problem among the various, connected, modes of the network, using the relative $w_i()$ weight functions to evaluate each time the cost of an edge. In this case we will found a route, passing through the various modes of the network from the source $s$ to the target $t$. In this scenario, we are allowed to choose also path exploiting various type of transportation: car, bus, trams, boats, together in the same path.



(A) Not Convenient Route (B) Convenient Route

FIGURE 6.1: Types of Routes.

Obviously, this kind of solution is not desirable, in fact, a 'well formed' route is a route passing only through certain types of modes depending on the *state* of the traveler: For instance, assume that we are a tourist in Rome and we want to visit some of the most beautiful places in the city. We are by foot and we want a route that uses the public transport and the road network, allowing us to reach the places we have chosen. The algorithm, as it is, can give us undesirable results finding that the most convenient path between the Colosseum and the Vatican Museums is taking the metro until a certain point, then take the car, take another bus, then the car again to reach the Museums.

It's straightforward that a path like this is not desirable, we'd like a path using only the public transportation to reach our destination (6.1). Another side-effect of using this algorithm on these types of networks is that while we are traveling on a train, the railway intersects a road, without a stop for the train, that can bring us to destination faster, the algorithm can suggest us to take that road. In general, we are forced to evaluate paths that are compatible with the state (foot, car, bicycle, etc. . . ) of the traveler.

### 6.2.3 Label Constrained Shortest Path Problem

Barrett [98] proposed an algorithm to solve the problem of not desirable routes associating to the graph an Automaton that, treating the modes and the states as part of a DFA, Deterministic Finite Automata, regulates the transition among the modes of the network. We will give a brief definition for a language and an automaton, then we will describe the algorithm based on these and its implications.

**Definition 6.4.** (Regular Languages). Let $\Sigma$ be an alphabet. Then a language $L$ over $\Sigma$ is regular if and only if it confirms the following construction rules:

1. The empty language $\{0\}$ is regular,

2. for each $\sigma \in \Sigma$ the singleton language $\{\sigma\}$ is regular,
3. if $L_1$ and $L_2$ are regular languages, then $L_1 \cup L_2$, $L_1 \cdot L_2$ and $L_1^*$ are also regular languages.

To describe a regular language $L$ we can use formalism like regular expression or automata. In our case we will give a brief definition for the DFA describing the language $L$.

**Definition 6.5.** (DFA, Deterministic Finite Automaton). A Deterministic Finite Automaton describing the language $L$ is given by a 5-tuple $\boldsymbol{A} = (Q, \Sigma, \delta, q_0, F)$ where:

1. $Q$ is the set of states composing the automaton,
2. $\Sigma$ is the alphabet, a finite set of symbols,
3. $\delta$ is the *transition function* $\delta : Q \times \Sigma \to \mathcal{P}(S)$,
4. $q_0$ is the initial state.
5. $F$ is the set of final states.

We say that a word $w$ is *accepted* by $\boldsymbol{A}$ if and only if exists a path from $q_0$ to $q_f \in F$ regulated by $\delta$.

By Kleene's Theorem [99] each regular language $L$ can be described by a finite automaton in the sense that every word $w \in \Sigma^*$ is accepted by this automaton. The language $L$ and the automaton $\boldsymbol{A}$ can be interchanged.

Given the definition of language and automata, we can define the Label Constrained Shortest Path Problem:

**Definition 6.6.** (Label Constrained Shortest Path Problem). Given an alphabet $\Sigma$, a language $L \subset \Sigma^*$, a weighted, directed graph $G = (V, E)$ with $\Sigma$-labeled edges and source and target nodes $s, t \in \Sigma$, we ask for a shortest path $P$ from $s$ to $t$, where the sequence of labels along the edges of the path creates a word of $L$. Thus given $P = [v_1, \ldots, v_k]$ it has to hold that:

$$label((v_1, v_2))label((v_2, v_3)(\ldots label((v_{k-1}, v_k)) \in L. \tag{6.1}$$

This definition needs no restrictions on the language $L$ but, for our purposes, a regular language is sufficient to model the transition among the modes of the network. In [98] the following theorem is proven:

**Theorem 6.7.** *the Label Constrained Shortest Path Problem restricted to Regular Languages, RegL-CSPP, can be solved in deterministic polynomial time.*

An algorithm for solving this problem operates on a *product graph* between the automaton $A$, describing the allowed transitions among the modes and the graph $\boldsymbol{G}$.

**Definition 6.8.** (Product Network). Given a $\Sigma$-labeled graph $G = (V, E)$ and a non-deterministic finite automaton $A = (Q, S, \delta, q_0, F)$, the product network $G^{\times} = (V^{\times}, E^{\times})$ is defined as follows:

1. The node set consists of product-nodes $(v, q) \in V^{\times}$ where $v \in V$ and $q \in Q$.
2. An edge $e^{\times} = (v_1, q_1), (v_2, q_2)$ between two product-nodes is included in $E^{\times}$ if and only if $e = (v_1, v_2) \in E$ and there is a label $\sigma \in \Sigma$ for which exists a transition $q_2 \in \delta(q_1, \sigma)$ in the automaton. The weight of $e^{\times}$ is set to the weight of $e$ and $label(e^{\times})$ is set to $\sigma$.

The resulting graph is uni-modal.

In [98] is proven that this assumption holds:

**Theorem 6.9.** *The RegL-CSPP for a $\Sigma$-labeled graph $G = (V, E)$ from source $s \in V$ to target $t \in V$ and a regular language $L \subseteq \Sigma^*$ can be reduced to the Shortest Path Problem as follows:*

1. *Construct a finite automaton $A = (Q, \Sigma, \delta, S, F)$describing $L$, where $S$ is the set of starting states,*
2. *construct the product network $G^{\times} = G \times A$*
3. *solve the Many to Many Shortest Path Problem for $G^{\times}$ using:*

$$\mathcal{S} = \bigcup_{q_s \in S} (s, q_s), \mathcal{T} = \bigcup_{q_f \in F} (t, q_f) \tag{6.2}$$

*where $\mathcal{S}$ and $\mathcal{T}$ are, respectively, the set of the sources and the set of the targets inside the product graph,* *4. from all resulting paths pick the one having minimal length.*

Let $P = [(v_1, q_1), \dots, (v_k, q_k)]$be the shortest path obtained by the algorithm induced from Theorem 2. Then the length of the path in $G$ is the same as the length $Len(P)$ in $G^\times$. The actual path in $G$ can be obtained by omitting the 'automaton part' of the product-nodes, thus, yielding $[v_1, \dots, v_k]$. On the other hand, the word conforming to $L$ along the path can be obtained by concatenating the edge labels:

$$word(P) = label((v_1, q_1), (v_1, q_2)) \dots label((v_{k-1}, q_{k-1}), (v_k, q_k)) \tag{6.3}$$

The creation of the product graph can be computed in time $\mathcal{O}(|G| \cdot |A|)$ which is also polynomial. Hence, the algorithm induced by Theorem 2 runs in polynomial time. The memory complexity and the space required to store the product graph $G^\times$ is also in $\mathcal{O}(|G| \cdot |A|)$ which is extremely expensive for large instances.

The memory complexity of the product graph can be easily avoided using the transition graph relative to the automaton $A$. We can compute implicitly the shortest paths for the Many to Many SPP using an constrained version of the Dijkstra algorithm regulated by the transition graph of the automaton.

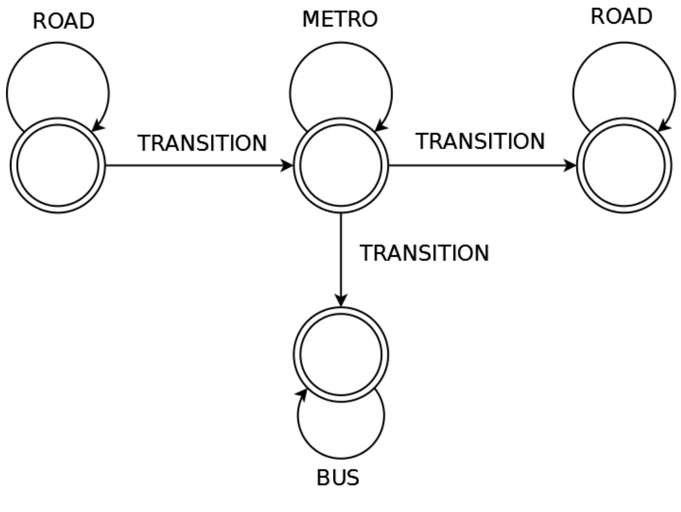


Figure 6.2: Automaton managing four states.

In this case, the memory complexity of $G^\times$ becomes $\mathcal{O}(|G|+|A|)$, reducing consistently the memory space required. For instance, we can assume that a traveller, by foot, can use the public transport and walk through the streets by foot, then, the the relative automaton will allow the moving through the various modes, like bus, trains or subways and the costs of the streets will be computed considering the average speed of a man walking.

A bike trip, instead, must be constrained by the streets or the transport networks. In fact is not allowed to ride a bicycle in a motorway or to bring it on a bus, instead is allowed to ride in urban centers or in secondary roads and to bring the bicycle on a train or a subway. All these aspects must be modeled by the automaton, that, during the evaluation of the successors states and the computation of the new tentative costs, allows the algorithm to expand or not a state. Here we propose an augmented version of the algorithm *Unimodal Routing*, taking into account a Multi-Modal Network and an automaton.

```
Algorithm RegL-CSPP$(s, t, \mathbf{V}, \mathbf{E}, w(), A)$
1.  Queue SQ=null // Generated States of $G^{\times}$ queue
2.  // Queue Initialization
3.  for each $(s, q_s) \in \mathcal{S}$ do
4.      SQ.Insert($(s, q_s)$,$((0,0),0)$ $0)$
5.  // Main cycle
6.  while SQ $\neq$ null do
7.      $((n, s), (p, q, val(p, q)), f(n, s))$ = SQ.First()
8.      if $(n, s) \in \mathcal{T}$
9.          then $reached++$
10.             if $reached == |\mathcal{T}|$
11.                 then break
12.     for each edge $e = (n, succ)$, $succ \in \Gamma_n$ do
13.         for each $q' \in \delta(s, label(e))$ do
14.             if $(succ, q') \notin SQ$
15.                 then SQ.Insert($(succ, q')$,$((n, s), f(n, s))$, $f(n, s) + w(e))$
16.                 else if $f((n, s)) + w(e) < f((succ, q'))$
17.                         then SQ.Update($(succ, q')$,$((n, s), f(n, s))$,$f((n, s))+$
                                 $w(e))$
```

In the algorithm *RegL-CSPP* we introduce the triple $((v, q), (p, r, f(p, r)), f(v, q))$ indicating a certain node $v$ in a state $q$ and its value, keeping trace of the predecessor triple of node $p$, state $r$ and value$f(p, r)$ also, to retrieve the path at the end of the computation.

In lines 3-4 we initialize the queue $SQ$ collecting all the $G^{\times}$ states produced during the computation, ordering following their value $f$. As in the algorithm *Unimodal Routing* we implemented it with an heap. More specifically, in line 4 we initialize all the states for the starting point $s \in \mathcal{S}$, setting the predecessors and the values to 0.

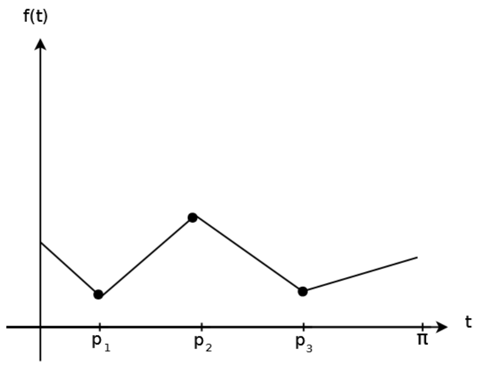
In lines 8-11 we check if all the targets states $t \in \mathcal{T}$ has been reached, in this case we stop the computation. In lines 12-18 we update the data for each successor of $n$. In line 13 we check if the transition of the edge $e$ is allowed by the transition function $\delta$ of A, if allowed, we check if the state has been already generated, if not we insert the new triple in $SQ$ (line 15), otherwise, we update the existing triple with the new value, if the new path enhance the solution (lines 17-18).

### 6.2.4 Time-Dependent Networks

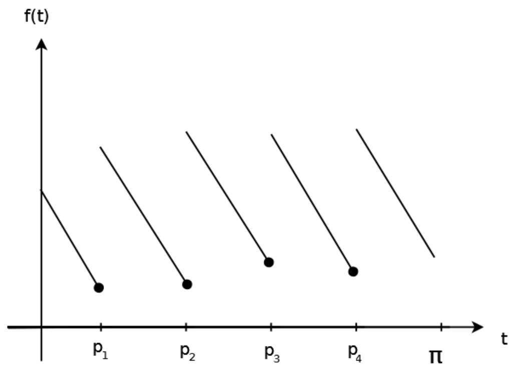
In time-dependent routing we do not longer have constant weights assigned to the edges. To accommodate for time-dependency, we replace the edge weights by arbitrary functions $f$ from some function space $\mathcal{F}$. The shortest s-t-path in a time-dependent model then depends on the departure time $\tau_s$ of the source node. This might result in shortest paths of different length for different departure times or, in general, even a completely different route. In the simplest case, we can describe these time-functions as periodic functions $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ with period $\Pi$, meaning that for each value $f(\tau) = f(\tau \mod \Pi)$.

To make the model more realistic, we need to express the $\mathcal{F}$ as a set of *piecewise linear functions*: A periodic function $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ is called *piecewise linear* if it consists of a finite number of segments of linear functions. Let $f$ be a piecewise linear function then $f$ can be described by a finite set $\boldsymbol{P}$ of interpolation points where each interpolation point $p_i \in \boldsymbol{P}$ consists of a departure time $\tau$ and an associated function value $f(\tau)$.

The value of $f$ for an arbitrary time $\tau$ is then computed by interpolation. This is done differently for time-dependent road networks and public transportation networks. Whereas in road networks we interpolate linearly between two subsequent interpolation points, the travel time function along a public transportation edge is interpreted as follows. First we have to wait for the next train or aeroplane to depart and then we have to add its mere travel time along that edge to that.



(A) Road Network Speed Profile



(B) Public Transport Speed Profile

FIGURE 6.3: Speed Profiles.

Hence, for some arbitrary time point t we use the nearest interpolation point $\tau$ in the future and interpolate by the formula:

$$f(\tau) = -\gamma \cdot (\tau_i - \tau) + f(\tau_i). \tag{6.4}$$

with $\gamma \in [-1, 0]$ as the fixed gradient for $f$. In figure 6.3 we show the two types of functions for the road network and the public transport network. For extending the Dijkstra algorithm to manage time-dependency we need to add as input the time $\tau_s$ of the day when the trip starts. There are two type of routing queries that we can perform on a time-dependent graph:

- *Time Queries*: The time-dependent version of Dijkstra's algorithm is almost identical to the time-independent version as illustrated in Algorithm *RegL-C-SPP* when computing time queries. The only changes to the algorithm that need to be made are the following two:
    1. we need to supply a departure time $\tau$ as additional input, as mentioned before,
    2. to evaluate the edge weights, we have to consider the current time at which we encounter the respective edge. Let $e = (v, w)$ be an edge of which the weight has to be evaluated, then the time at which we evaluate the function $f_e$ of the edge $e$ is the departure time $\tau$ plus the time along the path to $v$.
- *Profile Queries*: Using the previously described version of the time-dependent query algorithm yields only shortest paths for one particular departure time

$\tau$. While this seems to be a canonical generalization of the time-independent case, there is another type of query in time-dependent graphs, where we are not only interested in the shortest path at one time point, but at all times of day.

For example: in a railway network we state 8 o'clock as departure time $\tau$ for a query. Let's say there is a train departing at 8:00 to our destination takes 2 hours. But maybe there is another train departing at 9 o'clock that takes only 1 hour and 10 minutes. Taking the second train would be a suboptimal solution to the Earliest Arrival Problem (since the arrival time is 10 minutes later), but its sheer travel time is 50 minutes shorter. So, maybe it would be nice to present the user with the travel time for each possible departure time $\tau < \Pi$. In other words, the result of the query should be a piecewise linear function $f$ itself, where each interpolation point represents a shortest path for that particular time.

