Parallel heterogeneous Branch and Bound algorithms for multi-core and multi-GPU environments

Imen Chakroun

Ecole Doctorale Sciences Pour l'Ingénieur Université Lille 1 Nord-de-France
Laboratoire d'Informatique Fondamentale de Lille (UMR CNRS 8022)
Centre de Recherche INRIA Lille Nord Europe

Thèse présentée pour obtenir le grade de docteur
Discipline : Informatique

Parallel heterogeneous Branch and Bound algorithms for multi-core and multi-GPU environments

Défendue par :

Imen Chakroun

Octobre 2010 - Juin 2013

Devant le jury composé de:

Rapporteur :	Pierre Manneback, Professeur, Université de Mons, Belgique
Rapporteur :	Catherine Roucairol, Professeur, Université de Versailles
Examinateur :	Pierre Boulet, Professeur, Université Lille 1 Sciences et Technologies
Examinateur :	Stéphane Genaud, MCF HDR, ENSIIE Université de Strasbourg
Directeur de thèse :	Nouredine Melab, Professeur, Université Lille 1 Sciences et Technologies

Numéro d'ordre : 41136 | Année : 2013

Abstract:

Branch and Bound (B&B) algorithms are attractive for solving to optimality combinatorial optimization problems (COPs) by exploring a tree-based search space. Nevertheless, they are highly time-intensive when dealing with large problem instances (e.g. Taillard's FSP benchmarks) even using grid computing [Mezmaz et al., IEEE IPDPS'2007]. Massively parallel computing supplied through today's heterogeneous (GPU-enhanced multi-core) platforms [TOP500] is required to tackle more efficiently those instances. The challenge is therefore to exploit all the underlying levels of parallelism and thus to rethink accordingly the parallel models of B&B. In this thesis, we revisit the design and implementation of B&B for solving large COPs on (large) multi-core and multi-GPU platforms. The Flow-Shop scheduling problem (FSP) is considered as a case study.

A preliminary experimental study on some large FSP instances has revealed that the search tree is highly irregular (in shape and size) and very large (billions of billions of nodes), and the bounding operator is time-exorbitant (about 97% of B&B). Therefore, our first contribution is to propose a (single CPU core) GPU-accelerated approach (GB&B) in which only the bounding operator is performed on the GPU device. The approach deals with two issues: thread divergence [Chakroun et al., Concurrency and Computation: Practice and Experience 2012] and device hierarchical memory optimization [Melab et al., IEEE Cluster 2012]. Compared to a single CPU core-based implementation, speed-ups up to (×100) are obtained on Nvidia Tesla C2050. Although these good speed-ups, the performance analysis has shown that the overhead induced by the data transfer between CPU and GPU is high. Therefore, the aim of the second contribution [Chakroun et al., ICCS 2013] is to extend the approach (LL-GB&B) in order to minimize the CPU-GPU communication latency. Such objective is achieved through a GPU-based fine-grained parallelization of the branching and pruning operators in addition to the bounding one. The major and particularly challenging issue addressed here is thread divergence due to the strongly irregular nature of the explored tree mentioned above. Compared to a single CPU-based execution, LL-GB&B allows accelerations up to (×160) for large problem instances.

The third contribution [Chakroun et al., Journal of Parallel and Distributed Computing, 2013] consists in investigating the combination of GPU with multi-core processing. Two scenarios have been explored leading to two approaches: a concurrent (RLL-GB&B) and a cooperative one (PLL-GB&B). In the first one, the exploration process is performed concurrently by the GPU and the CPU cores. In the cooperative approach, the CPU cores prepare and off-load to GPU pools of subproblems using data streaming while the GPU performs the exploration. When combining multi-core and GPU, we figure out that using RLL-GB&B is not beneficial while PLL-GB&B enables an improvement up to (36%) compared to LL-GB&B. Recently computational grids such as Grid5000 (on some sites) have been enhanced with GPU accelerators, therefore the fourth contribution of this thesis is to address the combination of GPU and multi-core computing with large scale distributed computing. To do that, the different revisited algorithms have been put together in a heterogeneous meta-algorithm which automatically selects the one to be deployed according to the target hardware configuration. The meta-algorithm is coupled with the B&B@Grid approach proposed in [Mezmaz et al., IEEE IPDPS'2007]. B&B@Grid distributes the work units (search subspaces coded by intervals) among the grid nodes while the meta-algorithm selects and applies locally a parallel B&B algorithm on the received intervals. The combined approach allowed us to solve to optimality and efficiently some Taillard's FSP instances (20 jobs on 20 machines).

Keywords:

Parallel Branch-and-Bound, Heterogeneous computing, Graphics processing units, Multi-core computing, Grid'5000, Flowshop Scheduling Problem, Combinatorial Optimization, Exact Methods.

Résumé:

Les algorithmes Branch and Bound (B&B) sont attractifs pour la résolution exacte de problèmes d'optimisation combinatoire (POC) par exploration d'un espace de recherche arborescent. Néanmoins, ces algorithmes sont très gourmands en temps de calcul pour des instances de problèmes de grande taille (exemple : benchmarks de Taillard pour FSP) même en utilisant le calcul sur grilles informatiques [Mezmaz et al., IEEE IPDPS'2007]. Le calcul massivement parallèle fourni à travers les plates-formes de calcul hétérogènes d'aujourd'hui [TOP500] est requis pour traiter efficacement de telles instances. Le défi est alors d'exploiter tous les niveaux de parallélisme sous-jacents et donc de repenser en conséquence les modèles parallèles des algorithmes B&B. Dans cette thèse, nous nous attachons à revisiter la conception et l'implémentation des ces algorithmes pour la résolution de POC de grande taille sur (larges) plates-formes de calcul multi-coeurs et multi-GPUs. Le problème d'ordonnancement Flow-Shop (FSP) est considéré comme étude de cas.

Une étude expérimentale préliminaire sur quelques grandes instances du FSP a révélé que l'arbre de recherche est hautement irrégulier (en forme et en taille) et très large (milliards de milliards de noeuds), et que l'opérateur d'évaluation des bornes est exorbitant en temps de calcul (environ 97% du temps de B&B). Par conséquent, notre première contribution est de proposer une approche GPU avec un seul coeur CPU (GB&B) dans laquelle seul l'opérateur d'évaluation est exécuté sur GPU. L'approche traite deux défis: la divergence de threads [Chakroun et al., Concurrency and Computation: Practice and Experience 2012] et l'optimisation de la gestion de la mémoire hiérarchique du GPU [Melab et al., IEEE Cluster 2012]. Comparée à une version séquentielle, des accélérations allant jusqu'à (×100) sont obtenues sur Nvidia Tesla C2050. L'analyse des performances de GB&B a montré que le surcoût induit par le transfert des données entre le CPU et le GPU est élevé. Par conséquent, l'objectif de la deuxième contribution [Chakroun et al., ICCS 2013] est d'étendre l'approche (LL-GB&B) afin de minimiser la latence de communication CPU-GPU. Cet objectif est réalisé grâce à une parallélisation à grain fin sur GPU des opérateurs de séparation et d'élagage. Le défi majeur relevé ici est la divergence de threads qui est due à la nature fortement irrégulière citée ci-dessus de l'arbre exploré. Comparée à une exécution séquentielle, LL-GB&B permet d'atteindre des accélérations allant jusqu'à (×160) pour les plus grandes instances.

La troisième contribution [Chakroun et al., Journal of Parallel and Distributed Computing, 2013] consiste à étudier l'utilisation combinée des GPUs avec les processeurs multi-coeurs. Deux scénarios ont été explorés conduisant à deux approches: une concurrente (RLL-GB&B) et une coopérative (PLL-GB&B). Dans le premier cas, le processus d'exploration est effectué simultanément par le GPU et les coeurs du CPU. Dans l'approche coopérative, les coeurs du CPU préparent et transfèrent les sous-problèmes en utilisant le "streaming CUDA" tandis que le GPU effectue l'exploration. L'utilisation combinée du multi-coeur et du GPU a montré que l'utilisation de RLL-GB&B n'est pas bénéfique et que PLL-GB&B permet une amélioration allant jusqu'à (36%) par rapport à LL-GB&B. Sachant que récemment des grilles de calcul comme Grid5000 (certains sites) ont été équipées avec des GPU, la quatrième contribution de cette thèse traite de la combinaison du calcul sur GPU et multi-coeur avec le calcul distribué à grande échelle. Pour ce faire, les différentes approches proposées ont été réunies dans un méta-algorithme hétérogène qui sélectionne automatiquement l'algorithme à déployer en fonction de la configuration matérielle cible. Ce méta-algorithme est couplé avec l'approche B&B@Grid proposée dans [Mezmaz et al., IEEE IPDPS'2007]. B&B@Grid répartit les unités de travail (sous-espaces de recherche codés par des intervalles) entre les noeuds de la grille tandis que le méta-algorithme choisit et déploie localement un algorithme de B&B parallèle sur les intervalles reçus. L'approche combinée nous a permis de résoudre à l'optimalité et efficacement les instances (20 × 20) de Taillard.

Mots clés:

Branch-and-Bound Parallèlle, Calcul hétérogène, Processeurs Graphiques, Machines multi-coeurs, Problème d'ordonnancement du Flowshop, Grid'5000, Optimisation Combinatoire, Méthodes exactes.

Acknowledgments

During these three years of PhD, I have been receiving help, support and encouragement from many people I would like to thank through these lines.

First of all, I would like to express my deepest gratitude to my supervisor Pr. Nouredine Melab. I thank him for the effort he put into training me in the scientific field. Thanks to his experience and encouragement I was able to conduct my research in the right direction despite the times doubt. I learned a lot of him during these years both professionally and personally. It was a pleasure to work with him.

I am also very much grateful to Dr. Mohand Mezmaz and Dr. Ahcène Bendjoudi for their generous help. I would like to thank them for sharing their time very often with me and for the scientific discussions we have held.

I am pleased to thank the members of my thesis examination committee: Pr. Pierre Manneback and Pr. Catherine Roucairol for reviewing this work, for spending their time on careful reading and for their many valuable comments on how to improve the thesis manuscript. I also thanks Pr. Pierre Boulet and Dr. Stephane Genaud for agreeing to examine my thesis defense.

I shall not forget my colleagues and friends I have known during these three years of PhD, with whom I had exchanged so many useful tips and valuable ideas and nice and happy moments: Khedidja Seridi, Rahma Yengui, Hajer Sassi, Karima Boufaras, Sana Cherif, Hana Krichen, Chiraz Trabelsi, Nadia Dahmani, Ines Bahri.

It is also a pleasure to thank all the people working in the Dolphin Team: Katerina, Sezin, Sophie, Marie-Eléonore, Julie, Julie, Bayrem, Mostepha Redouane, Thé Van, Moustapha, Mathieu, Tuan, Yacine, Martin.

These acknowledgments would not be complete without thanking my family for their constant support and care. Today I feel that my parents hard work and dreams have been blossomed. I thank my father Abdelrazzek, my mother Jamila and my sister Olfa. I also wish to extend my thanks to my family-in-law for their kind support.

Finally, I would like to mention two other people who are very important in my life: My husband Mahmoud and my little coming son. A warm thanks to Mahmoud for everything, for making me so happy, for his comprehension and for his unconditional support and encouragement during these three years.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The Ph.D thesis, presented in this document, has been realized within the DOLPHIN [1] research group from CNRS/LIFL, Inria Lille-Nord Europe and Université Lille 1.

Branch-and-Bound (B&B) algorithms are well-known methods for solving to optimality NP-hard combinatorial optimization problems (COPs)[2] such as job scheduling, task allocation, network routing, etc. They are based on an implicit enumeration of all feasible solutions and return the guaranteed optimal one(s). The basic idea of a B&B algorithm is to traverse a subset of feasible solutions over a search space and eliminate others when they are not likely to lead to an optimal solution. The algorithm proceeds in several iterations during which it recursively decomposes the problem being solved into subproblems and progressively improves the best solution found so far. The generated and not yet examined subproblems are kept into a list initialized to the original problem. At each iteration, a subproblem is selected from this list, according to some strategy (depth-first, best-first,...), using the *selection operator*. The *branching operator* performs its decomposition into other subproblems. The *bounding operator* calculates a lower bound of each generated subproblem. Each subproblem having a lower bound greater than the best solution found so far is eliminated using the *pruning operator*, this means that it will not be decomposed.

In practice, COPs are often computation time-intensive, therefore even with highly efficient bounding and pruning operators only small instances can be solved in a reasonable amount of time using a single processing core [Garey 1976]. Over the last decades, parallel computing has been revealed as an attractive way to deal with larger instances. Because the design and implementation of parallel B&B is strongly influenced by the computing platform [Bader 2005], different architecture-oriented contributions have been proposed for Massively Parallel Processors (MPP) [Allen 1997], Networks

[1] Discrete multi-objective Optimization for Large-scale Problems with Hybrid dIstributed techNiques

[2] An optimization problem consists in minimizing or maximizing a cost function. Without loss of generality, in this Ph.D thesis the minimization case is considered.

or Clusters of Workstations (NOWs or COWs) [Tschöke 1995, Quinn 1990] and Shared Memory or SMP machines [Casado 2008]. The proposed approaches are based on three parallel models presented in [Gendron 1994]: parallel application of the operators on the generated subproblems (Type 1), parallel building and exploration of a B&B tree (Type 2), and parallel (cooperative or independent) building and exploration of several B&B trees (Type 3). These parallel approaches have been later revisited for large-scale computational grids using the Master-Worker paradigm [Mezmaz 2007a], the hierarchical paradigm [Bendjoudi 2012] and the Peer-to-Peer paradigm [Djamai 2011].

Recently, Graphics Processing Units (GPU accelerators) have emerged as a new popular support for massively parallel computing. Such resources supply a great computing power, are energy-efficient and highly available everywhere: laptops, desktops, clusters, etc. During many years, GPU computing has been used to speed up the execution of graphics and video applications. Its utilization has been extended to other application domains such as High Performance Computing (HPC). Indeed, GPU accelerators are more and more integrated into clusters, computational grids and clouds. These last years the first machines in the Top500 ranking include GPUs. One can say that the HPC hardware evolution follows the application needs, now the challenge is how to design and implement efficient algorithms for those GPU-enhanced environments? [3]

In combinatorial optimization, such challenge has been successfully addressed for meta-heuristics (near-optimal methods) [Luong 2011] but little attention has been given to exact methods such as B&B algorithms. Indeed, few works on GPU-based B&B [Lalami 2012, Carneiro 2011] and multi-core B&B [Casado 2008, Barreto 2010] exist. However, to the best of our knowledge, no contribution addressing B&B on heterogeneous environments combining multi-core processors with GPUs exists. In this Ph.D thesis, the major objective is to revisit the design and implementation of B&B algorithms for GPU-enhanced multi-core environments for solving challenging COPs. The revisited B&B should be portable in a transparent way on laptops, workstations, clusters and computational grids. Without loss of generality, the Flowshop Scheduling Problem (FSP) is considered as a case study. The problem consists in scheduling a pool of jobs on a set of machines with respect to two constraints: the jobs are processed on all the machines in the same order and each machine can not process more than one job

[3] Edsger DIJKSTRA, 1972 Turing Award Lecture, “The Humble Programmer": “To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

at a time. The objective is to find a processing order on each machine such that the time required to complete all jobs is minimized. The lower bound function used in this work is the one of Johnson proposed in [Johnson 1954] for two machines and generalized in [Lenstra 1978] to more than two machines.

Rethinking B&B algorithms for GPU-enhanced multi-core environments raises several design and implementation challenges related to GPU computing, multi-core computing, hybrid computing combining GPU and multi-core, and heterogeneous cluster and grid computing. The challenges and associated contributions are addressed following an incremental design methodology: B&B for a single CPU core combined with a single GPU, B&B for a multi-core CPU combined with a single GPU, B&B for a multi-core CPU combined with multiple GPUs, B&B for a cluster or grid of heterogeneous computational nodes. The addressed issues and proposed contributions are summarized in the following:

- Preliminary experiments we have carried out on some Taillard's problem instances [Taillard 1993a] have shown that the time spent by the B&B algorithm in evaluating the lower bounds of the examined subproblems is on average between 97% and 99% of its total execution time. Such result demonstrates the need to parallelize the bounding operation. The first challenge is thus to revisit the parallel bounding model on GPU considering a single CPU core and a single GPU. Such challenge is difficult because on the one hand, a GPU is a many-core co-processor device that provides a hierarchy of memories having different sizes and access latencies making challenging data placement and sharing. Besides, the GPU device provides a highly multi-threaded environment where the threads are scheduled and executed as warps[4] using the SIMD model. Such model is well-suited and very efficient for regular kernels. However, if the kernel code contains loops and conditional instructions another challenging issue has to be faced: thread or branch divergence. Such problem arises when for instance the threads of the same wrap have to execute simultaneously different branches of a conditional instruction. As the execution model is SIMD, the threads are executed in a serial way slowing down the execution. On the other hand, the Johnson's FSP lower bound function makes use of six data structures of different sizes and access frequencies. Moreover, the function contains several loops and conditional instructions making irregular its associated code.

  The first contribution of this Ph.D thesis consists in revisiting on GPU the parallel bounding model (Type 1). Having in mind the characteristics of both the lower

[4] A warp contains 32 threads in the G80 model, each thread executing the same code called a kernel

bound function and the GPU device mentioned above, the challenge is twofold: (1) defining a new approach for optimal mapping of the data structures of the lower bound function on the hierarchy of memories provided in the GPU device. A careful analysis is required of both the data structures (size and access frequency) and the GPU memories (size and access latency). (2) proposing a new approach for thread/branch divergence reduction through a thorough analysis of the different loops and conditional instructions of the bounding function.

- Even if the bounding operator is highly time-consuming, the experimental study has shown that there is no guarantee that its GPU-based acceleration will significantly improve the performances of the B&B. The parallel bounding-based algorithm requires, in fact, some additional tasks which induce a notable overhead: the preparation of the pool of subproblems on which the bounding operator is applied, the transfer of the pool from CPU to GPU, and the transfer of the computed lower bounds from GPU to CPU. Therefore, the second contribution of this thesis is to extend the former approach to minimize such overhead by further rethinking the design and implementation of a GPU-accelerated B&B based on the parallel tree exploration (Type 2). For achieving optimized CPU-GPU communications, we propose to revisit on GPU the parallel tree exploration model which is reflected by the parallelization on GPU of the branching and pruning operators as well even if they consume less time than the bounding operator. For achieving this, we investigate two different approaches for executing the branching, bounding and pruning operators on GPUs: the first multiple-nodes driven approach consists in exploring in parallel different sub-spaces of the tree, the second single-node driven approach limit the granularity of each thread to the application of an operator to a single node. For both approaches, the selection operator is executed on the CPU side but efficiently adjusted according to the tackled problem instance and the hardware configuration. Indeed, an adaptive version of the GPU-accelerated B&B is proposed where the selection operator is revisited so that the size of the selected pool is tuned dynamically using an empirical heuristic for parameters auto-tuning at runtime.

- Although the proposed GPU-accelerated B&B algorithms allow one to significantly reduce the execution time needed to explore the B&B search tree, further speedups could be reached if the multiple CPU cores available on the underlying platforms are judiscioulsy used. In this context, our contributions are (1) rethink the B&B algorithm for multi-core machines endowed with multiple processing cores without GPUs, (2) to propose a multi-core CPU-GPU accelerated B&B by investigating two patterns for combining multiple CPU cores and a single GPU and (3) to redesign the CPU-GPU accelerated B&B for multi-GPU enabled configurations.

- To be relevant to the arrival of GPU accelerators and the advent of multi-core processors in clusters and grids, we finally propose a large-scale distributed version of the heterogeneous multi-core GPU-accelerated B&B algorithm. The approach consists in hierarchically combining two levels of parallelism by (1) dividing the B&B tree exploration among multiple distributed grid nodes and (2) exploring in parallel each sub-tree. For achieving this, a B&B meta-algorithm is proposed and coupled with the B&B@GRID approach proposed in [Mezmaz 2007a]. Indeed, while B&B@GRID allows one to efficiently partition the B&B tree search among distant grid nodes, the meta-algorithm explores assigned sub-trees using the parallel B&B algorithm that best fits the targeted hardware configuration.

This thesis is organized into six chapters. Chapter 2 introduces all the background and prerequisites necessary to the comprehension of the global document namely the B&B algorithm as well as the FSP problem. It also provides an overview of the different parallelization strategies of B&B and a synthesis of the existing work dealing with parallel B&B classified by the target computational platform (multi-threaded many-core processors, shared memory multi-core systems and computational grids).

Chapter 3 describes our first contribution which consists in rethinking for GPU the parallel evaluation of bounds model. First, the design of the proposed GPU-accelerated B&B based on the parallel evaluation of lower bounds is introduced. Afterward, the thread divergence issue is addressed: the scenarios where the thread divergence occurs in the studied FSP lower bound, a review of different works of the literature for reducing thread divergence and details about the proposed mechanisms we propose to reduce the number of divergent branches within a warp are given. The memory access pattern is detailed next. Finally, details about the performed experimental study (the used experimental metrics, the experimented problem instances, etc.) are given and the obtained results are discussed.

Chapter 4 introduces the adaptive selection operator based on the auto-tuning heuristic and presents the two investigated approaches to reduce CPU-GPU communications latency: multiple-nodes driven GPU-accelerated and single-node driven GPU-accelerated B&B. The details on the parallelization of each operator are provided.

In Chapter 5, the two investigated approaches for the heterogeneous multi-core CPU-GPU accelerated B&B are presented together with the CPU-GPU accelerated B&B for multi-GPU enabled configurations.

In Chapter 6, the overall architecture of the parallel heterogeneous B&B for computational grids is introduced. The comprehensive description includes details about the B&B meta-algorithm and the used B&B@GRID approach.

Finally, some concluding remarks are drawn in Chapter 7. In addition, we propose some future extensions of the proposed approaches and some perspectives related to the evolution of the context of High Performance Computing.

Chapter 2

Parallel Branch and Bound algorithms

Contents

This first chapter presents all the background and prerequisites necessary to the comprehension of the global document. First, we introduce the Flowshop Scheduling Problem, a combinatorial optimization problem considered as a case study in this thesis. Thereafter, we introduce the Branch and Bound algorithm. An overview is made on the different parallelization strategies of Branch and Bound with the aim to accelerate the search process. Afterward, a synthesis of the existing work dealing with parallel B&B classified by the target computational platform (multi-threaded many-core processors, shared memory multi-core systems, computational grids) is presented. For each considered category of platform, related challenges are identified and discussed.

2.1 Introduction to combinatorial optimization

In combinatorial optimization, also referred to as discrete optimization, the goal is to find one or more (near-) optimal configuration(s) among a finite set of possible configurations (or solutions) optimizing a given objective also called cost function.

In practice, a wide range of problems in different industrial and economic fields, such as task allocation, job scheduling, network routing, cutting, packing, etc. can be modeled as NP-hard combinatorial optimization problems (COPs). Because of its practical relevance and without loss of generality, the focus of this work is on the Permutation Flowshop Scheduling Problem (FSP) which is one of the most known problems in combinatorial optimization and scheduling area.

2.1.1 The Permutation Flowshop Scheduling Problem

Permutation Flowshop Scheduling Problems [Bonney 1976, King 1980, Allahverdi 1999] are common in manufacturing environments in which a set of n jobs are to be processed on a series of m machines optimizing a given objective function. FSP consists in scheduling a set of n jobs on a set of m machines so that each of the jobs J_1, J_2, ..., J_n is processed on the machines M_1, M_2, ..., M_m organized in the line. Job J_i (i = 1, 2, ... , n) consists therefore of a sequence of m operations O_{i1}, O_{i2}, ... O_{im} where O_{ik} corresponds to the processing of J_i on M_k during an uninterrupted processing time p_{ik}. The objective is to find a processing order on each M_k such that the time required to complete all jobs, called makespan, is minimized. In the remainder of this thesis, FSP designates a permutation FSP [Allahverdi 1999, Hejazi 2005]. For $m = 2$, an optimal schedule for FSP can be found in O(n. log n) steps using Johnson's algorithm [Johnson 1954]. For $m \geq 3$, the problem has been shown to be NP-hard [Garey 1976].

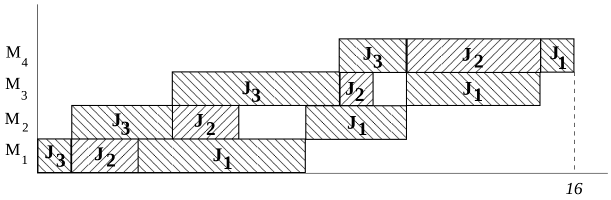
To sum up, a feasible solution of FSP should satisfy these constraints:

- A machine can not start processing a job if all the machines, which are located upstream, did not finish their treatment. Thus, the operation o_{ij} cannot be processed by the machine m_j if it is not completed on m_{j-1}.
- An operation can not be interrupted, and the machines are critical resources, because a machine processes one job at a time.
- The sequence of jobs should be the same on every machine, e.g. if j_3 is treated in position 2 on the first machine, j_3 is also executed in position 2 on all machines.

Figure 2.1 shows an example of an FSP instance (with $n = 3$ and $m = 4$) and its associated optimal solution.

	M_1	M_2	M_3	M_4
J_1	5	3	4	1
J_2	2	2	1	4
J_3	1	3	5	2

Processing Times



Optimal Solution

Figure 2.1: Illustration of a permutation FSP with $n = 3$ and $m = 4$. The table reports the processing times of the jobs on the machines. The Gantt diagram shows the optimal solution to the problem instance.

2.1.2 Resolution methods for combinatorial optimization problems

The techniques for solving COPs can roughly be classified into two main categories: exact and heuristic or approximate methods.

An approximate search method aims to find a near-optimal solution to the tackled problem in a reasonable time by exploring a selected part of the solution space in which good quality solutions are expected. Unlike exact methods, there is no guarantee of optimality of the found solutions. The most popular search algorithms in the class of heuristic methods are metaheuristics. Metaheuristics are general-purpose optimization methods that can be applied to any kind of problems as they contain limited problem-specific knowledge in their design line compared to specific heuristics. Metaheuristics are either single-solution based: one solution is initially considered and iteratively improved along with the solution space exploration process (Tabu search, simulated annealing, hill climbing, etc), or population-based: a set of solutions is considered and simultaneously or independently improved during the exploration process (Particle Swarm optimization, Ant Colonies, Evolutionary Algorithms, etc).