In our work we will consider only Time Queries. In fact, other types of queries in a time-dependent graph are extension or have as their foundation this type of query and proposing a parallel algorithm for this problem can be useful for all the others.

## 6.3 Serial Algorithm

Dean [100] proposed the extended version of the Dijkstra algorithm for time-dependent graphs. In this section we will propose the final version of the algorithm to compute the Shortest Path in a Multi-Modal Network with Time-Dependencies. This extension of the problem is particularly hard to solve, because we can't exploit some of the most famous and effective techniques for speeding-up the computation of Shortest Path or we need some pre-computation phases.

Algorithms like Contraction Hierarchies [101], bi-directional search, Arc-Flag [102, 103] or ALT (A* with Landmarks and Triangle Inequality) [104, 105] are not adaptable or need some long-time pre-computation for larger graphs. Pajor [106] proved that almost all of these methods are useless for the time-dependant case.

### 6.3.1 Time-Dependent Augmented Dijkstra Algorithm

**Algorithm** *T-D-RegL-CSPP*$(s, t, \mathbf{V}, \mathbf{E}, f_{time}, A, \tau_{start})$

```
1.  Queue SQ=null // Generated States of $G^{\times}$ queue
2.  Queue Initialization
3.  for each $(s, q_s) \in \mathcal{S}$ do
4.      SQ.Insert($(s, q_s)$,$((0, 0), \tau_0)$, $\tau_{start}$)
5.  // Main cycle
6.  while SQ $\neq$ null do
7.        $((n, h), (p, q, \tau_{p,q}), \tau_{n,h})$ = SQ.First()
8.        if $(n, h) \in \mathcal{T}$
9.           then $reached++$
10.               if $reached == |\mathcal{T}|$
11.                  then break
12.       for each $succ \in \Gamma(n)$ do
13.           $e = (n, succ)$
14.           for $q' \in \delta(h, label(e))$ do
15.               if $(succ, q') \notin SQ$
16.                  then SQ.Insert($(succ, q')$,$((n, h), \tau_{n,h})$, $\tau_{n,h} + f_{time}(e, \tau_{n,h})$)
17.                  else  if $\tau_{n,h} + f_{time}(e, \tau_{n,h}) < \tau_{succ,q'}$
18.                           then SQ.Update($(succ, q')$,$((n, h), \tau_{n,h})$,$\tau_{n,h} + f_{time}(e, \tau_{n,h})$)
```

We introduce the function $f_{time}()$ calculating the time for traveling the edge $e$. This function can be a collection of methods created to evaluate the traveling time, based on various factors: road type (primary, secondary, etc ...), the type of public transport (bus, subway, etc ...), the moment of the day (traffic profiles or other factors), the state of the automaton (depending if we are traveling by foot or on a bicycle). The algorithm is equivalent to the *RegL-CSPP* but here is computed the time for each query inside the multi-graph $\boldsymbol{G}$. Depending on the granularity of the problem, the time can be expressed in minutes or seconds.

## 6.4 GPU Algorithm

In this section, we propose an extension of the parallel Dijkstra algorithm proposed by [88] for computing the EAP in a Multi-Modal Time-Dependent Graph and its porting on a GPU using the CUDA [107] programming model. First, we present the basic algorithm, then we will introduce a new approach to compute the frontier set $F_i$ to enable the parallelism of the method.

### 6.4.1 GPU Dijkstra Algorithm

We can distinguish two parallelization alternatives that can be applied to Dijkstra's algorithm. The first one parallelizes the internal operations of the sequential Dijkstra algorithm, while the second one performs several Dijkstra algorithms through disjoint sub-graphs in parallel [108]. Our approach is focused in the first solution.

The key of the parallelization of a single sequential Dijkstra algorithm resides in the inherent parallelism of its loops. For each iteration, the outer loop selects a node to compute new distance labels. Inside this loop, the algorithm relaxes its outgoing edges in order to update the old distance labels, that is the inner loop. Parallelizing the outer loop implies to compute in each iteration $i$ a frontier set $F_i$ of nodes that can be settled in parallel without affecting the algorithm correctness. The main problem here is to identify this set of nodes $v$ which tentative distances $Val(v)$ from source $s$ must be the minimum shortest distance. Crauser et al.[109] and Crobak et al. [110] proposed two solutions addressing this problem. Parallelizing the inner loop implies to traverse simultaneously the outgoing edges of the frontier node. One of the algorithms presented in [111] is an example of this parallelization approach.

Following the approach in [109], we explain the method for identifying the frontier set $F_i$ and maximizing its cardinality. It's straightforward to highlight that bigger is the frontier set, higher is the level of parallelism of the method.

#### 6.4.1.1 Frontier Set

The method, in each iteration $i$, calculates the minimum tentative distance of the nodes belonging to the unsettled set, $U_i$. The node whose tentative distance is equal to this minimum value can be settled and becomes the frontier node. Its outgoing edges are traversed to relax the distances of the adjacent nodes. In order to parallelize the algorithm, it is needed to identify which nodes can be settled and used as frontier nodes at the same time. Martin et al. [112] inserts into the frontier set, $F_{i+1}$, all nodes with this minimum tentative distance with the aim to process them simultaneously. Crauser et al. [109] introduces a more aggressive enhancement, augmenting the frontier set with nodes with longer tentative distance. The algorithm computes in each iteration $i$, for each node of the unsettled set, $u \in U_i$, the sum of:

1. its tentative distance,
2. the cost of its outgoing edges.

Afterwards, it calculates the minimum of these computed values. Finally, those nodes whose tentative distance are lower or equal than this minimum value can be settled becoming the frontier set.

Introducing $\Delta_i$ as the threshold value computed in each iteration $i$ that holds that any unsettled node $u$ with $val(u) \leq \Delta_i$ can be safely settled. The bigger the $\Delta_i$ value, the more parallelism is exploited. However, depending on the particular graph being processed, the use of a very ambitious $\Delta_i$ may induce overheads that destroys any performance gain with respect to sequential execution.

The basic Dijkstra parallel method follows the idea proposed by Crauser [109] of incrementing each $\Delta_i$. For every node $v \in V$ , the minimum weight of its outgoing edges, that is, $\Delta_v = \min\{w(v,z) : (v,z) \in E\}$, is calculated in a pre-computation phase. For each iteration $i$ of the external loop, having all tentative distances of the nodes in the unsettled set, we define

$$\Delta_i = \min\{(val(u) + \Delta_v) : u \in U_i\} \tag{6.5}$$

More in general, we insert into the frontier set $F_{i+1}$ every node $v$ with $val(v) \leq \Delta_i$.

#### 6.4.1.2 GPU Implementation

Once defined the concept used for exposing the inherent parallelism inside the Dijkstra algorithm, we provide the complete pseudo-code for the method on the GPU.

```
Algorithm GPU Dijkstra($s, t, \mathbf{V}, \mathbf{E}, w(), \boldsymbol{\Delta_v}$)
1.  $\boldsymbol{vals}$, $\boldsymbol{preds}$, $\boldsymbol{U_i}$, $\boldsymbol{F_i}$
2.  $\Delta_i = \infty$
3.  // Data Structures Initialization
4.  for $i = 1$ to $|V|$ do
5.      if $i == s$
6.          then $vals[i] = 0$, $preds[i] = 0$
7.          else  $vals[i] = \infty$, $preds[i] = \infty$
8.  // GPU Algorithm
9.  Initialize<<<>>>($\boldsymbol{U_i}$, $\boldsymbol{F_i}$) // Frontier and Unsettled Nodes initialization
10. Initialize($\Delta_i$) //Threshold Initialization
11. while $\Delta_i \neq \infty$ do
12.     // Relax the frontier nodes
13.     RELAX-KERNEL<<<>>>($\boldsymbol{vals}$, $\boldsymbol{preds}$, $\boldsymbol{F_i}$, $\boldsymbol{U_i}$)
14.     // Update $\Delta_i$
15.     $\Delta_i$ = DELTA-UPDATE-KERNEL<<<>>>($\boldsymbol{vals}$, $\boldsymbol{U_i}$, $\boldsymbol{\Delta_v}$)
16.     // Update $F_{i+1}$
17.     FRONTIER-KERNEL<<<>>>($\Delta_i$, $\boldsymbol{F_i}$, $\boldsymbol{U_i}$)
```

```
Algorithm RELAX-KERNEL($\boldsymbol{vals}, \boldsymbol{preds}, \boldsymbol{F_i}, \boldsymbol{U_i}$)
1.  $n_{idx}$ = thread.id
2.  if $F_i[n_{idx}]$ == true
3.     then for each $u \in \Gamma_{n_{idx}}$ do
4.             if $U_i[u]$ == true
5.                then Start Atomic Operations
6.                     if $vals[u] > vals[n_{idx}] + w(n_{idx}, u)$
7.                        then $vals[u] = vals[n_{idx}] + w(n_{idx}, u)$
8.                             $preds[u] = n_{idx}$
9.                     End Atomic Operations
```

```
Algorithm FRONTIER-KERNEL($\Delta_i, \boldsymbol{F_i}, \boldsymbol{U_i}, \boldsymbol{\Delta_v}, \boldsymbol{vals}$)
1.  $n_{idx} = $ thread.id
2.  $F_i[n_{idx}] == false$
3.  if $U_i[n_{idx}] = true \wedge vals[n_{idx}] \leq \Delta_i$
4.      then $U_i[n_{idx}] = false$,$F_i[n_{idx}] = true$
```

The main procedure, *GPU Dijkstra*, uses three kernels to relax the nodes and create the frontier and unsettled nodes sets. In lines 3-11, we initialize the data structures used by the algorithm, lines 12-15 are the main loop, that stops once relaxed all the nodes in the graph or reached the $t$ node. The kernel function *Relax-Kernel* relaxes all the nodes inside $F_i$ (line 13), the *Delta-Update-Kernel* updates the $\Delta_i$ value using a parallel reduction. This procedure is a modified version of the *reduce3* procedure taken from the CUDA SDK that comes along with the CUDA package from Nvidia. *Frontier-Kernel* is the kernel function that creates the $F$ set at the next iteration.

The *RELAX-KERNEL* procedure updates the tentative distances of the nodes inside the $F$ set. Each thread elaborates a node $n_{idx}$, relaxing all its successors unsettled nodes $w \in \Gamma_{n_{idx}}$. The relaxation at line 6 is an atomic operation among the threads to avoid race conditions. We need to make this operation atomic because, at the same time, other threads can update the same memory location (the same unsettled node), generating inconsistent reads or writes.

The *FRONTIER-KERNEL* kernel generates the $U_{i+1}$ and $F_{i+1}$ sets. Each thread is assigned to a node and checks if the node tentative distance is less or equal to the $\Delta_i$ threshold and if the node is in the $U_i$ set. In this case the kernel inserts the node in the $F_i$ set.

The *DELTA-UPDATE-KERNEL* is a simple procedure implemented to avoid a data transfer between the CPU and the GPU. Basically, it's a parallel reduction among the nodes in the $U_i$ set, for finding the shortest tentative distance.

### 6.4.2 Dynamic Frontier Definition

We enhanced the *Frontier-Creation-Kernel* to address the Time-Dependency of our model. We need to evaluate for each iteration $i$ the $\Delta_v$ values. The main problem is that we can't evaluate a priori, the minimum cost among the outgoing edges from the $v$ vertex, because of the time dependency.

The only way to pre-calculate the value is to evaluate the minimum cost among the edges for each second of the day (the granularity of our problem), for each node, bringing to a great amount of memory used and a long computation time. We define the set $R_i$ as the set of the nodes $u \in U_i$ which tentative distances has been updated from the initial $\infty$ value, all possible members of the $F_{i+1}$ set at the $i+1$ iteration. For each node $r \in R_i$ we evaluate, at time $\tau$, the outgoing minimum cost edge.

$$\Delta_v = \min \{c = f_{time}(r, u, h, \tau_{r,u,h}) : r \in R_i, u \in \Gamma(r) \cap U_i, (r, u) \in \boldsymbol{E}, h \in A\} \tag{6.6}$$

Where $r$ is the node in the $R_i$ set, $u$ is a successor of $r$ in the $U_i$ set and $h$ a state of the automaton $A$. $f_{time}$ is the time function evaluating the cost of the edge from $r$ to $u$, in the state $h$ of the automaton at time $\tau$. The $\Delta_v$ is computed every iteration $i$, then, the values take part to the evaluation of the $\Delta_i$ threshold, as described in the *FRONTIER-KERNEL* to create the $F_i$ set.

### 6.4.3 GPU Time-Dependent Algorithm

The porting to the GPU environment implies some modification to the data structures used by the algorithm. We can't use dynamic data structures, performance killers for the GPU, and, to keep trace of the tentative distances and the predecessors, we will implement two bi-dimensional arrays *times* and *preds*, in which one dimension is the $|\boldsymbol{V}|$ cardinality and the other is the number $h$ of states in which the traveler can be (foot,car,etc ...). For each state of the traveler, we have an automaton $A_h$ that regulates the transitions among the modes of the network. For a more readable notation, we will call $A$ the macro-automaton composed by all the $A_t$ automata for each type of traveler.

```
Algorithm GPU T-D-RegL-CSPP($s, t, \mathbf{V}, \mathbf{E}, f_{time}(), \boldsymbol{\Delta_v}$)
1.    $\boldsymbol{vals}$, $\boldsymbol{preds}$, $\boldsymbol{U_i}$, $\boldsymbol{F_i}$, $\boldsymbol{R_i}$
```

```
2.  $\Delta_i = \infty$
3.  **for** $h = 1$ **to** $|TravelerStates|$ **do**
4.      **for** $i = 1$ **to** $|V|$ **do**
5.          **if** $i == s$
6.              **then** $vals[h][i] = 0$, $preds[h][i] = 0$
7.              **else** $vals[h][i] = \infty$, $preds[h][i] = \infty$
8.  // GPU Algorithm
9.  // Frontier, Unsettled Nodes and R initialization
10. Initialize<<<>>>($\boldsymbol{U_i}$, $\boldsymbol{F_i}$, $\boldsymbol{R_i}$)
11. Initialize($\Delta_i$) //Threshold Initialization
12. **while** $\Delta_i \neq \infty$ **do**
13.     // Relax the frontier nodes
14.     RELAX-KERNEL<<<>>>($\boldsymbol{vals}$, $\boldsymbol{preds}$, $\boldsymbol{F_i}$, $\boldsymbol{U_i}$, $A$)
15.     // Update $\Delta_v$
16.     $\Delta_v$=DYNAMIC-$\Delta_v$-KERNEL<<<>>>($\boldsymbol{vals}$, $\boldsymbol{U_i}$, $\boldsymbol{R_i}$, $A$)
17.     // Update $\Delta_i$
18.     $\Delta_i$ = DELTA-UPDATE-KERNEL<<<>>>($\boldsymbol{vals}$, $\boldsymbol{U_i}$, $\boldsymbol{\Delta_v}$)
19.     // Update $F_{i+1}$
20.     FRONTIER-KERNEL<<<>>>($\Delta_i$, $\boldsymbol{F_i}$, $\boldsymbol{U_i}$)
```

**Algorithm** *DYNAMIC-$\Delta_v$-KERNEL*($\boldsymbol{U_i}, \boldsymbol{R_i}, \boldsymbol{\Delta_v}, \boldsymbol{vals}, A, \tau$)

```
1.  $n_{idx}$ = thread.x.id
2.  $h_{idx}$ = thread.y.id
3.  **if** $R_i[n_{idx}]$==true
4.      **then** $\tau = vals[h_{idx}][n_{idx}]$
5.          **for** each $u \in \Gamma_{n_{idx}} \cap U_i$ **do**
6.              edge $e = (n_{idx}, u)$
7.              **for** each $q' \in \delta(h_{idx}, label(e))$ **do**
8.                  **if** $\Delta_v[n_{idx}] > f_{time}(e, \tau)$
9.                      **then** $\Delta_v[n_{idx}] = f_{time}(e, \tau)$
```

**Algorithm** *RELAX-KERNEL*($\boldsymbol{vals}, \boldsymbol{preds}, \boldsymbol{F_i}, \boldsymbol{U_i}, A$)

```
1.  $n_{idx}$ = thread.x.id
2.  $h_{idx}$ = thread.y.id
3.  **if** $F_i[n_{idx}]$ == true
4.      **then** $\tau = vals[h_{idx}][n_{idx}]$
```

```
5.        for each $w \in \Gamma_{n_{idx}}$ do
6.            edge $e = (n_{idx}, w)$
7.            for each $q' \in \delta(h_{idx}, label(e))$ do
8.                if $U_i[w] == \text{true}$
9.                    then Start Atomic Operations
10.                       $vals[h_{idx}][w] = min(vals[h_{idx}][w], vals[h_{idx}][n_{idx}] + f_{time}(e, \tau))$
11.                       $preds[h_{idx}][w] = n_{idx}$
12.                       End Atomic Operations
```

The *GPU T-D-RegL-CSPP* algorithm has the same behaviors of the original one, except for the *Dynamic-Delta$_v$-Kernel* at line 14, described in 6.4.2. The *Delta-Update-Kernel* and *Frontier-Kernel* are the same described in 6.4.1.2, only changing the input data. In *Relax-Kernel* we introduced the automaton $A$ that regulates, at line 7, the transition to the $w \in \Gamma$ nodes. Here we have two indexes to address the *vals* and the *preds* matrices.