Exact methods aim to find the optimal solution(s) to the problem and to prove its (their) optimality. This class of methods includes Branch and X methods [1], constraints programming, dynamic programming, A*, etc. Branch and X methods are tree-based and enumerative methods which intelligently (or implicitly) explore the whole search space in order to find the optimal solution to the problem. The problem is solved by subdividing it into smaller and simpler subproblems.

Although exact methods allow to resolve a problem with guarantee of optimality, they are computing intensive and very time consuming when tackling hard and large scale problem instances. The goal of this thesis is to design efficient exact algorithms for combinatorial optimization problems. Therefore, in the next section, a comprehensive overview of how this tree-based search algorithm operates is given.

2.2 Branch and Bound algorithms

Branch and Bound (B&B) algorithms [Gendron 1994, Papadimitriou 1982] are one of the most used tree-based exploratory methods for solving to optimality NP-hard discrete optimization problems. A B&B algorithm allows one to find the optimal solution(s) of a problem and to prove that no better other one exists. It performs an implicit enumeration of all feasible solutions and returns the guaranteed optimal one. The basic idea of the B&B algorithm is to traverse a subset of feasible solutions over a search space and eliminate others when they are not likely to lead to an optimal solution. The search space is explored by dynamically building a tree whose root node is the original problem. The leaf nodes are the potential solutions and the internal nodes are sub-spaces of the total solution space.

[1] X refers to Bound, Cut, Price, etc.

Each internal node contains a subproblem obtained by decomposition [2]. The construction of such tree and its exploration are performed using four operators: *branching*, *bounding*, *selection* and *pruning*.

2.2.1 Serial B&B

The algorithm proceeds in several iterations during which the best solution found so far, namely *best-sol*, is progressively improved. During the exploration process, the generated and not yet examined (pending) tree nodes are kept into a list, namely *pending-nodes*, initialized with the original problem. At each iteration of the algorithm, the following steps are performed:

- The *selection operator* chooses the next tree node to be explored from the *pending-nodes* list according to a defined selection strategy. Possible selection strategies include depth-first, breadth-first and best-first. Depth-first strategy explores the furthest left branch until the leaves are reached or the branch is eliminated. This strategy aims to reach a feasible solution to the problem as soon as possible in order to update the current best solution. Breadth-first strategy consists of exploring all the nodes of the level l before moving to the level $l+1$. Using this strategy, memory issues often arise making it unpractical to do such searches for larger problems. Best-first strategy uses the bounding function to perform a descending order of the set of pending nodes. It assumes that the next tree node to be explored is the node with the best bound which is more likely to quickly lead to an improving solution. The advantage of this strategy is the possible early improvement of the bound.

- The *branching operator* subdivides a solution space into two or more disjointed subspaces to be investigated in a subsequent iteration. This operator decomposes a given problem into smaller subproblems through the addition of constraints.

- The *bounding operator* computes a bound value (a lower bound for a minimizing problem such as considered in this thesis without loss of generality) of the optimal solution of each generated subproblem. A bounding function is used to estimate the quality of the solutions covered by the evaluated node. Tight bounds help the B&B to eliminate nodes of the solution space that are not likely to guide the search to an improving solution.

- Each subproblem having a lower bound greater than the cost of the best solution found so far, is eliminated using the *pruning operator*. This operator helps to reduce

[2] In the rest of the document, the terms node and subproblem will be used to refer to an internal node of the B&B tree.

the size of the space search to a computationally manageable size by cutting some branches of the tree.

Algorithm 1 gives the general template of the B&B method which, in summary, consists in recursively branching and bounding (sub)problems aside eliminating some of these and exploring the remaining ones according to a predefined strategy. The previous exploration steps are repeated until all solutions are explicitly or implicitly visited.

Algorithm 1 General template of the B&B Algorithm.

```
Create the initial problem;
Insert the initial problem into the tree as a root;
Set the Cost_best_solution to $+\infty$;
Set the Best_Solution to $\emptyset$;

while not_empty_tree() do
    Sub_Problem = Take_sub_problem();
    if Is_leaf ( Sub_Problem ) then
        Cost_best_solution = Cost_Of( Sub_Problem );
        Best_Solution = Sub_Problem;
    end
    else
        Lower_Bound = compute_lower_bound(Sub_Problem);
        if Lower_Bound $\leq$ Cost_best_solution then
            Branch(Sub_Problem);
            Insert child subproblems into the tree;
        end
        else
            Prune (Sub_Problem);
        end
    end
end
```

2.2.2 Illustration on the Permutation Flowshop Scheduling Problem

Figure 2.2 illustrates the B&B enumeration scheme applied to the 3-jobs FSP instance described in Figure 2.1. The resolution of the problem proceeds by building a search tree whose root node contains the original problem (empty schedule). The decomposition of this problem generates n sons, each of them designates a new subproblem. The son number i corresponds to the subproblem where job J_i is scheduled first on all machines. The recursive application of the decomposition operator on the generated subproblems allows one to develop the search tree. The number of potential schedules (permutations) is $n!$, which is highly exorbitant for large problem instances. For instance, for FSP Taillard's instances with 500 jobs, the search space is composed by 500! permutations.

A major powerful way to speed up the exploration of large search trees, is the use of an efficient bounding operator which allows to significantly reduce the size of the explored search space. Applied to a subproblem, such operator associates a value to its corresponding tree node using a lower bound function. As illustrated in Figure 2.2, a subproblem is not decomposed (and its tree node is pruned) if its lower bound value is greater than the cost of the best schedule found so far during the exploration of the search tree.

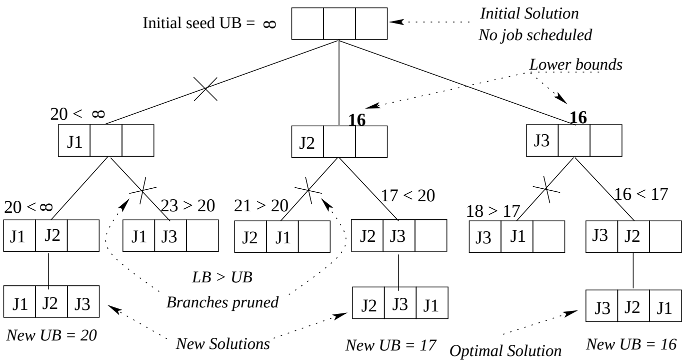


Figure 2.2: The search tree generated and explored by a B&B algorithm for solving an FSP with 3 jobs. Nodes with a lower bound (LB) greater (resp. lower or equal) than the current best solution are pruned (resp. decomposed or branched).

Using a bounding operator not only reduces the size of the B&B search tree but also impacts its shape. Indeed, the number of tree nodes at each level is unpredictable and depends on the number of branches pruned by the algorithm (due to their associated lower bounds) which yields to an irregular structure. This irregularity is analyzed in the following section.

2.2.3 Analysis of the irregularity of the B&B algorithm

In order to demonstrate the unpredictable and irregular nature of the workload characterizing B&B tree traversal, two preliminary analyses have been performed to (1) evaluate the intra-instance irregularity and (2) compare the differences between the structures of the B&B trees (inter-instances irregularity).

2.2.3.1 Intra-instance irregularity analysis

The irregularity of the tree explored by B&B is one of the major challenges for revisiting this algorithm on heterogeneous architectures. The evaluation of the irregularity of a B&B tree is not easy. Indeed, the size of the explored tree, when solving a small instance, is not large enough to be representative, while the resolution of a large instance needs several months and sometimes years of computing. In addition, the size of the explored tree, when solving a large instance, is huge to be stored in order to be analyzed. Therefore, the irregularity measures presented in this section are only based on instances of intermediate size. These instances are the 10 FSP Taillard's instances [Taillard 1993b] defined with 20 jobs and 20 machines, namely Ta021, Ta022, ..., and Ta030.

The resolution of these 10 instances with a serial B&B algorithm needs about 18 days of computing in total. The 10 explored trees are saved in 10 different files with a total size of about 120 GB. Each line of these files provides information on one subproblem. For each explored subproblem, the algorithm saves the ID of this subproblem, its depth in the tree, the value of the associated bound, and the ID of its father subproblem. The depth of a subproblem in a B&B tree is equal to the number of scheduled jobs. For example, the root problem has a depth equal to 0 since no job is scheduled for this problem. A subproblem with a depth 18 (i.e. 18 scheduled jobs) admits 2 possible solutions. In our B&B algorithm, subproblems with depth 18 are considered simple and are not decomposed. Therefore, the maximum depth in the obtained B&B tree is equal to 18.

The goal of the first analysis is to evaluate the irregularity within each instance's tree. Figure 2.3 shows the irregularity observed on the tree obtained when solving the instance Ta023. The ordinate of the figure represents the different possible depths. For each depth, this row-stacked histogram gives the percentage of subproblems with no children, the percentage of subproblems with 1 child, ... and the percentage of subproblems with 20 children. For the depth 10 for example, the percentage of subproblems with no children is equal to 53%, those with 1 child is equal to 23%, and those with 2 children is equal to 11%, etc.

Figure 2.3 shows that these percentages are different from a depth to another. At the depth 2 for example, the percentage of subproblems with 17 children (i.e. 41%) is greater than the percentage of subproblems with 15 children (i.e. 10%), while, at the depth 3, the percentage of subproblems with 17 children (i.e. 1%) is smaller than the percentage of subproblems with 15 children (i.e. 12%). Ta023 is selected to be plotted in Figure 2.3 because the size of its tree is the largest one. We also observed that the other 9 trees are as highly irregular.

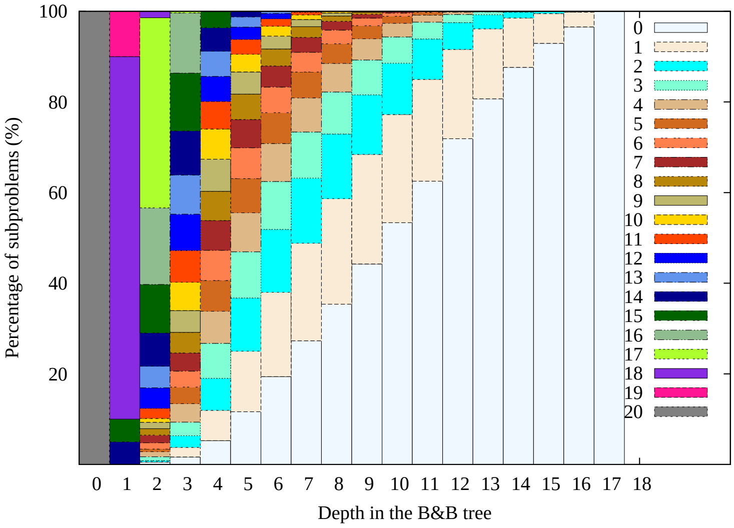


Figure 2.3: Percentage of subproblems with corresponding number of children per depth in the instance Ta023.

2.2.3.2 Inter-instances irregularity analysis

This subsection presents the inter-instances irregularity analysis. The goal of this analysis is to measure the differences between the explored trees when solving the 10 instances. This subsection compares thus the 10 obtained trees in Subsection 2.2.3.1.

Figure 2.4 shows the differences between the structures of the B&B trees explored during the resolution of these different 10 instances. The ordinate of the figure represents the number of subproblems and the abscissa the possible depths of the B&B tree. Each curve represents the number of subproblems explored at each level (depth) of the tree. Each curve corresponds to one of the 10 instances. The figure shows that the sizes of the B&B trees are very different even if these instances are defined by the same number of jobs and machines. For example, for the instance Ta025, 100.000.000 nodes are explored on the level 11 of the search tree, while it is about 50.000.000 for the instance Ta022 at the same level. As other example, the Ta023 B&B tree contains about 85 times more subproblems than the tree of Ta030. These 10 curves look like normal distributions with different means and standard deviations.

From the two analyses, one can conclude that the trees associated to the 10 instances exhibit different sizes and shapes demonstrating their irregularity. The same should go for the other Taillard instances. We believe that the conclusions drawn from the experiments are the same whatever is the shape of (how irregular is) the tree and thus for any tree-based application.

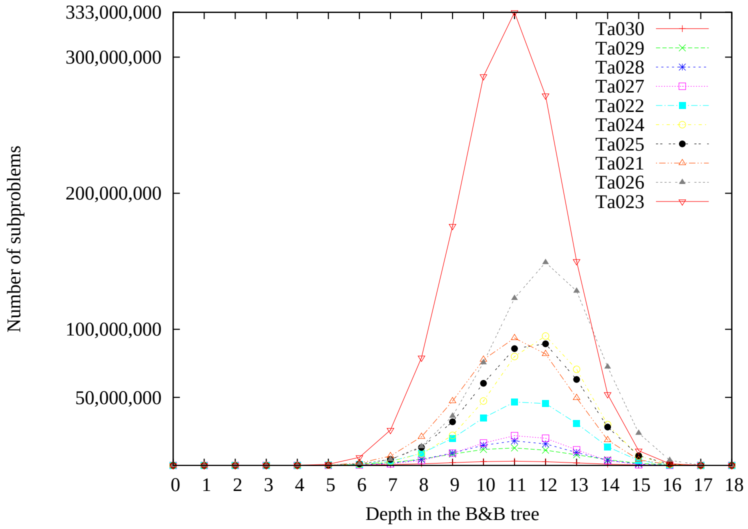


Figure 2.4: Comparison of the structures of the 10 standard instances of FSP defined with 20 jobs and 20 machines

Although B&B can be endowed with performant mechanisms (efficient bounding and pruning) allowing to significantly reduce the number of branches to be explored, the size of the remaining tree is still huge when tackling large instances making its serial traversal very time consuming. Therefore, the idea to parallelize them has naturally emerged as an attractive way to solve problems faster and to tackle larger problems. For example, an efficient parallel resolution of the FSP instance Ta056, performed in [Mezmaz 2007a], lasted 25 days with an average of 328 processors picked at 1195 processors, and a cumulative computation time of about 22 years.

2.3 Parallel Branch-and-Bound algorithms

The parallelization of B&B is well studied in the literature and many classifications have been conducted [Trienekens 1992, Gendron 1994, Roucairol 1996, Melab 2005, Crainic 2006]. In Appendix B, a thorough overview of the existing parallelization strategies for B&B algorithms is presented.

The most recent taxonomy is the one proposed by Melab in [Melab 2005] which is based on the classification of Gendron *et al.* [Gendron 1994]. In this taxonomy, four models of parallel B&B algorithms are identified: parallel multi-parametric model, parallel tree exploration model, parallel evaluation of the bounds, and parallel evaluation of a single bound.

2.3.1 Parallel tree exploration model

The parallel tree exploration model consists in launching several B&B processes to explore simultaneously different paths (sub-trees) of the same tree (see Figure 2.5). The different selection, branching, bounding and pruning operators are executed in parallel either in a synchronous or asynchronous fashion. In a synchronous mode, B&B processes need to exchange global information that accelerate the tree traversal such as the best solution found so far. In an asynchronous mode, B&B processes communicate unpredictably.

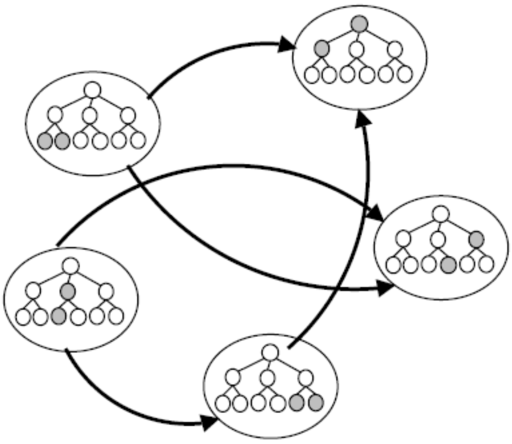


Figure 2.5: Illustration of the parallel tree exploration model.

Compared to the other models, the parallel tree exploration generates great interest and has been the subject of several existing research on parallel B&B algorithms. The degree of parallelism of this model is actually very high mainly for large problem instances which justify by itself its use on top of many-core and multi-core architectures. However, this parallelization strategy leads to unpredictable and unbalanced work units which raises several challenges on top of data-parallel platforms such as GPUs.

2.3.2 Parallel multi-parametric model

The multi-parametric parallel model consists in considering simultaneously several B&B processes which differ by one or more operator(s), or have the same operators differently parameterized (see Figure 2.6). Each B&B process explores a tree which is not necessarily the same compared to its concurrents. Parallel B&B algorithms may differ by the branching operator such as in [Miller 1993], by the selection operator [Janakiram 1988] or use different bounds such as in [Kumar 1984]. The main advantage of the multi-parametric model is its genericity enabling its use transparently to the user. However, it might lead to a redundant exploration of some subproblems which slow down the exploration time needed for traversing the B&B search tree.

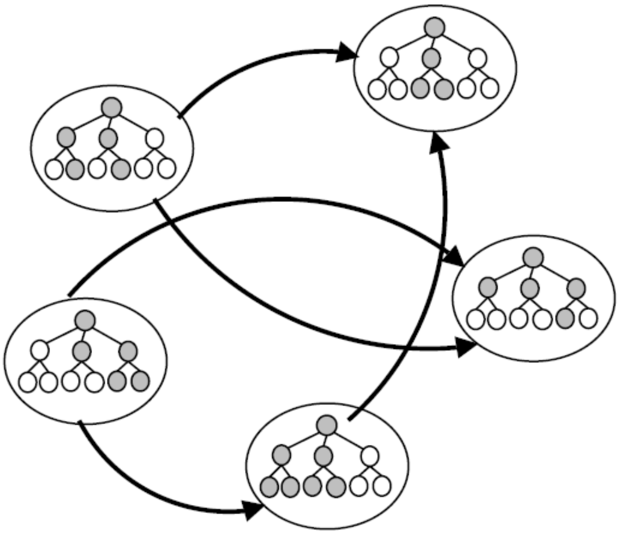


Figure 2.6: Illustration of the parallel multi-parametric model.

The multi-parametric parallel model is coarse-grained and well-suited for MIMD (Multiple Instruction Multiple Data) architectures rather than for data-parallel architectures such as GPU accelerators where the executed work units should be the same. Moreover, B&B processes might need to exchange information such as new best solutions which is not recommended in GPU computing since synchronization barriers significantly harm the performances. Finally, the degree of parallelism created by this parallelization strategy is not exploitable on highly parallel environments unless it is combined with other parallel models.

2.3.3 Parallel evaluation of the bounds

The parallel evaluation of bounds consists in launching only one B&B process where a parallel evaluation of the subproblems generated by the branching operator is performed (See Figure 2.7).

This model is data-parallel, intrinsically asynchronous and fine-grained (the cost of the bound evaluation) which is the execution model that better fits graphics processing units. However, the parallel bounding is profitable only if the evaluation operation of the bounds is time consuming mainly on architectures with high ratio of arithmetic operations.

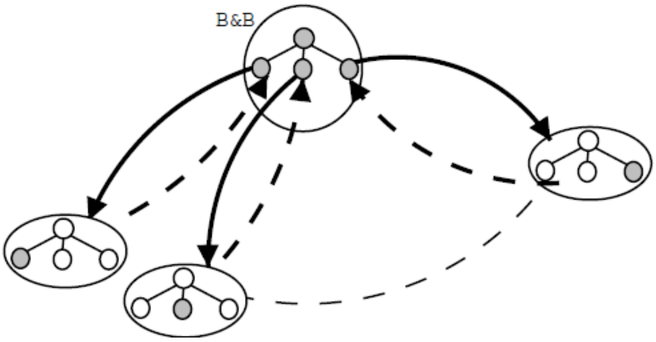


Figure 2.7: Illustration of the parallel evaluation of bounds model.

2.3.4 Parallel evaluation of a single bound/solution

The parallel evaluation of a single bound/solution is similar to the previous one (parallel evaluation of the bounds) as only one B&B process is used. In this model, a set of processes evaluate in parallel the bound/objective function of a single node. It requires the definition of new specific elements for the problem being processed, like partial objective functions and a method to aggregate these partial results. As the implementation of this model is naturally synchronous, it is crucial to memorize all the partial evaluation values for the solutions being evaluated.

The parallel evaluation of a single bound/solution is strongly problem-specific and should be considered when the bounding/objective function is too awkward and easy to parallelize. In our case, it is not well suited to the FSP problem considered as a case study in this thesis since the objective function can not be finely dissociated and the degree of parallelism induced is not profitable for the GPU computing power.

Because the design of parallel B&B is strongly influenced by the computational platform [Bader 2005], many architecture-oriented contributions for parallel B&B have been proposed [Djerrah 2006] ranging from networks or clusters of workstations [Tschöke 1995, Quinn 1990, Aida 2002] and shared memory machines [Casado 2008, Mans 1995] to Graphics Processing Units [Lalami 2012, Carneiro 2011]. As discussed above, the parallel tree exploration and the parallel evaluation of the bounds models better suit highly threaded data-parallel architectures such as GPU accelerators. Rethinking these two models is however not straightforward and many challenges have to be faced. In the following, an overview of the existing work dealing with these two parallel B&B models classified by the target computational platform is presented. For each considered category, related challenges are identified and discussed.

2.4 Parallel B&B for Graphics Processing Units

Graphics Processing Units (GPUs) are at the leading edge of many-core parallel computational platforms in several research fields. For years, the use of graphics processors was dedicated to high-definition 3D graphics. However, since NVIDIA released the Compute Unified Device Architecture (CUDA) programming model [NVIDIA Corporation 2011a], massive data processing capability of modern GPUs is attracting researchers to explore mapping more general non-graphics computations onto them.

The execution of a GPU program starts with host (CPU) execution. When a kernel function is invoked, or launched, the execution is moved to a device (GPU), where a large number of threads are generated to execute the kernel function many times in parallel leading to a valuable data parallelism. All the created threads are organized into thread blocks and grids of thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread identifier within its thread block. Threads are partitioned into groups of 32 threads called warps which execution is scheduled following a time-sharing strategy. A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block identifier within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. Active GPU threads have access to several memory spaces with different characteristics that reflect their distinct usages. These memory spaces include global, local, shared, texture, and registers. For further information about GPU programming and the GPU memory hierarchy please refer to Appendix A.

Little attention has been given to the study of B&B in massively parallel environments like GPUs. Our work [Chakroun 2011] was among prior contributions in this context. The first concern when designing an unpredictable and irregular tree-search algorithm such as B&B on GPUs (see Section 2.2.3), is to identify the best way to extract ample fine-grained data-level parallelism with a high ratio of arithmetic operations which best suit the GPU programming model taking into account that:

1- First, GPUs are based on the Simple Instruction Multiple Data (SIMD) programming model which assumes multiple processing elements performing the same operation on multiple data points simultaneously.

2- Second, the B&B search tree is unpredictable and extremely unbalanced and the variance in the size of the B&B sub-trees is very high making it difficult to evaluate the required computation and memory space for traversing each of them. Besides, search algorithms such as B&B have dynamic access behavior to data structures and varying computational workload which leads to a strongly input-dependent program behavior and to a challenging data mapping on the device memories.

3- Third, for large problem instances and therefore large search trees, the amount of data to be transferred between CPU and GPU is huge.

Therefore, because of variable program flow, GPU threads executing B&Bs may have different accesses to the memory hierarchy of the GPU and different behaviors within a same warp which induces thread divergence and impacts the throughput of the application. Moreover, to allow efficient solving of large problems, the overhead induced by the data transfer between CPU and GPU should be minimized. These three challenges are detailed in the following sections.

2.4.1 Thread divergence

To manage thousands of threads, each streaming multi-processor (see Appendix A.1) schedules and executes threads in groups of parallel threads called warps following a SIMT (Single-Instruction Multiple Thread) fashion. When a multi-processor is given one or more thread block(s) to execute, it splits it/them into warps that contain threads of consecutive and increasing thread identifiers with the first warp containing thread 0.

For each instruction of the flow, the multi-processor selects a warp that is ready to be run. A warp executes one common instruction at a time, so best performances are attained when all threads of a warp follow the same control flow path. When threads in the same warp follow different paths of control flow, the scenario is referred to as thread or branch divergence [Fung 2007, Meng 2010]. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path. When all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp. Different warps execute independently regardless of whether they are executing common or disjointed code paths.

For example, for an *if-then-else* construct, full efficiency is achieved when either all threads execute the *then* part or all execute the *else* part. When threads disagree on their execution path, the execution of the warp will require multiple passes through these divergent paths. One pass will be needed for threads that follow the *then* path and another pass for those that follow the *else* path. These passes are serially executed which induces extra execution time.

As quoted above, for GPU-based B&B, thread divergence often occur because the GPU programming model assumes multiple processing elements performing the same operation on multiple data points simultaneously. However, the parallel tree exploration and the parallel evaluation of bounds models are problem-dependent parallelisms that lead to an irregular computation depending on the data of each subproblem.

2.4.2 Memory access optimization

Memory access optimization is by far the most studied topic for improving GPU-based application performances [Ryoo 2008, Yang 2010, Mahmoudi 2013]. Indeed, many constraints are related to the use of the hierarchy of memory available on GPU architectures.

The first concerns for optimizing memory management is adjusting the pattern of accesses to the GPU device memory. Indeed, as detailed in Appendix A.3, CUDA-enabled devices use several memory spaces, which have different characteristics in terms of sizes and access latencies:

- At the thread-level, each thread has its own allocated registers and a private local memory. CUDA uses this local memory for thread-private variables that do not fit in the threads registers, as well as for stack frames and register spilling.
- At the thread block-level, each thread block has a shared memory visible to all its associated threads.
- At the grid-level, all threads have access to the same global memory. Texture and constant cached memories are two other memories accessible by all threads.

The goal of memory access optimization is generally to use as much fast memory and as little slow-access memory as possible which grants programmers to further improve the throughput of many high-performance CUDA applications. For example, for B&B applied to FSP, all thread blocks perform concurrent accesses to the six data structures of the problem when they execute the lower bound function. To optimize the performance of such application, the best mapping of the data structures is to copy them on the shared memory of the GPU device. However, for large problem instances the data structures do not fit in the shared memory for some GPU configurations. The challenge raised in this case, is therefore to find which data structure has to be mapped on which memory and in some cases how to split the data structures on different memories and efficiently manage their accesses.

The second issue that must be considered for optimizing memory access is memory coalescing. Memory coalescing occurs when threads of the same warp read global memory in an ordered pattern. If per-thread memory accesses within a warp constitute a contiguous range of addresses, accesses will be coalesced into a single memory transaction. Otherwise, accessing misplaced locations induces memory divergence and requires the processor to produce one memory transaction per thread. Because it leads to performance degradation, memory coalescing is one of the most critical aspect of performance optimization.