The $h_{idx}$ index represent the traveler's state and the index $n_{idx}$ the node to expand. In this case we use a bi-dimensional block indexing to create the CUDA computational grid on the GPU. In lines 9-12 we have the atomic updates for the tentative distances and the predecessors. The $\tau$ variable (line 4), represent the actual time, used to evaluate the edge cost with the $f_{time}$ function in line 10. The *Dynamic-$\Delta_v$-Kernel* exploit the same bi-dimensional indexing used for the *Relax-Kernel*, in line 5 we select the successors of the $n_{idx}$ vertex that can be in the $i + 1$ iteration regulated by the automaton, line 7, and we evaluate the cost of the edge at line 8, using the $f_{time}$ function, updating, if necessary, the $\Delta_v$ values in line 9.

## 6.5 Shared Memory Algorithm

Analyzing in detail the serial algorithm, we can observe that we can compute, for each traveler state $h$ a T-D-RegL-CSPP in a state-space of labels at least equal to the dimension of the graph's nodes set $\boldsymbol{V}$. Under this prospective, we can describe the state-space in a bi-dimensional matrix with $h$ rows, one for each traveler state, and $|\boldsymbol{V}|$ columns. For each $h$ state, we can compute a Time-Dependent-RegL-CSPP as described by the algorithm *T-D-RegL-CSPP* completely decoupled from the others. In fact we will have independent data structures (the $h$ row in the matrices for the predecessors and the tentative distances) and an independent priority queue. Noticed these peculiarities, we can re-write the algorithm exploiting the characteristics described above:

```
Algorithm T-D-RegL-CSPP2($s, t, \mathbf{V}, \mathbf{E}, f_{time}, A, \tau_{start}$)
1.  Queue Q=null // Generated States of $G^{\times}$ queue
2.  vals, preds
3.  // Data Structures Initialization
4.  for $h = 1$ to $|TravellerStates|$ do
5.      for $n = 1$ to $|V|$ do
6.          if $n == s$
7.            then $vals[h][n] = \tau_{start}, preds[h][n] = 0$
8.            else  $vals[h][n] = \infty, preds[h][n] = \infty$
9.  // Algorithm
10. for $h = 1$ to $|TravellerStates|$ do
11.     // Queue Initialization
12.     Q.Insert((s))
13.     while SQ $\neq$ null do
14.         $n$ =SQ.First()
15.         $\tau = vals[h][n]$
16.         if $(n, h) \in \mathcal{T}$
17.           then break
18.         for each $u \in \Gamma_n$ do
19.             edge $e = (n, u)$
20.             for $q' \in \delta(h, label(e))$ do
21.                 if $vals[h][u] > vals[h][n] + f_{time}(e, \tau)$
22.                   then $vals[h][u] = vals[h][n] + f_{time}(e, \tau)$
23.                        $preds[h][u] = n$
24.                        if Q.InQueue($u$)
25.                          then Q.Update($u$)
26.                          else  Q.Insert($u$)
```

In lines 4-8 we initialize the tentative distance matrix *vals* and the predecessors matrix *preds*. For each traveler state $h$ we initialize the start node to $\tau_{start}$ and the predecessor to 0.

In lines 10-26 we evaluate, for each traveler state, the T-D-RegL-CSPP as done in the original algorithm. As we can see, every $h$ state is independent from the others, allowing us to evaluate the states in parallel.

### 6.5.1 OpenMP Porting

OpenMP API [113] seems to be the best option to parallelize the algorithm proposed in the previous paragraph. In fact, we can easily parallelize the method, using the fork/join programming model, assigning to each thread a traveler state. With few pre-processors directives, we can exploit the parallelism inside multi-core/multi-threaded CPUs.

```
Algorithm T-D-RegL-CSPP OMP($s, t, \mathbf{V}, \mathbf{E}, f_{time}, A, \tau_{start}$)
1.  Queue Q=null // Generated States of $G^\times$ queue
2.  $\boldsymbol{vals}, \boldsymbol{preds}$
3.  // Data Structures Initialization
4.  for $h = 1$ to $|TravellerStates|$ do
5.      for $n = 1$ to $|V|$ do
6.          if $n == s$
7.             then $vals[h][n] = \tau_{start}, preds[h][n] = 0$
8.             else  $vals[h][n] = \infty, preds[h][n] = \infty$
9.  // Algorithm
10. SetThreads($|TravellerStates|$)
11. # start parallel region
12. $h$ = GetThreadID()
13. // Queue Initialization
14. Q.Insert((s))
15. while Q $\neq$ null do
16.       $n$ =Q.First()
17.       $\tau = vals[h][n]$
18.       if $(n, h) \in \mathcal{T}$
```

```
19.             then break
20.         for each $u \in \Gamma(n)$ do
21.             edge $e = (n, u)$
22.             for $q' \in \delta(h, label(e))$ do
23.                 if $vals[h][u] > vals[h][n] + f_{time}(e, \tau)$
24.                     then $vals[h][u] = vals[h][n] + f_{time}(e, \tau)$
25.                         $preds[h][u] = n$
26.                         if Q.InQueue($u$)
27.                             then Q.Update($u$)
28.                             else  Q.Insert($u$)
29. # end parallel region
```

As we can see, the modification to the code are minimal. We introduce only the directives in pseudo-code to the pre-processor in lines 9-11. In line 11 we simply get the thread id and use it to indexing the state. In line 9 we spawn a thread for each state. In line 10 we declare the parallel section (fork) and, finally, in line 29 we close it (join).

## 6.6 Computational Results

We tested our parallel methods reporting the *Speed-Up* factor with respect to the serial version of the algorithm. Out test machine is a workstation equipped with a Intel Core i7 920 @ 2,9 GHz with 6 Gigabytes of RAM and a GPU Nvidia GTX570 with 1,280 Gigabyte of GDDR5 memory on board, 480 CUDA Cores @ 1,464 GHz. For each instance we report the serial time, the relative parallel time and the *Speed-Up* factor obtained. For each instance, we evaluated the performance for an increasing number of traveller state $h$ (2,3,4,6,8) and, consequentially, for a bigger state-space for the algorithm. The time horizon is one day and the granularity of time is expressed in seconds (one day = 86400 seconds) . The Speed-Up factor is computed: $SpeedUp = Time_{serial}/Time_{parallel}$.

### 6.6.1 Data Sets Creation

The data set used as benchmark for the methods is a set of 10 instances generated as follows:

- we selected 10 urban centres from the OSM [114] repository,
- extracted the relative graph, keeping trace of the streets classes ( primary, secondary, etc . . . ),
- we created 4 dense random graphs of 500000, 400000, 300000 and 200000 nodes, for the modes over the geographical network,
- connected with transition edges all the nodes of the random graphs to the geographical network,
- connected with transition edges the modes among them.

We used the street classes from OSM to calculate the journey time using the speed limits relative to that class and the speed profile described in 6.2.4 as penalties. For the other modes, we assume that represent the public transport networks (metro, bus, trams, etc . . . ) we used a constant speed limit and the speed profiles for the public transport described in the previous sections. The path is calculated as depicted in figure 6.4, from an extreme point to another of the area, to force the method to visit all the graph.

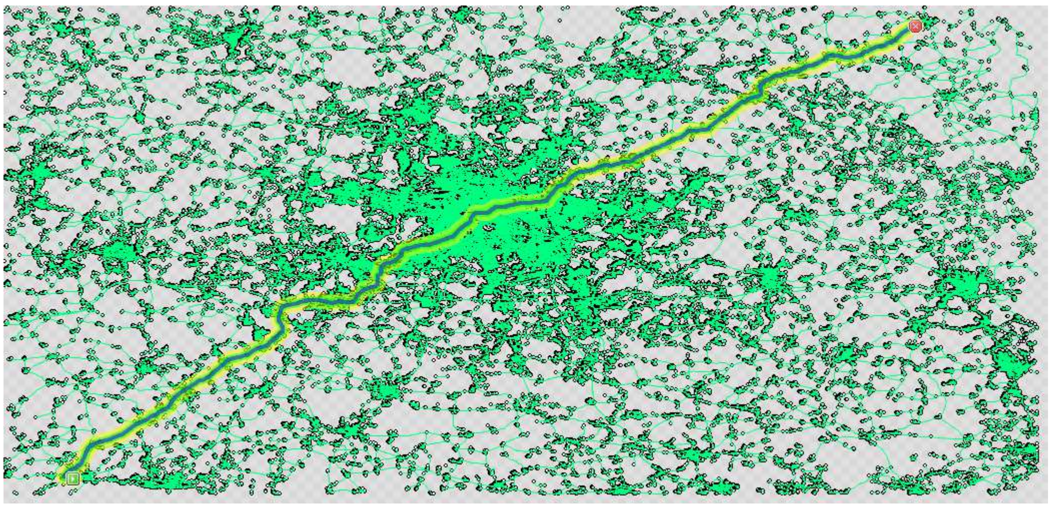


Figure 6.4: Test Path in Berlin

### 6.6.2 Results

In this section we provide the experimental results for the instances described above. First, we give some data relative to the instances dimensions, then, the relative computational times for the serial implementation, the GPU implementation and the parallel CPU implementation. We indicate with $\Phi = TravelerStates * \boldsymbol{V}$ the dimension of the state-space.

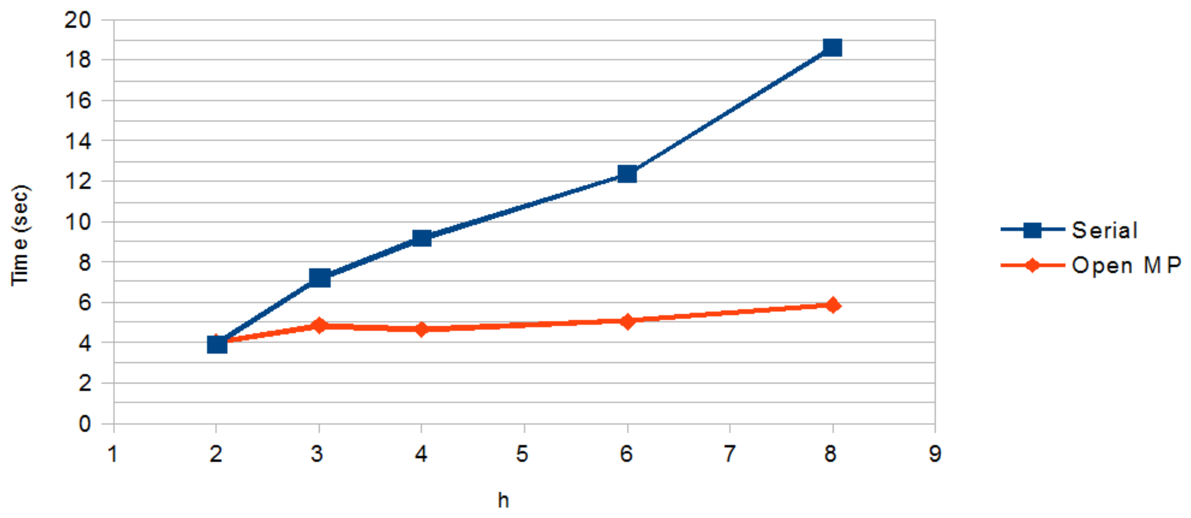


FIGURE 6.5: Serial times vs OpenMP times for the Berlin instance

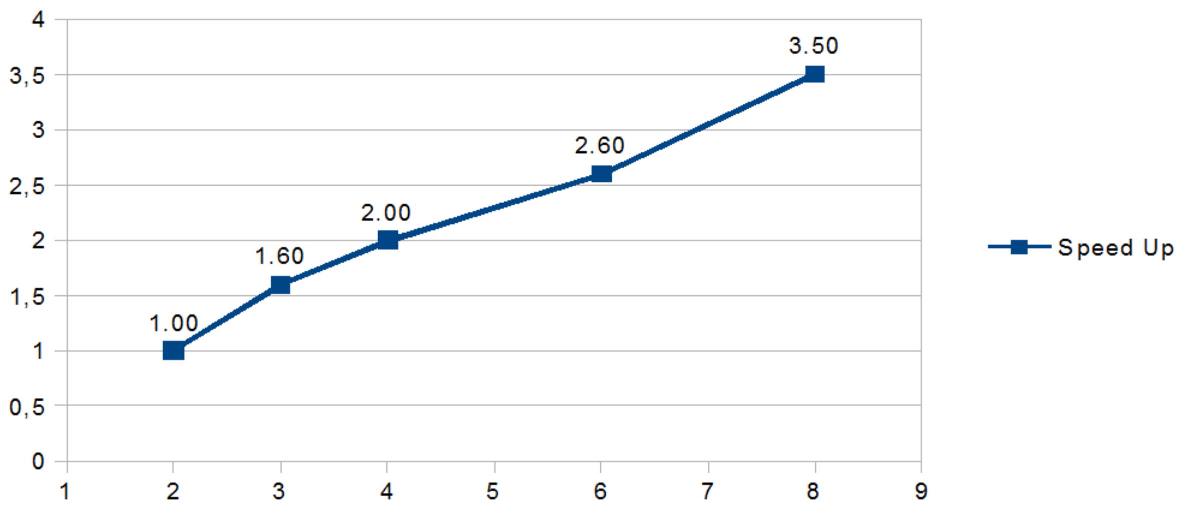


FIGURE 6.6: Speed-Up for different h values for the Los Angeles instance

Table 6.1: Test instances dimensions

| Instance | Nodes | Edges | Modes | Φ (h = 2) | Φ (h = 3) | Φ (h = 4) | Φ (h = 6) | Φ (h = 8) |
|---|---|---|---|---|---|---|---|---|
| Berlin | 1594184 | 26360404 | 5 | 3188368 | 4782552 | 6376736 | 9565104 | 12753472 |
| London | 1962806 | 27134087 | 5 | 3925612 | 5888418 | 7851224 | 11776836 | 15702448 |
| Los Angeles | 1845707 | 26957936 | 5 | 3691414 | 5537121 | 7382828 | 11074242 | 14765656 |
| Melbourne | 1636709 | 26438780 | 5 | 3273418 | 4910127 | 6546836 | 9820254 | 13093672 |
| Milan | 1515734 | 26154972 | 5 | 3031468 | 4547202 | 6062936 | 9094404 | 12125872 |
| Moskow | 1621233 | 26423233 | 5 | 3242466 | 4863699 | 6484932 | 9727398 | 12969864 |
| New York | 1650314 | 26519156 | 5 | 3300628 | 4950942 | 6601256 | 9901884 | 13202512 |
| Paris | 1654535 | 26457090 | 5 | 3309070 | 4963605 | 6618140 | 9927210 | 13236280 |
| Rome | 1490676 | 26098377 | 5 | 2981352 | 4472028 | 5962704 | 8944056 | 11925408 |
| Tokyo | 2600054 | 29232779 | 5 | 5200108 | 7800162 | 10400216 | 15600324 | 20800432 |