For GPU-based B&B, uncoalesced memory accesses occur very often and represent a very challenging issue. Indeed, during the exploration of the B&B tree, the number of nodes to generate are variable and depend on the level of the tree being explored mainly for FSP [3]. Due to such unstructured and unpredictable nature of the search tree and to unbalanced workload, threads of a same warp may read and write from/to different locations of the global memory.

2.4.3 CPU-GPU communication optimization

CPU-GPU communication optimization is a crucial issue encountered in GPU computing that aims to efficiently transfer data between the host and device. Because the peak bandwidth between the device memory and the GPU is much higher than the peak bandwidth between host memory and device memory, the policy of data transfers between the host and GPU devices can make or break the overall application performance.

For the GPU-based B&B, we applied the following recommended best practices for optimizing data transfer between the host and the device:

- Minimizing the amount of data transferred between host and device when possible even if that means running kernels on the GPU that get little speed-up compared to running them on the host CPU. Such technique has also been adopted in [Luong 2011].

- Using intermediate data structures in device memory that are operated on by the device and destroyed without ever being mapped by the host or copied to host memory.

- Batching many small transfers into one larger transfer which performs better than making each transfer separately since it eliminates most of the per-transfer overhead.

- Overlapping data transfers between the host and device with kernel execution and other data transfers such as assumed in [Mahmoudi 2013, Mahmoudi 2012]. Indeed, most modern GPUs have a copy engine that uses direct memory access to bypass

[3] For the knapsack problem for example the search tree is binary

the CPU when transferring data. This is much faster than normal data transfer and has the added benefit that the CPU is free to do something else. Furthermore, the copy engine is a separate unit from the kernel engine that runs the kernels on the GPU, which allows concurrent data transfer and kernel execution.

- Using non-blocking transfer functions that immediately return the control to the host and thereby allows CPU routines to be performed while the data are transferred to the device and the kernel is executed.

2.4.4 Related works

Designing irregular algorithms on top of SIMD architectures like GPUs is not straightforward. As quoted above, such algorithms use graph or pointer-based data structures such as trees or graphs and have irregular memory-access patterns. Therefore, few insights into exploiting the parallelism of irregular algorithms on top of GPUs exist. As far as the B&B algorithm is concerned, apart from our contribution only two works [Lalami 2012], [Carneiro 2011] deal with designing B&B on GPUs. Moreover, our work [Chakroun 2011] was the prior work investigating GPU-based B&B applied to FSP problems.

In [Carneiro 2011], the B&B is applied to the traveling salesman problem. The aim of this work is to use the GPU to exploit the advantages of depth-first search in parallel B&B algorithms, and evaluate the solutions space in parallel. In that proposed schema, the parallel tree exploration model is applied. Each node belonging to the *pending nodes* list is a concurrent depth-first search root. The depth-first search is performed by each thread, i.e. each thread is responsible for evaluating a small portion of the search space. When threads have finished their searches, they communicate the amount of solutions they have found and the best one. The reported results show that the GPU-based B&B is a dozen of times faster than the equivalent serial algorithm.

In [Lalami 2012], a GPU-accelerated B&B based on the parallelization of the bounding and branching operators is applied to the knapsack problem which is also a well-known combinatorial optimization problem. The revisited B&B algorithm is akin to the parallel evaluation of bounds model coupled with a parallel tree exploration model where a part of the operators are parallelized. In fact, it assumes that the decomposition and evaluation of the subproblems are performed on GPU while the pruning and selection operators are executed on the host side. Unlike other combinatorial optimization problems such as FSP, quadratic assignment problem QAP, traveling salesman problem TSP, etc. knapsack is solved using a binary search tree: at each level on the tree, a parent node has only two children. Hence, the workload computed by each thread of the device is the same and no irregular task balancing occurs. For FSP, applying a B&B induces a highly irregular workload due on the one hand to the unpredictable number of branches pruned by the algorithm and on the other hand to the representation of FSP. At each level of the tree, the number of new generated nodes and the number of promising nodes are variable and depend on the level of the tree being explored and on the best solution found so far. As exploration strategy, the author uses a breadth-first search strategy. This means that the pool of nodes that is off-loaded to the GPU are from the same tree level unless from successive levels. This selection strategy emphasizes the regular amount of the work flow that is assigned to each thread. Best obtained speedup in this thesis, have been registered for instances with great number of nodes and met a level around 20.

2.5 Parallel B&B for multi-core shared memory machines

Shared memory systems refer to a block of random access memory that can be accessed by several different central processing units (CPUs) in a multiple-processor computer system. This paradigm assumes that many simultaneous asynchronous processes (or threads of computation) share the same logical address space and access directly any part of the data structure in a parallel computation. Thanks to its single address space, the programmability of a parallel machine is enhanced by simplifying the issues of data partitioning, migration and load balancing.

2.5.1 Synchronization and caching issues

Usually, computations on multi-processor computers with shared memory can naturally synchronize on data structure states and thus need fewer explicit synchronization. However, for irregular applications with unpredictable execution traces and data localities, a characteristic of B&B as we demonstrate in Section 2.2.3, synchronization becomes an important obstacle to the correct and efficient implementation of parallel tree-search algorithms. Indeed, lack of synchronization could impacts the consistency of shared data and consequently the execution of the whole algorithm. Therefore, often, the recommendation when using multi-core shared memory machines, is to use locks as a synchronization mechanism since they enforce mutual exclusion and guarantee the coherence of the data.

Another issue encountered in multi-processor computers is ensuring cache coherence which refers to the consistency of data stored in local caches of a shared resource. Indeed, in a shared memory multi-processor machine with a separate cache memory for each processor, it is possible to have many copies of a data: one copy in the main memory and one in each cache memory. When one copy of a data is changed, the other copies must be changed also. Efficient cache coherence management ensures that changes in the values of shared data are propagated throughout the system in a timely fashion. Best known cache coherence mechanisms includes directory-based coherence (the data being shared is placed in a common directory that maintains the coherence between caches), snooping (individual caches monitor address lines for accesses to memory locations that they have cached), snarfing (a cache controller watches both address and data in an attempt to update its own copy of a memory location when a second process modifies a location in main memory).

In the following, an overview of the existing works of the literature that investigate parallel B&B algorithms on top of multi-core shared-memory systems is given.

2.5.2 Related works

In [Evtushenko 2009], the B&B solver (BNB-Solver), a software platform allowing the use of serial, shared memory and distributed memory B&B algorithm is presented. BNB-Solver is based on the parallel exploration of the tree and assumes that several CPU threads are used where each thread has its local pool of subproblems and shared a common pool of subproblems with other threads. In BNB-Solver, each thread executes a fixed number N of iterations of the global B&B algorithm [4]. During the N iterations, each thread stores the generated new subproblems in its local pool. It transfers a part of them from the local pool to the global pool when the N iterations end. Each thread tries to select from its local pool the next subproblem to be processed. If the local pool is empty, the thread selects a subproblem from the global pool. If the global pool is empty, the thread blocks itself until another thread puts at least one subproblem in the global pool. Once the global pool is not empty, blocked threads are released and take subproblems from the global pool. BNB-Solver ends when the global pool is empty and all threads are blocked.

PAMIGO [Casado 2008] (Parallel advanced multidimensional interval analysis global optimization) is a parallel B&B algorithm designed for shared memory multi-core architecture and based on the parallel tree exploration model. Two versions of this algorithm, called Global-PAMIGO and Local-PAMIGO, are proposed where POSIX Threads is used. In Global-PAMIGO, threads share the same pool of subproblems. In Local-PAMIGO, each thread has its own pool of subproblems. A synchronization mechanism between threads, using semaphores, is necessary in Global-PAMIGO. In Local-PAMIGO, a dynamic load balancing mechanism among threads and a termination detection mechanism are implemented. One of the major issues addressed by the authors of PAMIGO is the number

[4] Recall here that the B&B proceeds iteratively its search for the optimal solution

of threads. In Global-PAMIGO, the number of threads is always equal to the number of cores, and the number of threads in Local-PAMIGO must be equal to or less than the number of cores. In Local-PAMIGO, a thread stops when its local pool of subproblems is empty. At each iteration, a thread checks the number of running threads and creates a new thread if two conditions are met: the number of threads is less than the total number of computing cores and second the current thread contains at least two subproblems in its local pool. Local-PAMIGO ends when there is no more running threads, and Global-PAMIGO ends when the global pool is empty.

OpenMP has been used in [Paulavicius 2009] to implement a parallel B&B algorithm with combination of Lipschitz bounds for multi-core computers. The authors use breadth-first strategy to explore the B&B tree. Only the problem-specific bounding operator is parallelized in this approach.

In [Barreto 2010], the authors compare the serial, OpenMP (Shared memory parallel model) and MPI (Distributed memory parallel model) B&B approaches. These algorithms are used to solve a mixed integer programming problem. A common approach to solve this problem, using a B&B, is to convert subproblems of the mixed integer problem to linear programming problems, thereby eliminating some of the integer constraints, and then trying to solve these subproblems using an existing linear program approach. In the OpenMP approach of [Barreto 2010], threads share the same pool of subproblems. The obtained speedups were better in the MPI approach than the OpenMP approach. This work highlights the need to avoid misusing control and synchronization mechanisms in order to prevent the performance of an OpenMP B&B from a significant decrease.

2.6 Parallel B&B for computational grids

Large-scale distributed computing environments, usually called computational grids, is a hardware and software infrastructure that gather computers, storage systems and other devices that provide consistent and pervasive access to high-end computational capabilities [Foster 1998]. Grid applications are distinguished from traditional client-server applications by their simultaneous use of dynamic large pooling of resources from multiple administrative domains and complex communication structures.

2.6.1 Challenging issues

As defined in [Baker 2002], a computational grid is a collection of loosely-coupled, geographically distributed, heterogeneous computing resources that can provide significant computing power over long time periods. It is hence defined according to four main characteristics:

- Multi-domain of administration: resources of computational grids are distributed among multiple sets of administration domains which are managed according to different administration organizations. For example, security policies change from one site to another and enabling secure communications between multiple sites typically through firewalls might turn-out to a challenging issue.

- Heterogeneity of the resources: a grid can integrate hardware from multiple vendors, run various operating systems, and use different network protocols for remote communication, etc. In contrast, a single site of the grid is often composed of homogeneous resources, mostly aggregated into clusters.

- Volatility of the resources: a grid is a dynamic environment characterized by its resource's volatility. Indeed, computing resources are not expected to be always available for the application. The availability of the resources is variant due to hardware crash, software issues or any other system variance. Volatility poses several challenging issues such as: dynamic resource discovery, fault tolerance, data recovery, synchronization, etc. These issues are difficult to deal with at a hardware level as they are mainly dependent on the nature of the application being executed. They are instead tackled mostly at the application level.

- Scalability: Due to the number of available computational units and the wide-area network interconnection infrastructure, a computational grid is a large-scale system. The scale of the grid is expressed in terms of network communications and in terms of number of processing cores. Therefore, developers must deal with the scalability issue to ensure a safe running of grid applications mainly as far as communication between resources is concerned.

The characteristics of computational grids lead to a set of challenges when used for parallel optimization algorithms. For example, for B&B algorithms, the volatility and dynamic nature of resources may lead to the loss of one or several subproblem(s) and consequently the loss of one or several optimal solution(s) if no fault tolerance mechanism is used. Besides, distributed grid nodes are localed on distant networks with different bandwidths and connected via shared nation-wide networks. This property induces significant latency overhead in communication between processes running parallel B&B which considerably impacts their efficiency.

2.6.2 Related works

Designing parallel B&B on computational grids had a great interest and has been the subject of several research works. Many investigations of parallel B&B for large-scale systems have adopted the master-worker approach where information is centralized and therefore easily manageable but that lacks scalability. Scalability has been improved through hierarchical (hierarchical master-worker approaches) or fully decentralized organization of processes (Peer-to-Peer approaches). Whereas the work unit distribution differs from one approach to another, all approaches are based on the parallel tree exploration model which is coarse-grained with an important degree of parallelism mainly for large problem instances which justify by itself its relevance to grid computing. In the following, an overview of existing grid-based B&B algorithms are presented and classified according to underlying architecture.

2.6.2.1 Master-worker approaches

Most of applications developed for large scale environments are based on the master-worker paradigm. SETI@home [Anderson 2002] is one of the first large scale master-worker based applications. A central master process distributes computational tasks to workers at the edge of the Internet. When a worker completes its computation, it sends the results to the central master. Work units sharing, communication of the best solution, termination detection and checkpointing mechanisms are centralized within the master's commitment.

In [Mezmaz 2007a], a grid-enabled B&B algorithm based on the master-worker model is proposed. The major contribution of the authors is to propose an efficient encoding of the search space to reduce the size of exchanged messages. The overall parallel efficiency is then improved compared to previous solutions. The approach of [Mezmaz 2007a] can be considered as the best parallel master-worker B&B approach that can be applied in a large scale computational environment such as grids. In particular, it was successfully applied to find the optimal solution of an unsolved FSP hard instance, namely the Ta056 instance.

2.6.2.2 Hierarchical master-worker approaches

In [Aida 2002, Aida 2005], a hierarchical master/worker-based parallel B&B algorithm is proposed and deployed on a grid. This approach is aimed to minimize performance degradation caused by the communication overhead between the master process and worker processes. Processes are organized into sets, each set having a group of worker processes and one master process to coordinate them. In addition, a process called supervisor is in charge of controlling and coordinating all the sets of processes. One set of worker processes explores a given part of the search tree. The supervisor assigns a subset of solutions to the master of the set and this master dispatches the work to its worker processes. This supervisor, as well as the master process of each set of processes, is in charge of gathering and broadcasting the best solution found so far, thus accelerating the exploration process. The latter approach shows a limited scalability as it may create a bottleneck on the master processes and the supervisor process. The authors discuss the granularity of tasks, notably when tasks are fine-grained, the communication overhead is too high compared to the computation of tasks.

The granularity issue is studied in [Diconstanzo. 2007] but yet a hierarchical master/worker model is used therein. The architecture of the approach, named Grid'BnB is quite similar whereas the communication layer is a bit different. It is composed of four different types of entities: master, sub-master, worker, and leader. The master has the same role as the supervisor in the previous approach. The sub-master is in charge of coordinating one set of worker processes. Each set of processes comprises several workers and is deployed on a physical cluster. The cluster running the master process also hosts the sub-master processes which are in charge of communicating with other clusters. In those other clusters, one worker is chosen to be the leader. It is given a specific role, which is to handle communications with its sub-master. Thus, when a worker discovers a new best solution for the problem being solved, it broadcasts it to all the workers belonging to the same cluster, including the leader. This leader sends it to its sub-master process, which broadcasts it to the whole network through the master process.

In [Bendjoudi 2012], the authors suggest a hierarchical architecture to allow workers to communicate directly together after receiving a task from the master. Bendjoudi *et al.* [Bendjoudi 2013] have also proposed a fault tolerant adaptive hierarchical B&B, named FTH-B&B, in order to deal with the fault tolerance and scalability issues in large scale unreliable environments. Their algorithm is composed of several fault tolerant master/worker-based B&Bs, organized hierarchically.

2.6.2.3 Peer-to-Peer approaches

Parallel applications designed under the master/worker paradigm may often face scalability issues due to bottlenecks on the master. To overcome this limitation, some works consider the use of the Peer-to-Peer (P2P) paradigm for parallel B&B as in DIB by Finkel *et al.* [Finkel 1987] and in the work proposed by Iamnitchi *et al.* in [Iamnitchi 2000]. The latter proposes a fully decentralized approach for the B&B algorithm. The role of each process is to manage a local work pool and share it with other processes whenever they receive a request. The best solution is broadcast over the network through the most frequently sent messages. By distributing the communication load among multiple processes, this approach gains in terms of scalability.

Instead of dealing with the scalability issues at the applicative level, Di Constanzo *et al.* [Diconstanzo. 2007] proposed a more generic approach operating at the communication layer. The approach consists in defining a fully P2P infrastructure and providing an information routing mechanism. Processes are organized in a P2P fashion: one of them acts as the master and all others as workers. Whenever a worker needs to communicate with the master, its messages are routed/relayed by multiple peers before reaching the master. While this approach enables to provide a better scalability, it only distributes the communication load of the master through time and introduces additional delays for processing slaves's requests.

In order to overcome the limits of B&B@Grid [Mezmaz 2007a] in terms of scalability, Djamai *et al.* [Djamai 2011] designed a new fully Peer-to-Peer approach for the B&B algorithm parallelization. The approach is able to handle a number of nodes which is exponentially higher than a classical master/worker one. It provides a new work sharing mechanism where each peer shares its interval with its neighbors on request, avoiding redundancy during the tree exploration. It provides a distributed termination detection mechanism which detects the presence (or absence) of a work unit somewhere in the network by counting the number of unsuccessful requests broadcast to its neighbors.

2.7 Conclusion

In this chapter, we provided an overview on the topics related to the context of this thesis. First, we presented the combinatorial problem considered as case study in this thesis which is the Flowshop Scheduling Problem. Thereafter, a comprehensive description of serial B&B algorithms is given. Afterward, a summary of the different classifications of parallel B&B conducted in the literature is presented and an overview of the existing work dealing with parallel B&B classified by the target computing platform (multi-threaded many-core processors, shared memory multi-core systems, computational grids) is made. For each considered architecture, related challenges are identified and discussed and related works are presented.

Chapter 3

GPU-accelerated parallel bounding applied to FSP

Contents

Main publications related to this chapter

I.Chakroun, M.Mezmaz, N. Melab, and A.Bendjoudi.
Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm.

Concurrency and Computation: Practice and Experience - John Wiley & Sons, 2012.

N. Melab, I. Chakroun, M. Mezmaz and D.Tuyttens.
A GPU-accelerated Branch and Bound Algorithm for the Flow-Shop Scheduling Problem.
14th IEEE International Conference on Cluster Computing, CLUSTER'12. China, Beijin, September 24-28, 2012.

I. Chakroun, A. Bendjoudi and N. Melab.
Reducing Thread Divergence in GPU-based B&B applied to the Flow-Shop Problem.
In the LNCS Proceedings of 9th International Conference on Parallel Processing and Applied Mathematics (PPAM), Torun, Poland, September 9-11, 2011.

3.1 Introduction

The objective of this chapter is to present the design and implementation of the GPU-accelerated parallel bounding we propose for B&B algorithms. We particularly identified two main challenges raised when revisiting the design and implementation of the parallel evaluation of FSP lower bounds on GPU devices. The focus is first put on the thread divergence issue related to the SIMD execution model of the GPU. Afterward, the optimization of the access pattern to the hierarchy of GPU memories is addressed.

The remainder of this chapter is structured as follows: In Section 3.2 the FSP lower bound used in this thesis is presented and analyzed. Section 3.3 presents the design of the proposed GPU-accelerated B&B based on the parallel evaluation of lower bounds. Section 3.4 highlights the scenarios where the thread divergence occurs in the studied FSP lower bound, presents a review of different works of the literature for reducing thread divergence and details the mechanisms we propose to reduce the number of divergent branches within a warp. Section 3.5 explains the memory access optimizations we recommend for the data structures used in the lower bound kernel. Finally, in Section 3.6 details about the performed experimental study (the used experimental metrics, the experimented problem instances, etc.) are given and the obtained results are discussed.

3.2 Lower Bound for FSP

The bounding operator provides a lower bound (LB) for each subproblem generated by the branching operator. The more accurate the bound is, the more it allows to eliminate unfruitful nodes from the search tree. Therefore, the efficiency of a B&B algorithm depends strongly on the quality of its lower bound function. In this work, we use the lower bound proposed by Lenstra *et al.* [Lenstra 1978] for FSP, based on the Johnson's algorithm [Johnson 1954].

The Johnson's algorithm allows to solve optimally FSP with two machines ($m = 2$) using the following transitive rule $\preceq$:

$$J_i \preceq J_j \Leftrightarrow \min(p_{i,1} \ ; \ p_{j,2}) \leq \min(p_{i,2} \ ; \ p_{j,1})$$

We recall that $p_{k,l}$ designates the processing time of the job J_k on the machine M_l. From the above rule, it follows the Johnson's theorem:

Jonhson's theorem *Given P an FSP with $m = 2$, if $J_i \preceq J_j$ there exists an optimal schedule for P in which the job J_i precedes the job J_j.*

According to Johnson's theorem, FSP with $m = 2$ is solved with a time complexity of $O(n.logn)$. The optimal solution is obtained by first sorting in increasing order the jobs having a processing time shorter on the first machine than on the second one. Second, sorting in decreasing order the jobs having a shorter processing time on the second machine. In practice, Johnson's algorithm assumes to first identify the job with the smallest processing time (on either machine). If the smallest processing time involves machine 1, to schedule the job at the beginning of the permutation, else (the smallest processing time involves machine 2) to schedule the job at the end of the permutation.

In [J.R.Jackson 1956] and [L.G.Mitten 1959], the Johnson's rule has been extended by Jackson and Mitten with lags which allowed further Lenstra *et al.* to propose a lower bound for FSP with $m \geq 3$. A lag l_j designates the minimum duration between the starting time of the job J_j on the second machine and its finishing time on the first machine. Jackson and Mitten demonstrated that the optimal solution for FSP with $m = 2$ can be obtained using the following transitive rule $\preceq$:

$$J_i \preceq J_j \Leftrightarrow \min(p_{i,1} + l_i \ ; \ l_j + p_{j,2}) \leq \min(l_i + p_{i,2} \ ; \ p_{j,1} + l_j)$$

Based on this rule, Lenstra *et al.* [Lenstra 1978] have proposed the following lower bound for a subproblem associated to a partial schedule where a set $\jmath$ of jobs have to be scheduled on m machines. $P^*_{Ja}(\jmath, M_k, M_l)$ represents the Jackson-Mitten optimal solution for the subproblem that consists in scheduling the set $\jmath$ of jobs on the two machines M_k and M_l. The term $r_{i,k} = \sum_{l<k} p_{i,l}$ designates the starting time of the job J_i on the machine M_k. The other term $q_{j,l} = \sum_{k>l} p_{j,k}$ refers to the latency between the finishing time of J_j on M_l and the finishing time of the schedule.

$$LB(\jmath) = \max_{1 \le k < l \le m} \{P^*_{Ja}(\jmath, M_k, M_l) + \min_{(i,j) \in \jmath^2, i \ne j} (r_{i,k} + q_{j,l})\}$$

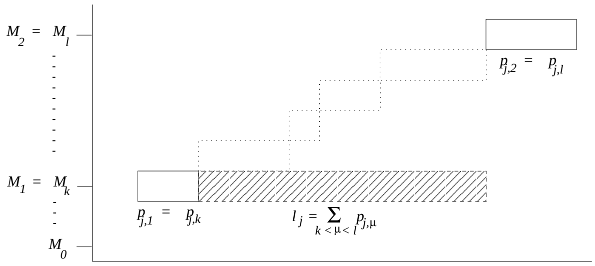


Figure 3.1: The lag l_j of a job J_j for a couple (k, l) of machines is the sum of the processing times of the job on all the machines between k and l.

According to this LB expression, the lower bound for the scheduling of a subset $\jmath$ of jobs is calculated by applying the Johnson's rule with lags considering all the couples (k, l) for $1 \le k, l \le m$ and $k < l$. As illustrated in Figure 3.1, the lag l_j of a job J_j for a couple (k, l) of machines is the sum of the processing times of the job on all the machines between k and l.

3.3 A GPU-accelerated B&B based on the parallel evaluation of bounds (GB&B)

As said previously, the time complexity of the Jonhson's algorithm for two machines is $O(n.logn)$. Therefore, the time complexity of the lower bound LB for m machines is $O(m^2.n.logn)$. The computation of the lower bound is consequently time intensive especially for problem instances for which m is high.

In order to experimentally evaluate its CPU time cost, we performed a preliminary study where we measure the execution time of each operator of the algorithm. Therefore, we carried out experiments on some standard's FSP instances, commonly known as Taillard's problem instances [Taillard 1993b, Taillard 1993a]. A serial version of the B&B algorithm is run during 600 seconds.

Table 3.1 reports the durations of the bounding operator of the B&B for several instances of 20 machines and a number of jobs ranging from 20 to 200. The results show that the bounding operator that calculates a lower bound for each subproblem, is the most costly operation in the B&B algorithm taking on average between 97% and 99% of its total execution time. Such result demonstrates the need to parallelize the bounding

Type of instances	Bounding (s)	Bounding / B&B (%)
200×20	582,968	97,16%
100×20	591,329	98,55 %
50×20	592,560	98,76 %
20×20	592,785	98,79 %

Table 3.1: Execution time of the bounding operator compared to the execution time of the whole B&B algorithm.

operator. Indeed, not only the parallel evaluation of bounds is akin to data-parallelism but is also a very time consuming operation as shown through the experimentation.

Therefore, we propose a parallel GPU-accelerated approach (GB&B) based on the parallel evaluation of bounds where the generation of the subproblems (pruning, selection and branching operations) to be solved is performed on CPU and the evaluation of their lower bounds (bounding operation) is executed on the GPU device.

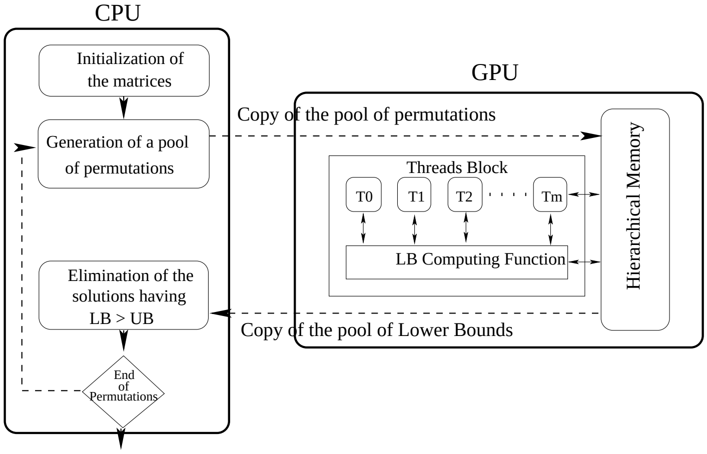

CPU
Initialization of the matrices
Generation of a pool of permutations
Elimination of the solutions having LB > UB
End of Permutations
GPU
Copy of the pool of permutations
Threads Block
T0
T1
T2
Tm
LB Computing Function
Hierarchical Memory
Copy of the pool of Lower Bounds

Figure 3.2: The overall architecture of the GPU-accelerated algorithm based on the parallel evaluation of bounds (GB&B).