Table 6.2: Computational results on CPU

| Instances | Serial | | | | | OpenMP | | | | | SpeedUp | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | h = 2 | h = 3 | h = 4 | h = 6 | h = 8 | h = 2 | h = 3 | h = 4 | h = 6 | h = 8 | h = 2 | h = 3 | h = 4 | h = 6 | h = 8 |
| Berlin | 3.90491 | 7.19491 | 9.16738 | 12.35290 | 18.62190 | 4.00861 | 4.83804 | 4.64429 | 5.04666 | 5.84201 | 1.0 X | 1.5 X | 2.0 X | 2.4 X | 3.2 X |
| London | 4.27784 | 7.43718 | 9.51450 | 13.39980 | 20.09110 | 4.12370 | 4.51029 | 4.80851 | 5.20207 | 6.24187 | 1.0 X | 1.6 X | 2.0 X | 2.6 X | 3.2 X |
| Los Angeles | 4.31991 | 6.99828 | 9.12968 | 12.88160 | 20.42397 | 4.22763 | 4.33566 | 4.63512 | 5.00295 | 5.91230 | 1.0 X | 1.6 X | 2.0 X | 2.6 X | 3.5 X |
| Melbourne | 4.00122 | 6.65853 | 8.29722 | 10.70190 | 16.39150 | 3.97519 | 4.33418 | 4.68507 | 5.03967 | 5.89474 | 1.0 X | 1.5 X | 1.8 X | 2.1 X | 2.8 X |
| Milan | 3.98671 | 6.97139 | 9.37055 | 12.58200 | 18.68020 | 4.05487 | 4.42006 | 4.72360 | 5.10111 | 5.82905 | 1.0 X | 1.6 X | 2.0 X | 2.5 X | 3.2 X |
| Moskow | 3.91440 | 6.70570 | 9.01145 | 11.84520 | 18.17520 | 3.99721 | 4.29074 | 4.65833 | 4.92247 | 5.77115 | 1.0 X | 1.6 X | 1.9 X | 2.4 X | 3.1 X |
| New York | 4.02293 | 6.91773 | 9.29118 | 12.68630 | 19.21360 | 3.85927 | 4.25159 | 4.71184 | 4.96807 | 5.82088 | 1.0 X | 1.6 X | 2.0 X | 2.6 X | 3.3 X |
| Paris | 4.17275 | 7.10199 | 9.21067 | 12.81390 | 19.63470 | 4.03054 | 4.39365 | 4.73818 | 5.13069 | 5.87103 | 1.0 X | 1.6 X | 1.9 X | 2.5 X | 3.3 X |
| Rome | 3.88276 | 6.86815 | 8.94360 | 12.50710 | 18.54690 | 3.76575 | 4.09594 | 4.49787 | 5.02301 | 5.71587 | 1.0 X | 1.7 X | 2.0 X | 2.5 X | 3.2 X |
| Tokyo | 2.91007 | 6.02751 | 9.13981 | 13.20910 | 16.69850 | 2.53121 | 4.39309 | 4.97617 | 5.24188 | 5.91094 | 1.1 X | 1.4 X | 1.8 X | 2.5 X | 2.8 X |

Table 6.3: Computational results on GPU

| Instances | Serial | | | | | CUDA | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *Name* | h = 2 | h = 3 | h = 4 | h = 6 | h = 8 | h = 2 | h = 3 | h = 4 | h = 6 | h = 8 |
| Berlin | 3.90491 | 7.19491 | 9.16738 | 12.35290 | 18.62190 | 15.29472 | 35.38592 | 45.31485 | 60.45892 | 90.10293 |
| London | 4.27784 | 7.43718 | 9.51450 | 13.39980 | 20.09110 | 15.85713 | 34.75631 | 44.36134 | 61.98234 | 100.42816 |
| Los Angeles | 4.31991 | 6.99828 | 9.12968 | 12.88160 | 20.42397 | 14.84052 | 35.02742 | 44.23401 | 62.09213 | 100.90235 |
| Melbourne | 4.00122 | 6.65853 | 8.29722 | 10.70190 | 16.39150 | 14.94752 | 34.46021 | 46.28461 | 61.45213 | 88.16702 |
| Milan | 3.98671 | 6.97139 | 9.37055 | 12.58200 | 18.68020 | 15.95023 | 33.67302 | 42.01237 | 62.23768 | 90.56330 |
| Moskow | 3.91440 | 6.70570 | 9.01145 | 11.84520 | 18.17520 | 14.95064 | 31.10204 | 43.47502 | 60.14586 | 91.09123 |
| New York | 4.02293 | 6.91773 | 9.29118 | 12.68630 | 19.21360 | 15.88630 | 36.67120 | 44.65321 | 60.27451 | 92.34551 |
| Paris | 4.17275 | 7.10199 | 9.21067 | 12.81390 | 19.63470 | 15.68335 | 34.05063 | 45.75230 | 63.35672 | 91.76812 |
| Rome | 3.88276 | 6.86815 | 8.94360 | 12.50710 | 18.54690 | 15.95033 | 33.12574 | 45.78124 | 61.01203 | 90.01445 |
| Tokyo | 2.91007 | 6.02751 | 9.13981 | 13.20910 | 16.69850 | 14.12753 | 35.69305 | 42.87932 | 60.36789 | 86.13599 |

## 6.7 Considerations and Future Work

As it emerged from the experimental test, for this particular problem, the GPU algorithm is not effective. The main reason is that the geographical network is a really sparse graph and the frontier set's $F_i$ cardinality is too small to allow the GPU to release its massive parallelism. The multi-core/ OpenMP version, instead, brings results near to the theoretical speed-up for a quad-core processor for the bigger instances. The other bottleneck for the GPU performance is the load balancing inside the device where most part of the computational kernel is composed by serial operations. The new Nvidia chips, the Maxwell series, partially resolve this problem, introducing a new feature called *Dynamic Parallelism*, allowing the kernel function to call another kernel, making the load balancing relative to non-uniform data structures (e.g. adjacency lists) more effective. As soon as possible we will provide a method exploiting this new feature.

# Chapter 7

# Membership Overlay Problem

In this chapter we will considerate a parallel version of the Subgradient Method, used to solve the Dual Lagrangean Problem. We choose a network design problem, the Membership Overlay Problem (MOP), relative to the Peer-to-Peer networks. We designed three parallel algorithms for the GPU Computing, Shared Memory and Distributed Memory environments exploiting, respectively, CUDA, OpenMP and MPI.

## 7.1 Introduction

Peer-to-Peer (P2P) networks actually represent a conspicuous part of the internet data traffic. This model, indeed, is the counterpart of the well-known and studied client-server model. A large number of network applications (legal or not) are already adopting this network model.

The proliferate of this kind of networks has brought to the attention of the academic community some challenging problems relative to the P2P model. In literature, for example, doesn't exist a precise and coherent definition and a precise and exhaustive description is hard to find.

Informally speaking, a P2P network is a totally decentralized network, composed by peers that share, exchange and distribute data and information. The success of this kind of networks is related to the anonymous identity of the members, allowing, in some case, the exercise of not totally legal trades.

Another peculiarity of P2P is the dynamic topology of the network. A peer can connect to the network for a limited time and share its data or its bandwidth with the others. Once disconnected, the topology changes and some routes for the data packets or connections must be modified to maintain the network performances.

This strictly dynamic feature is definitely interesting and has arose critical problems for some applications. The P2P paradigm can be a fundamental part of large scale distributed computation infrastructures or grid computing applications. The dynamic nature of the network topology implies a significant degradation of the performance or a non-optimal configuration of communications and connections, compromising the system scalability.

Most of the P2P applications are based on the TCP/IP communication protocol and, virtually, all the peers are connected to each other. A P2P network, instead, is at application level in the ISO/OSI stack having their routing and topology.

In this scenario, network design problems seem critical, mainly the ones dealing with the optimization of the connections for enhancing the network's performances.

The *Membership Overlay Problem* (*MOP*) is one of these network design problems that can arise in relation with a P2P environment. The problem consists in the creation of an overlay network that maximizes the bandwidth throughput among the peers inside the P2P application.

In this chapter we will describe a Lagrangean Relaxation for the problem and a Distributed Subgradient for solving the relative Lagrangean Dual Problem proposed originally by Boschetti et al. [115]. We will propose three parallel algorithms based on this Subgradient, designed to exploit three 'de-facto' standard parallel programming models, CUDA, OpenMP and MPI, relative to GPU, Shared Memory and Distributed Memory environments respectively.

## 7.2 Membership Overlay Problem

In this section we will illustrate the Membership Overlay Problem, giving first an informal definition, then a mathematical formulation, describing the inherent characteristics. MOP is a network design problem, in fact is focused on the creation of a network topology following precise performance, fault tolerance or efficiency constraints.

Network Design is actually one of the most interesting and studied class of problems in the CO field. It affects many real-world applications like supply chain logistics or telecommunications.

The problem consists in maximizing the throughput of a P2P network where the nodes (peers) are described by a percentage of on-line time and a bandwidth to the internet. The edges are the connections among these nodes, with a capacity. The solution of the problem is a subset of these edges that maximize the throughput, creating an overlay network defining a topology among the nodes.

Before the mathematical formulation, we will give an example to illustrate the problem.

Referring to the figure 7.1, we depicted a graph representing a network with the characteristics cited before Without lack of completeness, from now we will represent the network as a graph with nodes and edges.

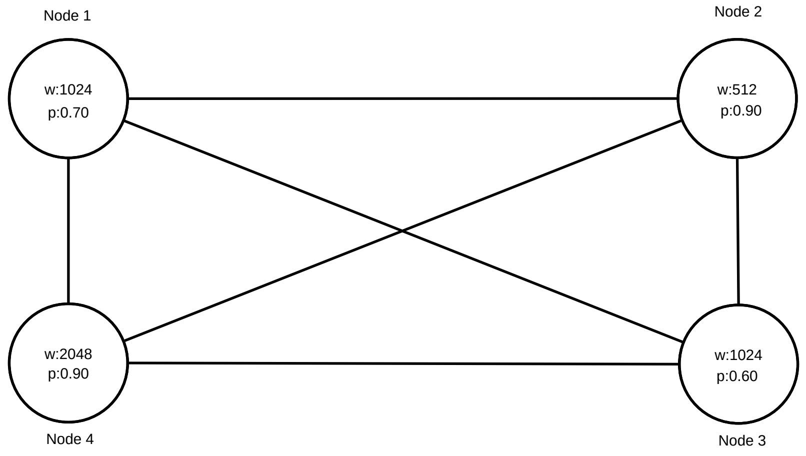


FIGURE 7.1: P2P network with 4 nodes.

Every node has a percentage, $p$, to be on-line and a bandwidth, $w$, representing the bandwidth of the internet connection. The graph is compete, due to the TCP/IP protocol. The edges are not oriented, and we can assume that are bi-directional. The edges have a capacity $b$ equal to the minimum value $w$ of the nodes that connects as described in figure 7.2 .

Every edge has a probability, $p'$, equals to the product of the $p$ probabilities of its extremes (figure 7.3). Once characterized the graph, we need to set other parameters that constraint the problem's solution. Birattari et al. [116] and Rardin and Uzsoy [117], proposed these parameters. To guarantee a minimum

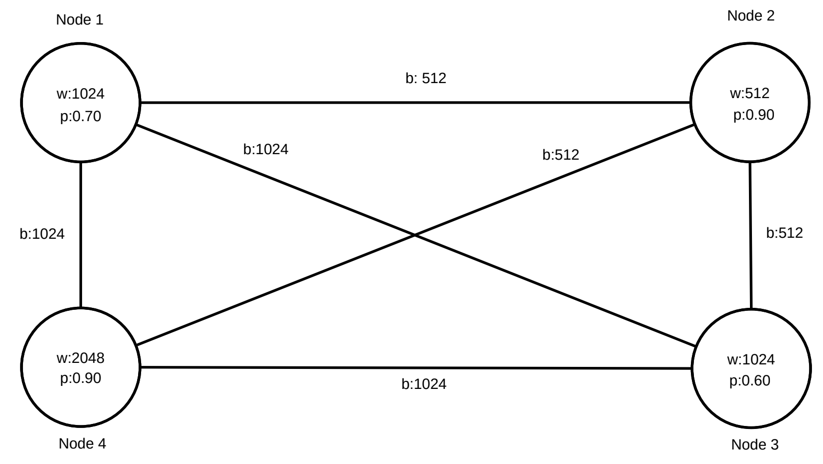


FIGURE 7.2: Edges bandwidth values.

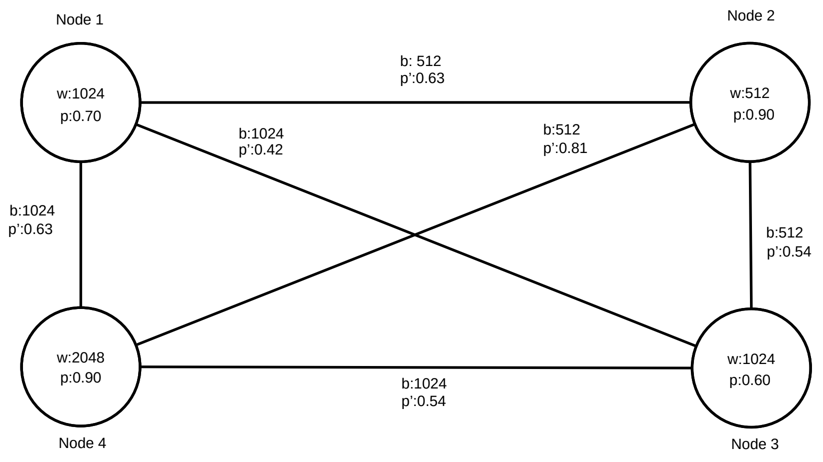


FIGURE 7.3: Edges p' values.

QoS (Quality of Service) is necessary to set a bandwidth lower bound, $l$, for each edge of 14 Kbps for instance, as shown in figure 7.4.

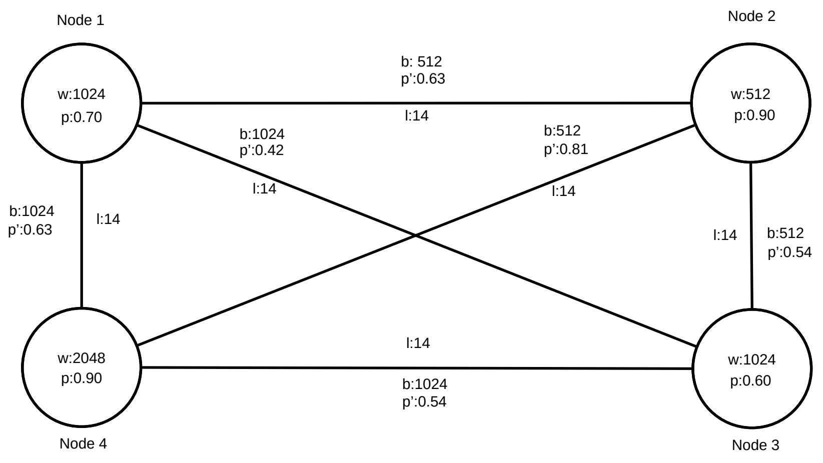


FIGURE 7.4: Edges lower bound $l$.

Similarly, is necessary to set a bandwidth upper bound, $u$, for each edge to avoid the congestion of the node and the network. For the results cited above, we can set this value to 256 Kbps (figure 7.5).