As illustrated in Figure 3.2, a pool of subproblems generated on CPU is selected according to a depth-first strategy and off-loaded to the GPU device to be evaluated by a pool of threads partitioned into blocks. Each thread applies the lower bound function to one subproblem. Once the evaluation is completed, the lower bound values corresponding to the different subproblems are returned to the CPU to be used by the pruning operator to decide either to be eliminated or to be decomposed. The process is iterated until the exploration is completed and the optimal solution is found.

The template of the GPU implementation of the parallel evaluation of bounds is given in *Algorithm* 2.

Algorithm 2 Kernel of the parallel computation of the lower bound on GPU.

```
Data: poolToEvaluate = Pool of nodes to be evaluated in parallel.
Result: poolOfBounds = Pool of returned bounds values.
__global__ void evaluateOnGPU(/*parameters*/)
{
int thread_idx = blockIdx.x * blockDim.x + threadIdx.x;
if Is_leaf( poolToEvaluate[thread_idx] ) then
    lower_bound = evaluateSolution(poolToEvaluate[thread_idx].permutation,PTM,
    /*other less important arguments*/);
end
else
    // Some necessary initializations before computing the lower bound
    lower_bound = computeLB(/*parameters*/);
end
poolOfBounds[thread_idx] = lower_bound;
}
```

The parallel evaluation of bounds is a problem-specific parallelism. This feature implies an irregular computation depending on the data of each subproblem which is reflected in several control flow instructions and that conducts to different behaviors. As explained in Section 2.4, when such data-dependent conditional instructions are executed on top of GPU, different behaviors within threads of a same warp occur. This leads to the thread divergence issue and impacts the throughput of the application.

3.4 Thread divergence reduction

This section discusses the thread divergence issue encountered when computing the lower bounds of FSP on GPU, presents a review of the different works that aims at reducing thread divergence and details the mechanisms we propose to reduce the number of divergent branches within a warp.

3.4.1 Problem statement in the FSP lower bound

In the FSP lower bound, thread divergence occurs due to two major factors: the locations of nodes within the search tree and the control flow instructions within the bounding operator.

Divergence related to the control flow instructions. Control flow refers to the order in which the instructions, statements or function calls are executed in a program. This flow is determined by conditional and loop instructions such as *if-then-else*, *for*, *while-do*, *switch-case*, etc. There are a dozen of such instructions in the algorithm of the bounding operator of FSP. The source code examples given below show two scenarios in which this kind of instruction is used.

- Example 1:

```
if( pool[thread_idx].index_start != 0 )
   time = TimeMachines[1] ;
else
   time = TimeArrival[1] ;
```

- Example 2:

```
for(int k = 0 ; k < pool[thread_idx].index_start; k++)
   jobTime = jobEnd[k] ;
```

In these two examples, *thread_idx* is the index associated to the current thread. Let us suppose that the instruction in Example 1 is executed by 32 threads where *pool[thread_idx].index_start* is equal to 0 for the first thread, and *pool[thread_ idx].index_start* is not equal to 0 for the other 31 threads. The first thread will execute the code associated with the "false" condition (*pool[thread_ idx].index_start == 0*) and the other threads will execute the code associated with the "true" condition (*pool[thread_ idx].index_start != 0*). Since the instruction decoder can only handle one branch at a time, these branches cannot be executed concurrently and would be executed in sequence. When the first thread executes the *else* construct, the remaining 31 threads of that warp which are not on the taken path are disabled. The problem in this case is that no other warps are allowed to run on that multiprocessor meanwhile because the warp is not completely idle which impacts the execution times.

The same scenario occurs during the execution of Example 2. Let us suppose that the instruction is executed by 32 threads, *pool[thread_ idx].index_start* is equal to 100 for the first thread, and *pool[thread_ idx].index_start* is equal to 10 for the other 31 threads. In this case, all threads will finish the first 10 iterations together. Two passes are required to execute each of the following iterations, one pass for those that take the iteration and another pass for those that do not.

Divergence related to the location of nodes. Below is given an example from the source code of the used LB showing how the execution flow depends on the position of the node in the search tree. In the following piece of code, three methods are used *is_leaf()*, *makespan()* and *lower_bound()*. *is_leaf()* tests if the node *_node* is a leaf or an internal node. If *_node* is a leaf, *makespan()* computes the cost of its makespan. Otherwise, *_node* is an internal node and *lower_bound()* computes the value of its lower bound.

```
if (_node.is_leaf())
   return _node.makespan();
else
   return _node.lower_bound();
```

3.4.2 Mechanisms for reducing branch divergence

Using GPUs has become increasingly popular in high performance computing. A large number of optimizations have been proposed to improve the performance of GPU programs. The majority of these optimizations target the GPU memory hierarchy by adjusting the pattern of accesses to the device memory [Ryoo 2008, Yang 2010, Sung 2010, Jang 2011]. In contrast, there has been less work on optimizations that tackle another fundamental aspect of GPU performance, namely its SIMD execution model. This section presents some major existing works related to the thread divergence reduction.

Dynamic Warp Formation (DWF) [Fung 2007] is a hardware mechanism proposed in order to improve the efficiency of SIMD branch execution. Every cycle the thread scheduler recomposes warps from the active threads by grouping those that are executing the same path into the same warp. To achieve this, DWF requires an additional hardware that does thread regrouping. Meng *et al.* [Meng 2010] also propose a hardware mechanism. This tool, called Dynamic Warp Subdivision (DWS), splits a warp into sub-warps at divergent branches that can be scheduled independently.

In [Zhang 2010], the proposed approach performs a runtime data remapping across multiple warps. It is proposed that the remapping happens on the fly because of the dependence of thread divergences on runtime values. The major inconvenient with this approach is that it requires a CPU-GPU pipelining scheme, feature that incurs extra host-device communications.

In [Han 2011], the authors intervene at code level and introduce two software-based optimizations: iteration delaying and branch distribution. Iteration delaying improves the utilization of execution units in the presence of a divergent branch within a loop, by executing only one branch path at each iteration and delaying the threads that follow the other path until later iterations. Branch distribution aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths.

The existing techniques for handling branch divergence either demand hardware support [Fung 2007] or require host-GPU interaction [Zhang 2010], which incurs overhead. Some other works such as [Han 2011] intervene at the code level. They expose a branch distribution method that aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths. In our work, we have also opted for software-based optimizations like [Han 2011]. In fact, we figure out how to literally rewrite the branching instructions into basic ones in order to make thread execution paths uniform. We also demonstrate that we could ameliorate performances by appropriately reordering data being assigned to each thread.

In the following, we outline the proposed mechanisms identified with the aim to reduce the number of divergent threads during the execution of the lower found function on GPU. The first thread-data reordering method targets the divergence that is induced by the loop constructs. The second approach called branch refactoring aims at decreasing the number of divergent branches that result from the *if-then-else* instructions.

3.4.2.1 Thread-data reordering

In the B&B applied to FSP, a node from the search tree corresponds to a permutation containing a set of scheduled and unscheduled jobs. The range of scheduled and unscheduled jobs depends on the level of the node in the tree. When computing in parallel the LB function, each thread picks one position from the set of unscheduled jobs, schedules it and computes the corresponding makespan.

A deep analysis of the used lower bound function shows that most of the loop constructs it encloses are related to the range of unscheduled jobs contained in the underlying permutation. Therefore, and because the selected pool that is off-loaded to GPU may contain subproblems from different levels of the tree, threads of a same warp may not finish loop's iterations at the same time and several passes will be used.

The purpose of thread-data reordering is to reduce the number of threads that might diverge on the loop constructs. The idea is to sort the pool of selected subproblems before it is off-loaded to the GPU device according to their position in the tree. As quoted above, this position dictates the range of the unscheduled jobs. This sorting process is used in order to make the pool as homogeneous as possible (selected nodes are from the same level in the tree unless from nearby levels).

3.4.2.2 Branch refactoring

The proposed branch refactoring approach consists in rewriting the conditional instructions so that threads of the same warp execute an uniform code avoiding their divergence. To do so, two major scenarios are studied and optimizations are proposed accordingly. These two scenarios correspond to the conditional instructions contained in the *LB* kernel code but can easily be generalized.

In the first scenario, the conditional expression is a comparison of the content of a variable to 0. The following example extracted from the pseudo-code of the lower bound *LB* illustrates such scenario.

```
if( pool[thread_idx].index_start != 0 )
   time = TimeMachines[1] ;
else
   time = TimeArrival[1] ;
```

The refactoring idea consists in replacing the conditional expression by two functions f and g as explained in Equation 3.1.

$$\begin{array}{l}
\begin{array}{lcl}
if(x \neq 0)\ a = b[1]; & & if(x \neq 0)\ a = b[1] + 0 \times c[1]; \\
 & \Rightarrow & \\
else\ \ a = c[1]; & & else\ \ a = 0 \times b[1] + c[1];
\end{array} \\
\\
\Rightarrow a = f(x) \times b[1] + g(x) \times c[1]; \\
\\
where: \\
\quad f(x) = \begin{cases} f(x) = 0 & if \quad x = 0 \\ 1 & else \end{cases} \\
and \\
\quad g(x) = \begin{cases} g(x) = 1 & if \quad x = 0 \\ 0 & else \end{cases}
\end{array} \tag{3.1}$$

The behavior of f and g perfectly fits the cosine trigonometric function. These functions return values between 0 and 1. An integer variable is used to store the result of the cosine function. Its value is 0 or 1 since it is rounded to 0 if it is not equal to 1. In order to increase the performances, runtime math operations are used: $sinf(x)$, $expf(x)$ and so forth. Those functions are mapped directly to the hardware level. They are faster but provide lower accuracy which does not matter in our case because the results are rounded to *int*. For NVIDIA GPUs, the throughput of $sinf(x)$, $cosf(x)$, $expf(x)$ is one operation per clock cycle [NVIDIA Corporation 2011a].

The refactoring result for the “if” pseudo-code given above is the following:

```
int coeff = cosf (pool[thread_idx].index_start);
time = (1 - coeff) * TimeMachines[1] +  coeff * TimeArrival[1];
```

The second “if” scenario considered compares two values between themselves as shown in Equation 3.2.

$$
\begin{aligned}
& if(x \gnsim y) \quad a = b[1]; \quad \Rightarrow \quad if(x - y \geq 1) \quad a = b[1]; \\
\Rightarrow \quad & if(x - y - 1 \geq 0) \qquad a = b[1]; \quad (x, y) \in N \\
\Rightarrow \quad & a = f(x, y) \times b[1] + g(x, y) \times a; \\
& \textit{where:} \qquad \text{f(x,y)} = \begin{cases} 1 & if \quad x - y - 1 \geq 0 \\ 0 & if \quad x - y - 1 < 0 \end{cases} \\
& \textit{and} \qquad \text{g(x,y)} = \begin{cases} 0 & if \quad x - y - 1 \geq 0 \\ 1 & if \quad x - y - 1 < 0 \end{cases}
\end{aligned}
\tag{3.2}
$$

For instance, the following example extracted from the pseudo-code of the lower bound LB illustrates such scenario.

```
 if ( TimeArrival[1] > min )
    Best_idx = Current_idx;
```

The same transformations as those employed for the first scenario are applied here using the exponential function. The exponential is a positive function which is equal to 1 when applied to 0. Thus, if x is less than y $\_\_expf(x - y - 1)$ returns a value between 0 and 1. If this result is rounded to an integer value 0 is obtained. Now, if x is greater than y $\_\_expf(x - y - 1)$ returns a value greater than 1 and since the minimum between 1 and the exponential is considered, the returned result would be 1. This behavior exactly satisfies our prerequisites.

The above “if” instruction pseudo-code is now equivalent to:

```
int coeff = min(1, __expf(TimeArrival[1] - min - 1));
Best_idx = coeff * Current_idx + ( 1 - coeff ) * Best_idx ;
```

As explained in Section 2.4, memory access optimization is a key enabler for improving GPU-based application performances. Indeed, many constraints are related to the use of the hierarchy of memory available on GPU architectures. For the FSP lower bound kernel, several data structures are used with different sizes and access rates. Our objective in the following section is to identify the best mapping of the LB data structures which should use as much fast memory and as little slow-access memory as possible.

3.5 Data placement optimization for the FSP lower bound

In this section, optimizing the data access pattern of the proposed parallel bounding approach is investigated. We particularly discuss how our approach maps the different data structures on the memory hierarchy of the GPU device taking into account the characteristics of the data structures and those of the different GPU memories.

3.5.1 Complexity analysis and implementation

In this section, the characteristics of the data structures used by the lower bound function presented in Section 3.2 are studied in terms of size and access frequency.

For an efficient implementation of the LB, illustrated in Figure 3.3, six data structures are required: the matrix PTM of the processing times of the jobs, the matrix of lags LM, the Johnson's matrix JM, the matrix RM of the earliest starting times of jobs, the matrix QM of their lowest latency times and the matrix MM containing the couples of machines.

In the LB expression (see Section 3.2), the computation of the term $P^*_{Ja}(\jmath, M_k, M_l)$ requires the calculation of the lag of each remaining job to be scheduled on the couple (M_k, M_l) of machines using its processing times on these machines (Johnson's rule with lags). Such computation is repeated for each couple (M_k, M_l) of machines with $1 \leq k, l \leq m$ and $k < l$. To avoid the repetitive computation of the lags, they are computed once at the beginning of the algorithm and stored in the matrix LM. The dimension of LM is $n \times \frac{m\times(m-1)}{2}$, where n and m are respectively the number of jobs to be scheduled and m the number of machines. LM is accessed $n' \times \frac{m\times(m-1)}{2}$ times, n' being the number of remaining jobs to be scheduled in the subproblem for which the lower bound is being calculated. The processing times of all the jobs on all the machines are stored in the matrix PTM. This matrix has a dimension of $n \times m$ and is accessed $n' \times m \times (m-1)$ times.