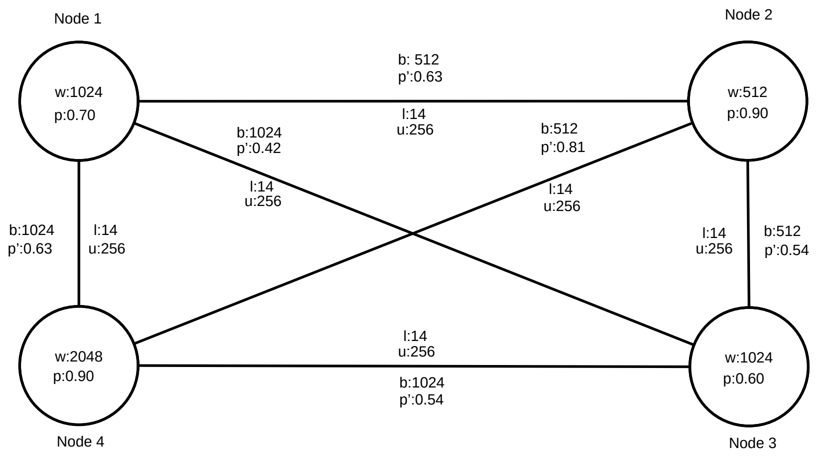


FIGURE 7.5: Edges upper bound $u$.

In figure 7.6 is described the optimal solution for the proposed example.

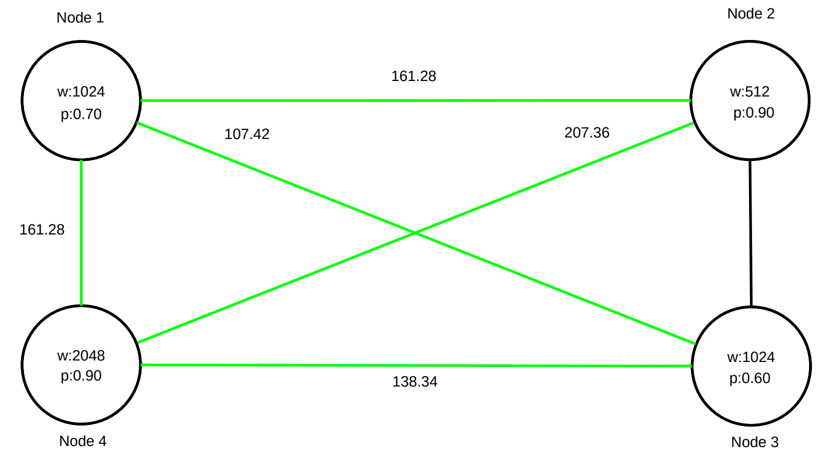


FIGURE 7.6: MOP Optimal Solution.

More in details, the value for an edge is given by $u_{ij}p'_{ij}$ and the optimal value of the problem will be the sum of these values relative to the edges in solution: $\sum u_{ij}p'_{ij}$. In figure 7.6 we show the problem's solution, coloring in green the edges. The final values is 775.68 and the edge $(2, 3)$ has been rejected from the solution. The resulting sub-graph is the overlay network that maximize the throughput for the P2P network.

### 7.2.1 Definition and Mathematical Formulation

Given a graph $G = (V, E)$ where $n = |V|$, the vertices are the peers of the P2P network and the edges the possible connections among the vertices. Two edges $i$ and $j$ are connected by the edge $(i, j)$ if both nodes can send messages each others, using the underlying routing structure (the internet typically). Each node $i$ can enter and exit the network, according to the P2P model, and, when is on-line, can share a limited amount of bandwidth. Each node is characterized by two weights: $p_i$ e $w_i$, respectively the connection time, expressed in percentage ($1 =$ always on line, $0 =$ never on line) like described by Saroiu et al. [118] and the available bandwidth of its connection. $\delta(i)$ is the neighbors set for the node $i$.

The Membership Overlay Problem consists in finding a sub-graph $G' = (V, E')$ from $G$. The edges of $G'$ define that two nodes decide to allocate a part of their bandwidth to communicate between them. If $b_i$ e $b_j$ are the bandwidths of node $i$ and $j$, the available bandwidth for the edge $(i, j)$ is $b_{ij} = min\{b_i, b_j\}$. The two bandwidth values can be equal to the $w_i$ and $w_j$ values, saturating the nodes or inferior in relation to the other nodes in the graph. Anyway, we will define an upper bound,$u_{ij}$ and a lower bound, $l_{ij}$, to guarantee a minimum QoS and to avoid the network's congestion, respectively.

The graph $G'$ obtained will have the following peculiarities:

1. the global throughput is maximized;
2. the graph $G'$ diameter is logarithmic, creating a connected graph;
3. the total bandwidth used by each node $i$ is less or equal to $b_i$.

From these first considerations, we can assume that for each node $i$ is not necessary the global knowledge of the graph, but only the state of the nodes in $\delta(i)$.

Following the definition given by [117], we can define MOP a *control* problem that : 'must be solved frequently and it involves decision over a relatively short horizon'. Solutions for these kind of problems 'must be obtained in near real time, algorithms must run in fractions of seconds' and 'quality matters somewhat less' than speed.

This kind of problems is often linked to critical applications in robotics or automations in general, also in high-precision tools or car control units. Taking into account these factors, seems mandatory design fast and effective algorithms for giving a good solution to the problem in a short execution time.

In the next section we will provide a Mixed Integer formulation for the static version of MOP (SMOP), from this formulation will be derived a polynomial upper bound and a relaxation framework.

### 7.2.2 MIP Formulation

Static Membership Overlay Problem can be formulated as follows. We have two set of decision values $\{x_{ij}\}$ and $\{\xi_{ij}\}$, $(i,j) \in E$. The $x_{ij}$ continuous variables defines the bandwidth between $i$ e $j$ and $0 \leq x_{ij} \leq u_{ij}$. The binary variables $\xi_{ij}$ are 1 if the edge $(i,j)$ is used to connect the nodes, 0 otherwise. The MIP formulation is the following:

$$\begin{aligned} z_{SMOP} = max \sum_{(i,j)\in E} p_{ij}x_{ij} & & (1) \\ s.t. \sum_{j\in\delta(i)} x_{ij} \leq b_i, & \quad i \in V & (2) \\ l_{ij}\xi_{ij} \leq x_{ij} \leq u_{ij}\xi_{ij}, & \quad (i,j) \in E & (3) \\ \xi_{ij} \in \{0,1\}, & \quad (i,j) \in E & (4) \end{aligned} \tag{7.1}$$

where $p_{ij} = p_i \times p_j$ for each edge $(i,j) \in E$ and $\delta(i)$ represent the neighborhood of $i$ in $G$ (in our case $\delta(i) = V/\{i\}$, $G$ is complete). The objective function (1) maximizes the total bandwidth given by the sum of all the assigned bandwidths to each connection $(i,j) \in E$ weighted by their up-times (on-line times) $p$. The constraints (2) ensure that the bandwidth assigned to the $i$ node does not exceed the limit $b_i$.

Static Membership Overlay Problem is an NP-Hard problem, setting $l_{ij} = u_{ij}$, for each edge $(i,j) \in E$. SMOP can be described as a Multidimensional Knapsack Problem:

$$
\begin{aligned}
z_{MKP} = max \sum_{(i,j)\in E} p_{ij}u_{ij}\xi_{ij} & & (5)\\
s.t. \sum_{j\in\delta(i)} u_{ij}\xi_{ij} \le b_i, & \quad i \in V & (6)\\
\xi_{ij} \in \{0,1\}, & \quad (i,j) \in E & (7)
\end{aligned}
\tag{7.2}
$$

That is the generalization for the Knapsack 0-1 where the bin has more than one dimensional constraint and the objective is to maximize the use of the bin.

## 7.3 Linear and Lagrangean SMOP Relaxations

### 7.3.1 LP Relaxation

The SMOP LP relaxation, $z_{LP}$, is the following:

$$
\begin{aligned}
z_{LP} = max \sum_{(i,j)\in E} p_{ij}x_{ij} & & (8)\\
s.t. \sum_{j\in\delta(i)} x_{ij} \le b_i, & \quad i \in V & (9)\\
l_{ij}\xi_{ij} \le x_{ij} \le u_{ij}\xi_{ij}, & \quad (i,j) \in E & (10)\\
0 \le \xi_{ij} \le 1 & \quad (i,j) \in E & (11)
\end{aligned}
\tag{7.3}
$$

We relaxed the (7) constraint, making it continuous: $0 \le \xi_{ij} \le 1$. This relaxation gives us an upper bound to the integer solution for SMOP, bound that we will use to evaluate the effectiveness of the Lagrangean bound that we will describe in the next section. We can easily compute the LP relaxation using a linear solver (CoinMP or CPLEX).

### 7.3.2 Lagrangean Relaxation

The Lagrangean relaxation for SMOP can be formulated associating a non negative penalty $\lambda_i$ to each constraint (2). The formulation, called $z_{LR}(\boldsymbol{\lambda})$, is the following:

$$\begin{aligned} z_{LR}(\boldsymbol{\lambda}) = max \sum_{(i,j)\in E} p'_{ij}x_{ij} + \sum_{i\in V} b_i\lambda_i & & (12) \\ s.t.\, l_{ij}\xi_{ij} \le x_{ij} \le u_{ij}\xi_{ij}, \quad & (i,j)\in E & (13) \\ \xi_{ij} \in \{0,1\}, \quad & (i,j)\in E & (14) \end{aligned} \tag{7.4}$$

that is equivalent to the problem:

$$\begin{aligned} z_{LR}(\boldsymbol{\lambda}) = max \sum_{(i,j)\in E} p'_{ij}x_{ij} + \sum_{i\in V} b_i\lambda_i & & (15) \\ s.t.\, 0 \le x_{ij} \le u_{ij}, \quad & (i,j)\in E & (16) \end{aligned} \tag{7.5}$$

where $p'_{ij} = p_{ij} - \lambda_i - \lambda_j$ and the $\{\xi_{ij}\}$ variables are not necessary.

Given $\lambda$, the optimal value of $z_{LR}(\boldsymbol{\lambda})$ is computed according to the following observations :

- if $p'_{ij} \ge 0$ we use all the bandwidth available, then the edge will be part of the solution: $\xi_{ij} = 1$ e $x_{ij} = u_{ij}$
- if $p'_{ij} < 0$, the connection is discarded: $\xi_{ij} = 0$ e $x_{ij} = 0$

The Dual Lagrangean Problem associated can be described as follows:

$$z_{LR}(\boldsymbol{\lambda^*}) = min\{z_{LR}(\boldsymbol{\lambda}) : \boldsymbol{\lambda} \ge 0\} \tag{7.6}$$

## 7.4 Subgradient Algorithm

The Subgradient algorithm proposed by Shore in [43] and successfully used by [39–42] is an iterative procedure that, at each iteration $k$, computes a new approximation $\boldsymbol{\lambda^{k+1}}$ of the Lagrangean multipliers in such a way that, for $k \to +\infty$, $\lambda^k$ is an optimal or near optimal solution of the corresponding Lagrangean Dual. Let $x^k$ of cost $z_{LR}(\boldsymbol{\lambda^k})$ obtained at iteration $k$ by solving problem 7.5 setting $\boldsymbol{\lambda^k} = \boldsymbol{\lambda^{k+1}}$. The Lagrangean multipliers can be updated as follows:

$$\lambda_i^{k+1} = max\{0, \lambda_i^k + \alpha^k g_i^k\}, \quad i \in V \tag{7.7}$$

where:

$$g_i^k = \sum_{j \in \delta(i)} x_{ij}^k - b_i, \quad i \in V \tag{7.8}$$

is the $i$-th component of the subgradient $\boldsymbol{g^k}$ and $\alpha^k$ is the length of the step along the search direction given by the subgradient itself. In literature has been proposed several versions of the step size $\alpha^k$ update. In this chapter we will take in consideration the standard one proposed by Polyak [119] and a constant one (called *quasi-constant*) for a fully distributed algorithm proposed by [115].

The standard update rule can be described as follows:

$$\alpha^k = \beta^k \frac{\bar{z} - z_{LR}(\boldsymbol{\lambda^k})}{\|\boldsymbol{g^k}\|} \tag{7.9}$$

where $\bar{z}$ is an overestimate of $z_{LR}(\boldsymbol{\lambda^*})$. Polyak proved the convergence of the method for $\epsilon \leq \beta^k \leq 2$. Instead of a overestimate of the Lagrangean function, is possible, in many application, substitute it with:

$$\alpha^k = \beta^k \frac{0.001 z_{LR}(\boldsymbol{\lambda^k})}{\|\boldsymbol{g^k}\|} \tag{7.10}$$

Usually, the $\beta^k$ step is initialized with a value dependent to the considered problem and updated (e.g $\beta^{k+1} = 0.5\beta^k$) if after and arbitrary number of iteration the $z_{RL}(\boldsymbol{\lambda^k})$ is not improved.

Our implementation of the standard subgradient algorithm implements the following update rule for the Lagrangean penalties:

$$\lambda_i^{k+1} = max\{0, \lambda_i^k + \beta \frac{0.01 z_{LR}(\boldsymbol{\lambda})}{\|\boldsymbol{g^2}\|} g_i^k\} \tag{7.11}$$

## 7.5 Distributed Subgradient

The Lagrangean relaxation proposed in 7.3.2 doesn't take into account the dynamic aspect of a P2P net, considering static the topology of the network. In fact, in a P2P environment, the nodes join and exit dynamically from the net and for each node is impossible to have a consistent view of the graph's status and the optimization process described above is not effective.

This aspect makes impossible to built an optimization process based on global parameters (e.g. the $\boldsymbol{g}$ subgradient).

Deeply analyzing the P2P environment we can observe that:

- each node has a local knowledge of the net. It is aware only of the status of its neighbors,
- each node can optimize its parameters only looking at the behavior of its neighbors,
- the potentially infinite time horizon of the net brigs to a constant optimization, based on the changing graph topology (e.g. selecting a new set of connections because of the exit of some nodes from the net),
- each node, computing its local optimization step, can contribute to the global optimization of the graph, in an asynchronous fashion.

Aware of these observations we can draw some guidelines for the design of a distributed algorithm:

- the optimization step is local, each node compute its values through its knowledge of the net and its connections,
- the optimization process must be asynchronous,
- the optimization process must be distributed among the nodes of the graph,
- once each node computed its optimization step, communicate to others its results and create the overlay network. When one or more nodes exit the net, each node optimize again, considering the new topology.

These considerations allow us to indicate the behavior of a distributed subgradient:

- each node will compute its Lagrangean penalty $\lambda_i$, according to its local knowledge,

- each node will compute its part of the global objective function,
- each node will update its penalty exploiting a local method,(e.g. $\|\boldsymbol{g}\|$ can't be considered),
- each node will compute its optimization step using the updated penalties from the other nodes.

We report, after the previous considerations, a dynamic, asynchronous and fully distributed subgradient. This approach doesn't solve the Lagrangean problem of the original graph $G$, but for each node of the graph. Given:

$$G_h(V_h, E_h) \tag{7.12}$$

containing only the $h$ node and its neighborhood $\delta(h)$, with:

$$\begin{aligned} V_h &= \delta(i) \cup \{h\} \\ E_h &= \{(i,j) \in E : i,j \in V_h\} \end{aligned} \tag{7.13}$$

and a set of Lagrangean penalties $\boldsymbol{\lambda} = \{\lambda_0, \ldots, \lambda_k\}$ with $k = 1, \ldots, |V_h|$ for each subgraph $G_h(V_h, E_h)$, $h \in V$, we solve the following problem:

$$\begin{aligned} z_{LR}^h(\boldsymbol{\lambda}) = max \sum_{(i,j)\in E_h} \frac{1}{2} p'_{ij} x_{ij} + b_h \lambda_h & \qquad\qquad (17) \\ s.t.\, 0 \le x_{ij} \le u_{ij}, \qquad\qquad (i,j) \in E_h & \qquad\qquad (18) \end{aligned} \tag{7.14}$$

This problem is derived directly from the 7.5, considering only the $h$ node. The global value of the objective function is given by $z_{LR}(\boldsymbol{\lambda}) = \sum_{h\in V} z_{LR}^h(\boldsymbol{\lambda})$.

The coefficient $\frac{1}{2}$ is included for avoiding the double evaluation of the node and its edges.

## 7.5.1 Algorithm foundations

The proposed formulation suggests an algorithm that follows the steps:

**Step 1** At each iteration $k$ each node $h$ request the Lagrangean penalties from its neighborhood $i \in \delta(h)$,

**Step 2** exploiting the new penalties, the node $h$ locally optimize, solving the problem $z^h_{LR}(\boldsymbol{\lambda})$ and computing its solution $x_{ij}$,

**Step 3** node $h$ updates its penalty$\lambda_h$, $\lambda_h = max\{0, \lambda_h + \alpha^k_h g_h\}$.

It's trivial to notice that we can't perform at each step the exchange of the $\lambda_h$ penalties because of the computational cost of the communications. To avoid this problem and enhancing the algorithm's performances, [115] proposed a two-level optimization process:

**I Level (Core Optimization)** An internal loop in which $h$ optimize its $z^h_{LR}(\boldsymbol{\lambda})$ value, using its $\lambda_h$ penalty and keeping constant the penalties $\lambda_i$ from its neighborhood,

**II Level (External Optimization)** once each node $h \in G$ completed their optimization step, each penalty is updated and sent to the neighborhood, then, completed the communication, it's possible to each node to perform another Core Optimization step.

### 7.5.2 Quasi-constant step size update

The main problem afflicting the distributed subgradient is the update of the $\lambda_h$ penalties. We can't use a standard step update rule, like the one described in 7.4. The $h$ node doesn't have a consistent knowledge of the network and to perform other communications will degrade the method's performances.

It's necessary use an update rule that exploit constant and local parameters. This rule is named *quasi-constant* step size rule:

**Step 1** initially $\alpha^0 = \alpha_{start}$ an arbitrary small value,

**Step 2** once the bound is improved, the step size is incremented $\alpha^{k+1} = \gamma\alpha^k$, con $\gamma > 1$, by means of a constant defined *a priori*,

**Step 3** if the bound is not improved for a given number of iterations, the step size is reduced $\alpha^{k+1} = max\{\gamma^{'}\alpha^k, \alpha_{min}\}$ where $0 < \gamma^{'} < 1$ and $0 < \alpha_{min} < \alpha_{start}$, otherwise $\alpha^{k+1} = \alpha^k$.

The subgradient $g_h^k$ is also computed using only the local $x_h$ solution.

The rule for updating the Lagrangean penalties is:

$$\lambda_h^{k+1} = max\{0, \lambda_h^k + \alpha_h^k g_h^k\} \tag{7.15}$$

### 7.5.3 Algorithm description

We can summarize the two level optimization procedure as follows:

**Algorithm** *Inner-Subgradient*$(h, \boldsymbol{\lambda'})$

```
1.  Initialize $\lambda_i = \lambda'_i$ for each $i \in V_h = \delta(h) \cup \{h\}$
2.  while $it < InnerMaxIter$ do
3.      Solve $z_{LR}^h(\boldsymbol{\lambda})$
4.      Update only the node $h$ penalty: $\lambda_h = max\{0, \lambda_h + \alpha_h g_h\}$
5.      if $z_{LR}^h(\boldsymbol{\lambda}) < z_{LR}^h(\boldsymbol{\lambda'})$
6.         then $\lambda'_h = \lambda_h$
```

**Algorithm** *External-Optimization*()

```
1.  while $et < ExtMaxIter$ do
2.      for $h = 1$ to $|V|$ do
3.          Inner-Subgradient($h$, $\boldsymbol{\lambda}$)
4.      Send each $\lambda_h$ to each $j \in \delta(h)$
```

The *External-Optimization* algorithm manages the external optimization step, calling the *Inner-Subgradient* procedure for each node of the graph (line 3). Each node $h$ do its optimization step, solving its Lagrangean Problem $z_{LR}^h(\boldsymbol{\lambda})$ (line 3), updating its penalty (line 4) and, re-optimizing with the new $\lambda$, for a given number of iterations. At the end of the main procedure *External-Optimization* the best penalties for each node is send to the other peers of the network (line 4).

## 7.6 Computational Results

In this section we propose the computational result regarding the distributed subgradient algorithm, compared to the standard one and the LP relaxation of the problem, computed using CPLEX and CoinMP.

The test sets are provided by Boschetti et al. [115]

We call STD the standard subgradient algorithm and DIST the distributed one. The standard subgradient is implemented using the standard update rule proposed by Polyak: $\alpha^k = \beta^k \frac{0.001 z_{LR}(\boldsymbol{\lambda^k})}{\|\boldsymbol{g^k}\|}$, with these parameters:

- $\beta^0 = 0.005$, start step,
- $\Delta_k = 75$, number of iterations before computing a smaller step: $\beta^{k+1} = 0.95\beta^k$,
- $MaxIter = 10000$, max number of iterations,
- *Stop Condition*: if the bound is not improved at least of 0.1% in the last 3000 iterations, the method is stopped.

The parameters of the DIST algorithm are:

- $\alpha_{start} = 0.0025$,
- $\alpha_{min} = 0.000005$,
- $\gamma = 1.005$,
- $\gamma' = 0.95$.
- $ExtMaxIter = 500$.
- $InnerMaxIter = 20$.
- $\Delta_k = 10$.

The CPLEX version is the 11.2 and the CoinMP one is 1.7. We report the average value of the instances in the set, the execution times and the Gap% ($Gap = 100 \times \frac{z_{LRopt} - z_{LP}}{z_{LP}}$) between the LP relaxation and the results computed by the subgradient algorithms.

The test machine is equipped with an Intel i7 Core 920 @ 2.8 GHz and 6 Gigabytes of RAM.

TABLE 7.1: Gaps and execution times for **LP**, **STD**, **DIST**.

| **Problem Group** | **LP** | | | **STD** | | **DIST** | |
|---|---|---|---|---|---|---|---|
| Name | $z_{LP}(AVG)$ | $CoinMP_t$ | $CPLEX_t$ | Time | Gap% | Time | Gap% |
| grafo50a-14 | 8181.31 | 0.06 | 0.04 | 0.61 | 0.0100 | 0.04 | 0.2810 |
| grafo100a-14 | 18898.20 | 0.23 | 0.05 | 2.78 | 0.0020 | 0.14 | 0.4070 |
| grafo250a-14 | 70508.01 | 3.69 | 0.35 | 7.89 | 0.0200 | 2.55 | 0.5430 |
| grafo500a-14 | 151190.40 | 92.96 | 1.91 | 52.65 | 0.0100 | 9.82 | 0.3800 |
| grafo750a-14 | 238698.34 | 761.80 | 4.79 | 150.09 | 0.0330 | 37.50 | 0.0000 |
| grafo1000a-14 | 316579.60 | 3203.72 | 10.95 | 330.12 | 0.0150 | 52.01 | 0.0001 |
| grafo50a-128 | 8271.55 | 0.06 | 0.02 | 0.62 | 0.0010 | 0.05 | 0.2950 |
| grafo100a-128 | 21850.80 | 0.18 | 0.05 | 2.43 | 0.0021 | 0.17 | 0.1180 |
| grafo250a-128 | 68858.90 | 3.85 | 0.41 | 7.89 | 0.0250 | 2.53 | 0.3900 |
| grafo500a-128 | 153809.50 | 108.38 | 1.90 | 52.65 | 0.0100 | 9.83 | 0.0060 |
| grafo750a-128 | 242610.80 | 770.96 | 5.03 | 150.09 | 0.0343 | 37.56 | 0.0010 |
| grafo1000a-128 | 336517.60 | 2734.91 | 9.06 | 330.12 | 0.0124 | 52.06 | 0.0001 |
| grafo50b-14 | 1092.54 | 0.03 | 0.02 | 0.64 | 0.0300 | 0.03 | 0.5640 |
| grafo100b-14 | 2298.66 | 0.09 | 0.08 | 2.65 | 0.0010 | 0.16 | 0.4630 |
| grafo250b-14 | 5447.14 | 0.72 | 1.35 | 8.56 | 0.0110 | 2.58 | 0.5850 |
| grafo500b-14 | 11198.03 | 4.93 | 12.37 | 52.65 | 0.0060 | 9.96 | 0.4740 |
| grafo750b-14 | 16524.70 | 17.17 | 47.03 | 110.09 | 0.0231 | 38.04 | 0.4350 |
| grafo1000b-14 | 22448.11 | 50.93 | 164.76 | 200.15 | 0.0260 | 53.78 | 0.4330 |
| grafo50b-128 | 1565.68 | 0.03 | 0.01 | 0.65 | 0.0100 | 0.04 | 0.4320 |
| grafo100b-128 | 3243.71 | 0.09 | 0.09 | 2.38 | 0.0020 | 0.16 | 0.3780 |
| grafo250b-128 | 8110.50 | 0.71 | 1.40 | 8.54 | 0.0200 | 2.57 | 0.5330 |
| grafo500b-128 | 16062.21 | 3.98 | 13.76 | 52.65 | 0.0140 | 9.94 | 0.4920 |
| grafo750b-128 | 23748.90 | 12.85 | 52.10 | 150.09 | 0.0190 | 37.96 | 0.4840 |
| grafo1000b-128 | 32271.93 | 31.71 | 204.95 | 223.12 | 0.0050 | 53.77 | 0.4950 |

It's straightforward to notice that CPLEX outperforms the execution time of CoinMP, confirming it's efficiency. The Lagrangean relaxations proposed provide good quality bounds, competitive with the LP relaxation of the problem. For the biggest instances, the execution times are comparable to the CPLEX one and in some cases, like the grafo1000b-128 set where the execution time are better than the CPLEX one. Obviously, the STD algorithm has better results than DIST, having a global knowledge of the network, using global parameters, like described in 7.4, but, for our purposes this approach is not applicable.

## 7.7 Shared Memory Algorithm

The shared memory algorithm proposed exploits the inherent parallelism inside the distributed subgradient. The problem granularity is straightforward: each node $h$ is an independent entity to consider and we can map a subset of nodes for each thread spawned in a parallel cycle. We used OpenMP because is a de-facto standard for the shared memory parallel programming model and for its portability. The simplicity of implementation and the non-invasive pre-processor calls to the APIs enable a fast code deploying, adding few lines of code to the serial version.

Each node has its own data structures to compute its $z_{LR}(\boldsymbol{\lambda})$ value and a private array $\boldsymbol{\lambda_{nodes}}$ for storing the penalties relative to the other nodes. The used processor for testing the algorithm, an Intel Core i7 @ 2.8 GHz, implements the *Hyper-Threading* [120] proprietary technology by Intel giving to a quad-core processor the ability to act like a processor with the double of cores (in our case 8 cores). In our test, we reported the best speed-up value spawning 8 threads, theoretically, one thread per core.

The load balancing is totally managed by OpenMP, that assigns to each thread an equal number of nodes, as shown in picture 7.7, relative to an instance with 1000 nodes.

As mentioned before, the insertion of the OpenMP directories is not invasive and the parallel algorithm has few modifications compared to the serial one.

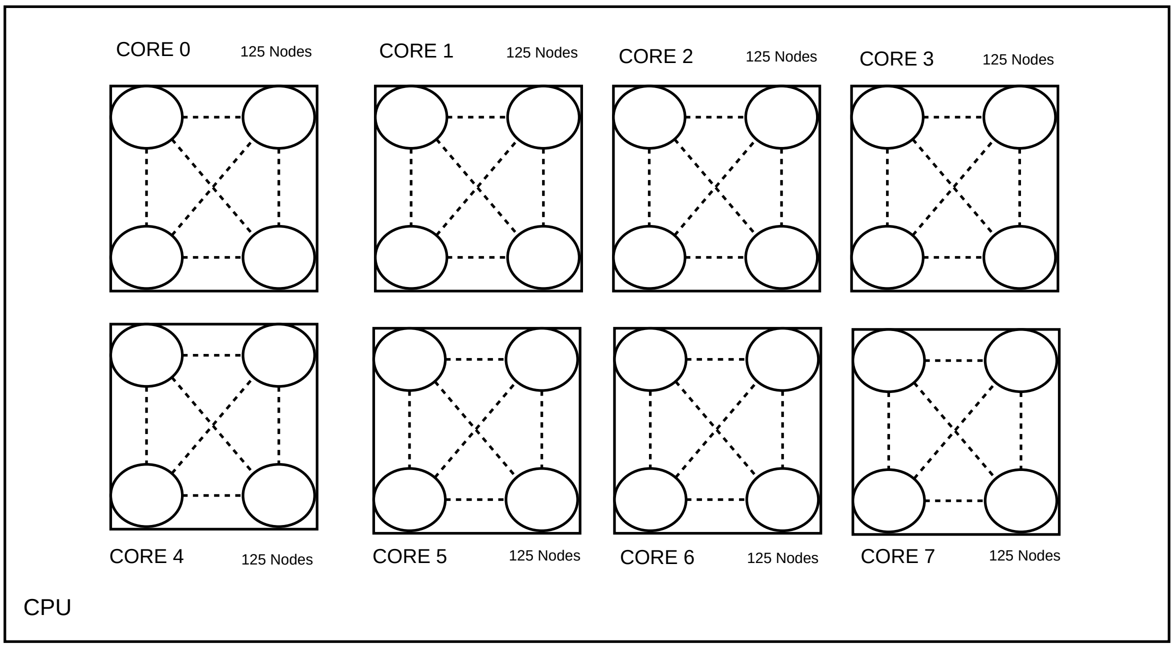


FIGURE 7.7: Nodes deploy in a eight core processor.

```
**Algorithm** *Inner-Subgradient*$(h, \boldsymbol{\lambda'})$
1. Initialize $\lambda_i = \lambda'_i$ for each $i \in V_h = \delta(h) \cup \{h\}$
2. **while** $it < InnerMaxIter$ **do**
3.     Solve $z^h_{LR}(\boldsymbol{\lambda})$
4.     Update only the node $h$ penalty: $\lambda_h = max\{0, \lambda_h + \alpha_h g_h\}$
5.     **if** $z^h_{LR}(\boldsymbol{\lambda}) < z^h_{LR}(\boldsymbol{\lambda'})$
6.       **then** $\lambda'_h = \lambda_h$
```

```
**Algorithm** *External-Optimization-OMP*()
1. **while** $et < ExtMaxIter$ **do**
2.     # parallel for private($h$, $\boldsymbol{\lambda_{node}}$)
3.     **for** $h = 1$ **to** $|V|$ **do**
4.         Inner-Subgradient($h$, $\boldsymbol{\lambda_{node}}$)
5.     # end parallel for
6.     **Update** each $\boldsymbol{\lambda_{node}}$ array with the new $\lambda_h$ penalties
```

The differences between the two algorithms are minimal: we added in line 2 the OpenMP directories, opening a parallel *for* (fork) and declaring private for each thread the $\boldsymbol{\lambda_{node}}$ and the $h$ variables. Each thread, once partitioned the indexes $h$, will compute the *Inner-Subgradient*.

We don't need explicit synchronization directives because the end of the parallel region is an implicit synchronization point for the threads.

At line 5 of the main procedure, closed the parallel region (join), the algorithm updates the new $\lambda_h$ penalties in the $\boldsymbol{\lambda_{node}}$ array of each node.

## 7.8 Distributed Memory Algorithm

In this case we used the hybridization of MPI with OpenMP described in 2.3.1 where the shared memory model is used to enhance the intra-node performances of the message passing model. The algorithm exploits another level of parallelism with the possibility to divide the graph among the cluster's node and inside each node, with OpenMP.

Each MPI *task*, deployed on a different cluster's node, performs the External Optimization step of a given subset of nodes $V_{task}$. Then, the $V_{task}$ set is computed in parallel, in the same fashion described in 7.7.

At the end of each inner cycle, once each task computed the $z^h_{LR}(\boldsymbol{\lambda})$ relative to its $V_{task}$ subset of nodes, we need a synchronization primitive (barrier) to synchronize the tasks and broadcast consistent penalties values.

The communication step is implemented with a *broadcast* communication primitive that updates the other tasks with the new Lagrangean penalties. The computational tests has been conducted on a experimental cluster implemented with Microsoft HPC 2008 Server with:

- 33 HP PCs with: Intel Pentium E6800 @ 3.2 GHz, 4 Gbyte of RAM,
- Windows 7 Professional Edition 64 bit, for each node,
- network switch: HP Procurve 2650.

The bottleneck relative to this cluster is the slow network that connects the machines (computation nodes). For our purposes, it's sufficient to show the scalability of the method in a message passing environment.

In the 7.10 section we will observe that over a certain number of compute nodes, the method doesn't scale anymore, and the execution times are slowed down by the network and the communications.

## 7.9 GPU Algorithm

In this section we propose a many-core algorithm for running the algorithm on a many-core platform. As in the other chapters, we used the Nvidia CUDA parallel programming model for its reliability and more effective programming and debugging tools.

For this algorithm we need to explore deeply the *Inner-Subgradient* procedure. It's necessary, indeed, to break into small pieces the steps of the Core Optimization and design four kernels for executing the method on a GPU.

First, we propose a extended version for the Core Optimization and, then, we will design the GPU algorithm.

```
Algorithm Inner-Subgradient-Extended($h, \boldsymbol{\lambda'}$)
1.  while $k < InnerMaxIter$ do
2.      if $z^h_{LR}(\boldsymbol{\lambda^k}) < z^h_{LR}(\boldsymbol{\lambda^{k-1}})$ // If the bound is improved
3.        then Compute subgradient: $g^k_h = \sum_{j\in\delta(h)} x^k_{hj} - b_h$
4.             Compute $\alpha^k$: $\alpha^k = \alpha^{k-1}\gamma$
5.             Save best $\lambda_h$ and $X^k_h$
6.             Update $\lambda^k_h$: $max\{0, \lambda^{k-1}_h + \alpha^k\}$
7.             Update $\boldsymbol{\lambda^k}$ with $\lambda^k_h$
8.             //Update X:
9.             for each $j \in \delta(h)$ do
10.                ($p_{hj} - \lambda^k_h - \lambda_j > 0$)?$x_{hj} = u_{hj}$ : $x_{hj} = 0$
11.       else Compute subgradient: $g^k_h = \sum_{j\in\delta(h)} x^k_{hj} - b_h$
12.            if $\Delta > \Delta_k$
13.              then $\alpha^k = max\{\alpha^{k-1}\gamma', \alpha_{min}\}$
14.            Update $\lambda^k_h$: $max\{0, \lambda^{k-1}_h + \alpha^k\}$
15.            Update $\boldsymbol{\lambda^k}$ with $\lambda^k_h$
16.            //Update X:
17.            for each $j \in \delta(h)$ do
```

```
18.                 $(p_{hj} - \lambda_h^k - \lambda_j > 0)?x_{hj} = u_{hj} : x_{hj} = 0$
19.             $\Delta++$
20.     $k++$
```

In line 2 the algorithm checks if the bound has been improved. If yes, the subgradient $g_h$ is computed (line 3), updated the step $\alpha$, line 4, saved the penalties and the solution (line 5). In lines 6-10, the algorithm updates the actual solution (lines 9-10) and the Lagrangean penalties. Otherwise, the $g_h$ subgradient is computed anyway together with the other parameters without saving the penalties and the solution. In lines 4 and 12-13 the $\alpha$ step update is computed following the *quasi-constant* rule described before.

The main idea behind the GPU algorithm is to execute in a concurrent fashion each $k$ iteration of each $h$ node in the problem. Following the CUDA programming model, we can assign to each computation block a node $h$ and compute in parallel each step of the *Inner-Subgradient* algorithm.

To minimize the communications between the HOST and the GPU, has been implemented three support kernels that manipulate on the device the data structures and manage the communications of the Lagrangean penalties.

```
Algorithm External-Optimization-GPU
1.  $BLKS = h$
2.  $THDS = n$
3.  $Shared_{mem} = THDS$
4.  while $et < ExtMaxIter$ do
5.      Update-X-Kernel<<<BLKS,THDS>>>($\boldsymbol{\lambda}'$, $x_{ij}$)
6.      while $k < InnerMaxIter$ do
7.          Compute-$z_{LR}$-Kernel<<<BLKS, THDS, $Shared_{mem}$>>>($z_{LR}(\boldsymbol{\lambda}')$,
            $x_{ij}$)
8.          Inner-Subgradient-Kernel<<<BLKS, THDS, $Shared_{mem}$>>>($\boldsymbol{\lambda}'$,
            $z_{LR}(\boldsymbol{\lambda}')$, $x_{ij}$)
9.      Update-Lambda-Kernel<<<BLKS,THDS>>>($\boldsymbol{\lambda}$, $\boldsymbol{\lambda}'$)
```

The peculiarity of this algorithm is the design's shift from executing the relaxation of a certain number of nodes in parallel to executing the same relaxation's step for all the nodes in parallel. The *Update-X-Kernel* (line 5), updates the $x_{ij}$ solution for the Lagrangean problem in parallel for each node (the number of blocks is the same in each kernel) at the beginning of each Core Optimization step. The *Compute-$z_{LR}$-Kernel* (line 7), evaluates, for each node $h$ its actual objective function value. The *Update-Lambda-Kernel* (line 9), finally, at the end of the Core Optimization, send the updated penalties to each node.

**Algorithm** *Inner-Subgradient-Kernel*$(\boldsymbol{\lambda'}, z_{LR}(\boldsymbol{\lambda'}), x_{ij})$

```
1.  $h = blockID_x$
2.  $th_{idx} = threadID_x$
3.  $THDS = blockDIM_x$
4.  // Shared Memory Initialization
5.  shared[$th_{idx}$] = 0
6.  times = $|V| / THDS$
7.  reminder = $|V| \% THDS$
8.  **if** $th_{idx} <$ reminder
9.     **then** times++
10. Thread-Synchronization()
11. **if** $z^h_{LR}(\boldsymbol{\lambda^k}) < z^h_{LR}(\boldsymbol{\lambda^{k-1}})$
12.    **then**
13.           // **Compute** subgradient: $g^k_h = \sum_{j \in \delta(h)} x^k_{hj} - b_h$
14.           **for** $t = 0$ **to** times **do**
15.                index = $h * V + (th_{idx} + t * THDS)$
16.                shared[$th_{idx}$] += $x_{ij}$[index]
17.           Thread-Synchronization()
18.           // Reduction
19.           **for** $s = THDS/2$ **to** $0, s/ = 2$ **do**
20.                **if** $th_{idx} < s$
21.                   **then** shared[$th_{idx}$] += shared[$th_{idx} + s$]
22.                Thread-Synchronization()
23.           $g^k_h$ = shared[0]
24.           // **Compute** $\alpha^k$: $\alpha^k = \alpha^{k-1}\gamma$
25.           // **Save** best $\lambda_h$ and $X^k_h$
26.           // **Update** $\lambda^k_h$
27.           $\lambda[h] = max\{0, \lambda^{k-1}_h + \alpha^k\}$
28.           // **Update** $\boldsymbol{\lambda^k}$ with $\lambda^k_h$
29.           //Update X:
```

```
30.             for $t = 0$ to times do
31.                 index $= h * V + (th_{idx} + t * THDS)$
32.                 $(p[index] - \lambda[h] - \lambda[(th_{idx} + t * THDS)] > 0)?x[index] = u_{hj}$ :
                    $x[index] = 0$
33.       else  // Compute subgradient: $g_h^k = \sum_{j \in \delta(h)} x_{hj}^k - b_h$
34.             for $t = 0$ to times do
35.                 index $= h * V + (th_{idx} + t * THDS)$
36.                 shared$[th_{idx}]$ += $x_{ij}$[index]
37.             Thread-Synchronization()
38.             // Reduction
39.             for $s = THDS/2$ to $0, s/ = 2$ do
40.                 if $th_{idx} < s$
41.                   then shared$[th_{idx}]$ += shared$[th_{idx} + s]$
42.                 Thread-Synchronization()
43.             $g_h^k$ = shared[0]
44.             if $\Delta > \Delta_k$
45.               then $\alpha^k = max\{\alpha^{k-1}\gamma', \alpha_{min}\}$
46.             // Update $\lambda_h^k$
47.             $\lambda[h] = max\{0, \lambda_h^{k-1} + \alpha^k\}$
48.             // Update $\boldsymbol{\lambda^k}$ with $\lambda_h^k$
49.             //Update X:
50.             for $t = 0$ to times do
51.                 index $= h * V + (th_{idx} + t * THDS)$
52.                 $(p[index] - \lambda[h] - \lambda[(th_{idx} + t * THDS)] > 0)?x[index] = u_{hj}$ :
                    $x[index] = 0$
53.             $\Delta + +$
```

The algorithm spawns a block for each $h$ node. In lines 14-23 and 34-43 the subgradient is computed using a modified version of the parallel reduction suggested by [59]. In the case the bound is improved, the new $\alpha$ is computed (line 24), the node's Lagrangean penalty updated (line 27) and created the new solution for the $h$ node, lines 30-32. Otherwise, in the *else* branch of the *if-then-else* statement, starting at line 33, the instructions are the same, the only difference is in the penalty update.

The data structures are indexed in a row-major fashion. The *index* value, in lines 15, 31, 35 and 51 is the one-dimensional address of the structures accessed by the $th_{idx}$ thread.

The notable performances of this method is given by the use of parallel reductions that seems to fit very well in the cluster processors of the GPU, together with the use of the shared memory. The Kepler architecture, with 192 Cuda Cores per cluster processor of our test device, seems to manage very well the reductions, due to the great amount of cores in the same cluster (preliminary test done on a Fermi GPU, with cluster of 32 Cuda Cores, showed that the reductions step was the bottleneck for the method).

## 7.10 Computational Results

We present the computational results relative to the Speed-Up factors obtained on the different parallel platforms considered. The testbed machine used for the OpenMP and the CUDA algorithms is the same used in the paragraph 7.6 with an Nvidia GeForce GTX 770 with 2 GigaBytes of GDDR5 memory. The cluster used for the MPI algorithm is the one described in 7.8.

The tuning parameters are the same used in 7.6

We report the *SpeedUp* factor, as the ratio between the serial time of the algorithms and the parallel ones, $SpeedUp = Time_{serial}/Time_{parallel}$.

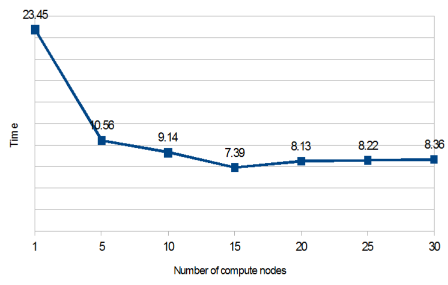
Table 7.2: Speed-Ups of the CUDA, MPI and OpenMP algorithms.

| **Problem Group** | **Serial (AVG)** (sec.) | **Parallel (AVG)** (sec.) | | | **SpeedUp** | | |
|---|---|---|---|---|---|---|---|
| Name | Serial Time | OpenMP | MPI | CUDA | OpenMP | MPI | CUDA |
| grafo50a-14 | 0.041 | 0.024 | 0.650 | 0.023 | 1.708 X | 0.063 X | 1.754 X |
| grafo100a-14 | 0.144 | 0.055 | 0.829 | 0.024 | 2.618 X | 0.174 X | 5.937 X |
| grafo250a-14 | 2.550 | 0.652 | 1.522 | 0.171 | 3.911 X | 1.675 X | 14.873 X |
| grafo500a-14 | 9.828 | 2.445 | 1.993 | 0.415 | 4.020 X | 4.931 X | 23.697 X |
| grafo750a-14 | 37.509 | 9.370 | 5.747 | 1.343 | 4.003 X | 6.527 X | 27.933 X |
| grafo1000a-14 | 52.015 | 13.378 | 6.390 | 1.799 | 3.888 X | 8.140 X | 28.919 X |
| grafo50a-128 | 0.050 | 0.025 | 0.642 | 0.027 | 2.000 X | 0.078 X | 1.864 X |
| grafo100a-128 | 0.171 | 0.058 | 0.837 | 0.029 | 2.948 X | 0.204 X | 5.964 X |
| grafo250a-128 | 2.537 | 0.665 | 1.544 | 0.171 | 3.815 X | 1.643 X | 14.805 X |
| grafo500a-128 | 9.837 | 2.098 | 1.989 | 0.421 | 4.689 X | 4.946 X | 23.350 X |
| grafo750a-128 | 37.566 | 9.281 | 6.729 | 1.354 | 4.048 X | 5.583 X | 27.754 X |
| grafo1000a-128 | 52.062 | 13.467 | 7.400 | 1.798 | 3.866 X | 7.035 X | 28.953 X |
| grafo50b-14 | 0.03 | 0.022 | 0.643 | 0.025 | 1.655 X | 0.057 X | 1.456 X |
| grafo100b-14 | 0.16 | 0.049 | 0.889 | 0.028 | 3.388 X | 0.187 X | 6.022 X |
| grafo250b-14 | 2.58 | 0.654 | 1.535 | 0.172 | 3.948 X | 1.682 X | 14.986 X |
| grafo500b-14 | 9.96 | 2.136 | 1.998 | 0.426 | 4.667 X | 4.989 X | 23.384 X |
| grafo750b-14 | 38.04 | 9.335 | 6.792 | 1.365 | 4.075 X | 5.601 X | 27.860 X |
| grafo1000b-14 | 53.78 | 13.387 | 7.400 | 1.823 | 4.018 X | 7.268 X | 29.502 X |
| grafo50b-128 | 0.041 | 0.023 | 0.680 | 0.027 | 1.783 X | 0.060 X | 1.493 X |
| grafo100b-128 | 0.164 | 0.055 | 0.899 | 0.029 | 2.971 X | 0.182 X | 5.752 X |
| grafo250b-128 | 2.578 | 0.641 | 1.531 | 0.174 | 4.022 X | 1.684 X | 14.799 X |
| grafo500b-128 | 9.947 | 2.340 | 1.873 | 0.429 | 4.251 X | 5.311 X | 23.202 X |
| grafo750b-128 | 37.960 | 9.381 | 6.834 | 1.362 | 4.046 X | 5.555 X | 27.863 X |
| grafo1000b-128 | 53.778 | 13.729 | 7.123 | 1.816 | 3.917 X | 7.550 X | 29.615 X |

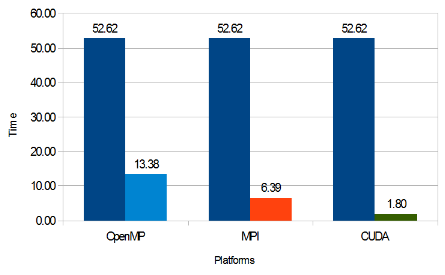
The experimental results show clearly that the GPU algorithm obtains the best Speed-Up factor among the proposed solutions, lowering the execution time more than one order of magnitude. The effectiveness of this method is a consequence of the massively parallel architecture implemented in the GPU, allowing us to execute in a parallel fashion a large number of operations. The other two solutions, indeed, obtained good results, proving the intrinsic parallel nature of the distributed subgradient.

For what concerns the MPI algorithm, we are aware that the used cluster is not the best option to test our method. We are sure that, executing the algorithm on faster (mainly on the communication side) infrastructure, the execution times can be lower. We used only 15 compute nodes, because, as shown in figure 7.8a, a larger number of compute nodes brings to a degradation of the performance, due to communications.

The OpenMP algorithm has the quality that is almost the same with respect to the serial version, due to the intelligent design of the OpenMP's directives, allowing to achieve good results modifying few parts of the original code.



(A) MPI Performance degradation for grafo1000a-128



(B) Platforms comparison

FIGURE 7.8: Performances evaluation.

## 7.11 Considerations and Future Work

In this chapter we presented three parallel algorithms for finding a bound to the Membership Overlay Problem, relative to the Peer-2-Peer network model. The results reported show the efficiency of the GPU algorithm, executed on a mid-level consumer device. The others algorithm, indeed, showed good results, proving the method's high scalability. We planned to apply this new kind of relaxation to other

Combinatorial Optimization problems, to verify the reliability and the generality of the method.

# Chapter 8

# Conclusions

In this Thesis we presented new optimization algorithms designed to run on the state of the art, inherently parallel processors available on the market. The need to exploit these new architectures in Combinatorial Optimization is becoming ever more urgent. The advent of technologies and paradigms like *cloud computing* or *big data* for example, containing at their core notable CO problems, together with the increasing dimension of real-life instances (vehicle routing, resource scheduling and optimization, network design, etc...) requires consistent and reliable developments of both theoretical and practical issues. Moreover, the computation of solutions for these problems must be really fast in most cases, from decision support softwares to scientific applications. This work has provided parallel methodologies for solving or enhancing the solution methods of some important problems in literature, methodologies that can be applied to a wide spectrum of real-word situations.

The speed-up factors obtained are suggestive of the potential behind these architectures. Not only the hardware parallelism is growing, but also the quantity of memory got far beyond the 8/16 gigabytes of RAM for each computation node or workstation, and 2/4 gigabytes of memory for accelerators. To exploit in a better way the computational resources of these devices, also at the dawn of *exa-scale* era, can bring an effective enhancement in both industrial and academic fields.

Our envisioned future work includes the porting of these algorithms on a larger number of hardware platforms, like AMD devices or FPGA, by exploiting the OpenCL parallel programming model to take advantage also of the peculiar characteristics of these devices (faster computation, better double precision throughput, larger number of cores, etc ...). Another interesting development related to the processors evolution is the insertion of a many-core processor (GPU) in the same silicon die of a canonical multi-core CPU. This solution is called APU, Accelerated Processing Unit. HSA, Heterogeneous System Architecture, proposed by AMD is a new environment designed to specifically exploit these processors. The great advantage of this approach is avoiding the PCI Express communications between the HOST and the GPU, thus exploiting the whole system memory. In this technological period, is mandatory to take into account the massively parallel implementation of these processors during the design of new methodologies and algorithms. Our aim is to design, starting from the well established CO's theoretical foundations, new algorithms targeted to run on these new devices.

# Bibliography

[1] Francesco Strappaveccia. A parallel algorithm for a distributed lagrangean relaxation for the membership overlay problem, 2011. Master Thesis.

[2] Wikipedia. Supercomputer — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Supercomputer`. [Online; accessed 23-January-2015].

[3] Allan R Director-Hoffman. *Supercomputers: directions in technology and applications*. National Academy Press, 1989.

[4] Top 500. http://www.green500.org/, 2015.

[5] Top 500. http://www.top500.org/, 2014.

[6] Jerry Banks et al. *Handbook of simulation*. Wiley Online Library, 1998.

[7] Wikipedia. Simulation — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Simulation`. [Online; accessed 23-January-2015].

[8] Wikipedia. Computational fluid dynamics — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Computational_fluid_dynamics`. [Online; accessed 23-January-2015].

[9] Wikipedia. Computational chemistry — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Computational_chemistry`. [Online; accessed 23-January-2015].

[10] Wikipedia. Multi-agent systems — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Multi-agent_system`. [Online; accessed 23-January-2015].

[11] Wikipedia. Computational astrophysics — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Computational_astrophysics`. [Online; accessed 23-January-2015].

[12] Wikipedia. Computational physics — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Computational_physics`. [Online; accessed 23-January-2015].

[13] Berk Hess, Carsten Kutzner, David Van Der Spoel, and Erik Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3):435–447, 2008.

[14] Wikipedia. Bioinformatics — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Bioinformatics`. [Online; accessed 23-January-2015].

[15] Michael Friendly and Daniel J Denis. Milestones in the history of thematic cartography, statistical graphics, and data visualization. *Seeing Science: Today American Association for the Advancement of Science*, 2008.

[16] Wikipedia. Data acquisition — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Data_acquisition`. [Online; accessed 23-January-2015].

[17] Wikipedia. Data analysis — Wikipedia, the free encyclopedia, 2015. URL `http://en.wikipedia.org/wiki/Data_analysis`. [Online; accessed 23-January-2015].

[18] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[19] Gregory Piateski and William Frawley. *Knowledge discovery in databases*. MIT press, 1991.

[20] David J Hand, Heikki Mannila, and Padhraic Smyth. *Principles of data mining*. MIT press, 2001.

[21] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

[22] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[23] Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. Openfoam: A c++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, volume 1000, pages 1–20, 2007.

[24] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, et al. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, 2009.

[25] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[26] Sayantan Sur, Matthew J Koop, and Dhabaleswar K Panda. High-performance and scalable mpi over infiniband with reduced memory usage: an in-depth performance analysis. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 105. ACM, 2006.

[27] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Wiley Plus/Blackboard Stand-alone to accompany Operating Systems Concepts with Java (Wiley Plus Products)*. John Wiley & Sons, 2006.

[28] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, may 2008. URL `http://www.openmp.org/mp-documents/spec30.pdf`.

[29] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[30] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[31] Nvidia Developer Zone. https://developer.nvidia.com/, 2014.

[32] Khronos Group. Opencl, august 2009. URL `https://www.khronos.org/opencl/`.

[33] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982. ISBN 0-13-152462-3.

[34] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[35] George B Dantzig. *Linear programming and extensions*. Princeton university press, 1998.

[36] George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.

[37] Ilog CLEX. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/, 2014.

[38] Robin Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[39] Boris Teodorovich Polyak. The conjugate gradient method in extremal problems. *USSR Computational Mathematics and Mathematical Physics*, 9(4): 94–112, 1969.

[40] Michael Held, Philip Wolfe, and Harlan P Crowder. Validation of subgradient optimization. *Mathematical programming*, 6(1):62–88, 1974.

[41] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[42] Michael Held and Richard M Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical programming*, 1(1):6–25, 1971.

[43] Naum Zuselevich Shor, Krzysztof Kiwiel, and Andrzej Ruszcaynski. *Minimization methods for non-differentiable functions*. Springer-Verlag New York, Inc., 1985.

[44] George L Nemhauser and Laurence A Wolsey. *Integer and combinatorial optimization*, volume 18. Wiley New York, 1988.

[45] Karla Hoffman and Manfred Padberg. Lp-based combinatorial problem solving. *Annals of Operations Research*, 4(1):145–194, 1985.

[46] Richard Bellman. Dynamic programming. *Princeton University Press*, 1957.

[47] Ailsa H Land and Alison G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[48] G. Wäscher, H. Haussner, and H. Schumann. An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109—1130, 2007.

[49] P. Gilmore and R. Gomory. Multistage cutting problems of two and more dimensions. *Operations Research*, 13:94–119, 1965.

[50] P. Gilmore and R. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1074, 1966.

[51] J. C. Herz. Recursive computational procedure for two-dimensional stock cutting. *IBM Journal of Research and Development*, 16(5):462–469, September 1972. ISSN 0018-8646. doi: 10.1147/rd.165.0462. URL `http://dx.doi.org/10.1147/rd.165.0462`.

[52] N. Christofides and C. Whitlock. An algorithm for two-dimensional cutting problems. *Operations Research*, 25:30–44, 1977.

[53] J. E. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *Journal of the Operational Research Society*, 36:297–306, 1985.

[54] G.F. Cintra, F.K. Miyazawa, Y. Wakabayashi, and E.C. Xavier. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191(1):61 – 85, 2008. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/j.ejor.2007.08.007. URL `http://www.sciencedirect.com/science/article/pii/S0377221707008831`.

[55] Mauro Russo, Antonio Sforza, and Claudio Sterle. An improvement of the knapsack function based algorithm of gilmore and gomory for the unconstrained two-dimensional guillotine cutting problem. *International Journal of Production Economics*, 145(2):451 – 462, 2013.

ISSN 0925-5273. doi: http://dx.doi.org/10.1016/j.ijpe.2013.04.031. URL `http://www.sciencedirect.com/science/article/pii/S0925527313001953`.

[56] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. ISBN 978-3-905674-09-5. URL `http://dl.acm.org/citation.cfm?id=1413957.1413966`.

[57] Ben D. Lund and Justin W. Smith. A multi-stage CUDA kernel for Floyd-Warshall. *CoRR*, abs/1001.4108, 2010. URL `http://arxiv.org/abs/1001.4108`.

[58] Vincent Boyer, Didier El Baz, and Moussa Elkihel. Solving knapsack problems on GPU. *Computers & Operations Research*, 39(1):42–47, 2012.

[59] Mark Harris. Optimizing parallel reduction in cuda. *Nvidia developer zone*, 2009. URL `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`.

[60] Ramser J. Dantzig G. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959.

[61] Paolo Toth and Daniele Vigo, editors. *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001. ISBN 0-89871-498-2.

[62] Wasil Edward A. Golden Bruce L., Raghavan S., editor. *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*. Springer, 2008. ISBN 978-0-387-77778-8.

[63] VeroLog Web Site. http://www.verolog.eu/, 2014.

[64] Marco Boschetti and Vittorio Maniezzo. A set covering based matheuristic for a real-world city logistics problem. *International Transactions in Operational Research*, pages n/a–n/a, 2014. ISSN 1475-3995. doi: 10.1111/itor.12110. URL `http://dx.doi.org/10.1111/itor.12110`.

[65] Wright J.R. Clarke G. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[66] Jaikumar R. Fisher M. L. A generalized assignment heuristic for vehicle routing. *Networks*, 11:109–124, 1981.

[67] Paolo Toth and Daniele Vigo. The granular tabu search and its application to the vehicle-routing problem. *INFORMS J. on Computing*, 15(4):333–346, 2003. ISSN 1526-5528.

[68] Wen-Chyuan Chiang and RobertA. Russell. Simulated annealing metaheuristics for the vehicle routing problem with time windows. *Annals of Operations Research*, 63(1):3–27, 1996. ISSN 0254-5330.

[69] Luca Maria Gambardella, Éric Taillard, and Giovanni Agazzi. Macs-vrptw: A multiple colony system for vehicle routing problems with time windows. In *New Ideas in Optimization*, pages 63–76. McGraw-Hill, 1999.

[70] Christian Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *COMPUTERS AND OPERATIONS RESEARCH*, 31: 2004, 2001.

[71] Jari Kytöjoki, Teemu Nuortio, Olli Bräysy, and Michel Gendreau. An efficient variable neighborhood search heuristic for very large scale vehicle routing problems. *Computers & OR*, 34(9):2743–2757, 2007.

[72] Yannis Marinakis and Magdalene Marinaki. A hybrid particle swarm optimization algorithm for the open vehicle routing problem. In Marco Dorigo, Mauro Birattari, Christian Blum, AndersLyhne Christensen, AndriesP. Engelbrecht, Roderich Groß, and Thomas Stützle, editors, *Swarm Intelligence*, volume 7461 of *Lecture Notes in Computer Science*, pages 180–187. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32649-3.

[73] Wei Ou and Bao-Gang Sun. A dynamic programming algorithm for vehicle routing problems. In *Proceedings of the 2010 International Conference on Computational and Information Sciences*, ICCIS '10, pages 733–736, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4270-6.

[74] P. Toth Christofides N., A. Mingozzi. Exact algorithms for the vehicle routing problem based on spanning tree and shortest path relaxation. *Math. Programming*, 10:255–280, 1981.

[75] Jens Lysgaard, Adam N. Letchford, and Richard W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming A*, 100(2):423–445, 2004.

[76] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342–354, 1992.

[77] R. Roberti Baldacci R., A. Mingozzi. New route relaxation and pricing strategies for the vehicle routing problem. *Operation Research*, 59:1269–1283, 2011.

[78] Rafael Martinelli, Diego Pecin, and Marcus Poggi. Efficient elementary and restricted non-elementary route pricing. *European Journal of Operational Research*, 239(1):102–111, 2014.

[79] AndréR. Brodtkorb, TrondR. Hagen, Christian Schulz, and Geir Hasle. Gpu computing in discrete optimization. part i: Introduction to the gpu. *EURO Journal on Transportation and Logistics*, 2(1-2):129–157, 2013. ISSN 2192-4376.

[80] Marco A. Boschetti, Vittorio Maniezzo, and Francesco Strappaveccia. Using gpu computing for solving the two-dimensional guillotine cutting problem. Submitted for publication, 2014.

[81] M. L. Balinski and R. E. Quandt. On an integer program for a delivery problem. *Operations Research*, 12(2):pp. 300–304, 1964. ISSN 0030364X.

[82] Toth P. Christofides N., Mingozzi A. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981.

[83] Salani M. Righini G. Symmetry helps: bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization*, 3(3):255–273, 2006.

[84] Salani M. Righini G. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. *Computers and operations research*, 36(4):1191–1203, 2009.

[85] Salani M. Righini G. New dynamic programming algorithms for the resource constrained elementary shortest path. *Networks*, 51(3):155–170, 2008.

[86] Pawan Harish, Vibhav Vineet, and PJ Narayanan. Large graph algorithms for massively multithreaded architectures. *Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74*, 2009.

[87] Aydın Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Computing*, 36(5):241–253, 2010.

[88] Hector Ortega-Arranz, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano. A new gpu-based approach to the shortest path problem. In *High performance computing and simulation (HPCS), 2013 international Conference on*, pages 505–511. IEEE, 2013.

[89] Sumit Kumar, Alok Misra, and Raghvendra Singh Tomar. A modified parallel approach to single source shortest path problem for massively dense graphs using cuda. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, pages 635–639. IEEE, 2011.

[90] Christian Schulz. Efficient local search on the gpu—investigations on the vehicle routing problem. *Journal of Parallel and Distributed Computing*, 73 (1):14–31, 2013.

[91] SINTEF. http://www.sintef.no/, 2014.

[92] VRPLIB. http://www.or.deis.unibo.it/index.html, 2014.

[93] Feiyue Li, Bruce Golden, and Edward Wasil. Very large-scale vehicle routing: new test problems, algorithms, and results. *Computers & Operations Research*, 32(5):1165–1179, 2005.

[94] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[95] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[96] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[97] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[98] Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[99] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951.

[100] Brian C Dean. *Continuous-time dynamic shortest path algorithms*. PhD thesis, Massachusetts Institute of Technology, 1999.

[101] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.

[102] Ulrich Lauther. Slow preprocessing of graphs for extremely fast shortest path calculations. In *Lecture at the Workshop on Computational Integer Programming at ZIB*, volume 10, page 11, 1997.

[103] Ulrich Lauther. An extremely fast, exact algorithm for finding shor test paths in static networks with geographical background. *Geoinformation und Mobilitat - von der Forschung zur praktischen Anwendung*, 22:219–230, 2004.

[104] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[105] Andrew V Goldberg and Renato Fonseca F Werneck. Computing point-to-point shortest paths from external memory. In *ALENEX/ANALCO*, pages 26–40, 2005.

[106] Thomas Pajor. *Multi-modal Route Planning*. PhD thesis, Universitat Karlsruhe Institut fur theoretische Informatik, 2009.

[107] Nvidia. Nvidia developer zone, 2007. https://developer.nvidia.com/, Accessed: 2013-12-03.

[108] Dhirendra Pratap Singh and Nilay Khare. A study of different parallel implementations of single source shortest path algorithms. *International Journal of Computer Applications*, 54(10):26–30, 2012.

[109] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science 1998*, pages 722–731. Springer, 1998.

[110] Joseph R Crobak, Jonathan W Berry, Kamesh Madduri, and David A Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[111] Marios Papaefthymiou and Joseph Rodrigue. Implementing parallel shortest-paths algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 30:59–68, 1997.

[112] Pedro J Martín, Roberto Torres, and Antonio Gavilanes. Cuda solutions for the sssp problem. In *Computational Science–ICCS 2009*, pages 904–913. Springer, 2009.

[113] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[114] Steve Coast. Open street map, 2004. http://www.openstreetmap.org/.

[115] Marco A Boschetti, Vittorio Maniezzo, and Matteo Roffilli. A fully distributed lagrangean solution for a peer-to-peer overlay network design problem. *INFORMS Journal on Computing*, 23(1):90–104, 2011.

[116] Mauro Birattari, Luis Paquete, Thomas Strutzle, and Klaus Varrentrapp. Classification of metaheuristics and design of experiments for the analysis of components tech. rep. aida-01-05. 2001.

[117] Ronald L Rardin and Reha Uzsoy. Experimental evaluation of heuristic optimization algorithms: A tutorial. *Journal of Heuristics*, 7(3):261–304, 2001.

[118] Stefan Saroiu, P Krishna Gummadi, and Steven D Gribble. Measurement study of peer-to-peer file sharing systems. In *Electronic Imaging 2002*, pages 156–170. International Society for Optics and Photonics, 2001.

[119] Boris Teodorovich Polyak. Minimization of unsmooth functionals. *USSR Computational Mathematics and Mathematical Physics*, 9(3):14–29, 1969.

[120] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.