In order to avoid relaunching the Johnson's algorithm for each couple of machines and

```
int computeLB(){
  LB=maxInteger;
  for (index=0;index< $\frac{m\times(m-1)}{2}$;index++){
     M1=MM[index][0];
     M2=MM[index][1];
     timeOnM1= $\min_{0\leq j\leq n}$(RM[M1][j]);
     timeOnM2= $\min_{0\leq j\leq n}$(RM[M2][j]);
     for (i=0;i<n;i++){
        job=JM[i][index];
        if (job not yet scheduled){
           timeOnM1=timeOnM1+PTM[job][M1];
           if (timeOnM2>timeOnM1+LM[job][index])
              timeOnM2+=PTM[job][M2];
           else
              timeOnM2=timeOnM1+LM[job][index]+PTM[job][M2];
        }
     }
     timeOnM2+= $\min_{0\leq j\leq n}$(QM[M2][j]);
     LB=max(timeOnM2,LB);
  }
  return LB;
}
```

Figure 3.3: Pseudo-code implementing the LB function

each subset of jobs, the Johnson's algorithm is computed once to find the optimal solutions on the couples of machines. These optimal solutions are then stored in the Johnson's matrix JM. This matrix has the same dimension as LM and is accessed $n \times \frac{m\times(m-1)}{2}$ times during the computation of the lower bound.

To reduce the computation time cost of the term $\min_{(i,j)\in J^2, i\neq j}(r_{i,k} + q_{j,l})$ in the LB expression, two matrices are defined, namely RM and QM. They are used to store respectively the lowest starting and latency times of all the jobs on each machine. Their dimension is m and are accessed $m \times (m-1)$ times and $\frac{m\times(m-1)}{2}$ times respectively. Finally, the MM matrix that contains all the couples of machines has a dimension and access frequency of $m \times (m-1)$.

The complexities in terms of size and access frequency of the different data structures are summarized in Table 3.2 where the columns represent respectively the name of the data structure, its size and the number of times it is accessed.

Matrix	**Size**	**Number of accesses**
PTM	$n \times m$	$n' \times m \times (m-1)$
LM	$n \times \frac{m\times(m-1)}{2}$	$n' \times \frac{m\times(m-1)}{2}$
JM	$n \times \frac{m\times(m-1)}{2}$	$n \times \frac{m\times(m-1)}{2}$
RM	m	$m \times (m-1)$
QM	m	$\frac{m\times(m-1)}{2}$
MM	$m \times (m-1)$	$m \times (m-1)$

Table 3.2: The different data structures of the LB algorithm and their associated complexities in memory size and numbers of accesses. The parameters n, m and n' designate respectively the total number of jobs, the total number of machines and the number of remaining jobs to be scheduled for the subproblems for which the lower bound is being computed.

3.5.2 Data placement pattern of the lower bound on GPU

When identifying the best mapping of the data structures on the memory hierarchy of the GPU device, our focus is put on the shared memory which is a key enabler for many high-performance GPU applications. Indeed, because it is on-chip, shared memory has much higher bandwidth and lower latency than local and global memory. However, for large problem instances (large n and m) the data structures especially JM and LM, do not fit in the shared memory for some GPU configurations. In order to achieve further performances, we also take care of adequately use the global memory by judiciously configuring the L1 cache which greatly enables improving performance over direct access to global memory.

Table 3.3 reports the sizes of each data structure for different Taillard's problem instances [Taillard 1993a, Taillard 1993b]. The sizes are given in number of elements and in bytes (between brackets).

Taking into consideration the sizes of each data structure presented in Table 3.3, our challenge is to find which data structure has to be mapped on which memory and for large problem instances how to split the data structures on different memories and efficiently manage their accesses. The sizes in bytes reported in Table 3.3, are computed knowing that in our implementation the elements of JM and PTM are `unsigned char` (one byte)

Nb.Jobs × Nb.machines	JM	LM	PTM	RM, QM	MM
200 × 20	38.000 (38KB)	38.000 (76KB)	4.000 (4KB)	20 (0.04KB)	380 (0.76KB)
100 × 20	19.000 (19KB)	19.000 (38KB)	2.000 (2KB)	20 (0.04KB)	380 (0.76KB)
50 × 20	9.500 (9.5KB)	9.500 (19KB)	1.000 (1KB)	20 (0.04KB)	380 (0.76KB)
20 × 20	3.800 (3.8KB)	3.800 (7.6KB)	400 (0.4KB)	20 (0.04KB)	380 (0.76KB)

Table 3.3: The sizes of each data structure for the different experimented problem instances. The sizes are given in number of elements and in bytes (between brackets).

and that the elements of LM, RM, QM and MM are `unsigned short int` (2 bytes). It is important here to highlight that the types of the data of the used matrices impact the size of each matrix. For instance, a matrix of 100 integers has a size of 400 bytes while the same matrix with 100 unsigned chars has a size of 100 bytes. In order to minimize the size of each of the used matrices, we analyzed the ranges of their values and defined their data types accordingly. For instance, in PTM all the processing times have positive values varying between 0 and 100. Therefore, we defined PTM as a matrix of `unsigned char` having values in the range $[0, 255]$. Using the `unsigned char` type instead of the integer type allows us to reduce by 4 times the memory space occupied by PTM. According to Table 3.3:

- The data structures RM, QM and MM are small sized matrices. Therefore, their impact on the performances is not significant whatever is the memory they are offloaded to. Indeed, preliminary experiments prove that putting them on the shared memory allows a very poor performance improvement.

- The LM data structure is the double of the JM in memory size but with a much lower access frequency. Therefore, it is better to map JM on the shared memory.

- The PTM has almost the same access frequency than JM but requires less memory space.

Consequently, the focus is put on the study of the performance impact of the placement of JM and PTM on the shared memory. Three placement scenarios of JM and PTM are experimented and studied: (1) Only PTM is stored in shared memory and all others are placed on global memory; (2) Only JM is stored in shared memory and all others are placed on global memory; (3) PTM and JM are stored together in shared memory and all others are placed on global memory.

3.6 Experiments

In this section, we introduce the protocol used for the experiments presented in the rest of the document. The problem instances experimented for the different proposed approaches, the hardware configurations and the several experimental metrics are also detailed.

3.6.1 Experimental settings and parameters tuning

In the following, details are given about the experimented problem instances, the used experimental hardware and the experimental metrics.

3.6.1.1 Experimental settings

To evaluate the performance of the proposed GPU-based parallel approaches, we have considered the Taillard's FSP benchmarks proposed in [Taillard 1993a, Taillard 1993b]. These standard instances are often used in the literature to evaluate the performance of methods that minimize the makespan. Optimal solutions of some of these instances are still not known. These instances are divided into groups of 10 instances. In each group, the 10 instances are defined by the same number of jobs and the same number of machines. The groups of 10 instances have different numbers of jobs, namely 20, 50, 100, 200 and 500, and different numbers of machines, namely 5, 10 and 20. For example, there are 10 instances with 200 jobs and 20 machines belonging to the same group of instances. In our experiments, only the instances where the number of machines is equal to 20 and the number of jobs equal to 20, 50, 100, 200 are used. Indeed, instances where the number of machines is equal to 5 or 10 are easy to solve. For these instances, the used bounding operator gives so good lower bounds that it is possible to solve them in few minutes using a serial B&B. In particular, preliminary experiments have shown that using a single CPU-based B&B performs better than a GPU-accelerated B&B for the instances with 5 machines and that small accelerations (on average around 11) are obtained with 10 machines with the GPU-based B&B compared to the serial version of the algorithm. Therefore, these instances do not require the use of a GPU.

In our experiments, we also omit instances with 500 jobs because they do not fit in the memory of the GPU. Indeed, as shown in Table 3.4, the size of the data structures (data used for the computing the FSP lower bound, intermediate structures where the subproblems are generated, etc.) used for these category of instances are significant and reaches the limit size dedicated by the GPU global memory. Indeed, because ECC is on [NVIDIA Corporation 2010], the size of the provided global memory is decreased by 12.5% and is only about 2.45 GB. Nevertheless, the 2.3 GB of data used for the instances with 500 jobs can't be accomodated since the GPU devote some segments of the global memory to accomodate the kernel instructions which are stored in global memory that is inaccessible to the user but are prefetched into an instruction cache during execution.

Problem instances	Size of the used data (MB)
20 × 20	126.04
50 × 20	261.08
100 × 20	486.16
200 × 20	936.32
500 × 20	2286.80

Table 3.4: Size of the data structures used the by each group of instance.

The different approaches we propose have been implemented using C-CUDA 4.0. The experiments have been carried out using an Intel Xeon E5520 bi-processor coupled with a GPU device. The bi-processor is 64-bit, quad-core and has a clock speed of 2.27GHz. The GPU device is an Nvidia Tesla C2050 with 448 CUDA cores (14 multiprocessors with 32 cores each), a clock speed of 1.15GHz, a 2.8GB global memory, a 49.15KB configurable shared memory, and a warp size of 32.

3.6.1.2 Tuning the number of blocks and number of threads

A kernel function running on a GPU is generally tuned by two leading parameters: the number of threads per block N and the total number of threads S. Tunning the number of active threads is, in fact, a key point to maximize hardware utilization. For the GB&B approach, the execution of the lower bound on GPU assumes that each thread is associated to one subproblem; each thread applies the lower bound function to one subproblem. Therefore, when the size of the pool off-loaded to GPU is equal to N×S, N×S threads should be triggered at the kernel launching.

Preliminary experiments, reported in Table 3.5, show the comparison between average execution time for all the instances obtained with different numbers of blocks and block sizes. The first column represents the size of the pool off-loaded to the GPU. The other columns give the corresponding number of blocks, number of threads per block and average normalized execution time. The average normalized execution times are calculated for all instances with 20, 50, 100 and 200 jobs over 20 machines. For each row, the execution times are normalized and divided by the execution time obtained with the pair (number of blocks × number of threads per block) given the same pool size and having the lower number of blocks. For instance, for a pool size of 4096, all the execution times are divided by the execution time obtained using 16 blocks and 256 threads per block. For this pool size, the obtained execution time using 32 blocks and 128 threads per block is almost half (54%) of the execution time obtained using 16 blocks and 256 threads per block.

During the tuning process, the primary concern when choosing the number of blocks per grid was keeping the entire GPU busy. Indeed, this parameter should be larger than the number of multiprocessors of the used device so that all multiprocessors have at least one block to execute. Thus, the number of blocks is first initialized as the nearest power of 2 from the number of the multiprocessors detected. Namely on the C2050 GPU used card, there are 14 multiprocessors so we started the number of blocks from 16.

Pool size	**#Blocks × #Threads**	16×256	**32×128**	64×64	128×32	256×16		
4096	**Average normalized time**	1	**0.547**	0.579	0.762	0.859		
Pool size	**#Blocks × #Threads**	16×512	**32×256**	64×128	128×64	256×32	512×16	
8192	**Average normalized time**	1	**0.503**	0.524	0.606	0.722	0.808	
Pool size	**#Blocks × #Threads**	16×1024	32×512	**64×256**	128×128	256×64	512×32	1024×16
16384	**Average normalized time**	1	0.523	**0.494**	0.549	0.600	0.733	0.813
Pool size	**#Blocks × #Threads**	32×1024	64×512	**128×256**	256×128	512×64	1024×32	
32768	**Average normalized time**	1	0.948	**0.898**	0.969	1.083	1.336	
Pool size	**#Blocks × #Threads**	64×1024	128×512	**256×256**	512×128	1024×64		
65536	**Average normalized time**	1	0.924	**0.864**	0.959	1.073		
Pool size	**#Blocks × #Threads**	128×1024	**256×512**	512×256	1024×128			
131072	**Average normalized time**	1	**0.939**	1.117	1.128			
Pool size	**#Blocks × #Threads**	256×1024	512×512	**1024×256**				
262144	**Average normalized time**	1	0.922	**0.866**				

Table 3.5: Average normalized execution times as a function of the number of blocks and the number of threads per block.

The results show that the worst execution times are always obtained with a number of blocks equal to 16. As quoted above, with less than 16 blocks some of the multiprocessors of the device are idle. With 16 blocks all the multiprocessors are used, however there is only one block per multiprocessor which does not allow to hide the latency of the memory accesses. With more than 16 blocks the speed scales better. The results also show that for all the pool sizes except 4096 a block size equal to 256 gives the best results. Therefore, in the rest of the document, the block size (i.e. the number of threads per block) is equal to 256 in all our experiments. The experimentally found best value for the block size (i.e. 256) has been consolidated using the CUDA occupancy calculator provided by NVIDIA

[NVIDIA Corporation 2008]. This tool allows one to easily calculate the best block size based on register and shared memory usage of the kernel.

3.6.2 Experimental protocol

To assess the performances of the different proposed approaches presented in this document, we calculate their speedups. These speedups are obtained by comparing the GPU-accelerated B&B versions to a serial B&B version deployed on a single CPU core. However, most of the Taillard's FSP instances used in our experiments are extremely hard to solve. Indeed, the resolution of these instances requires several months of computation on one CPU core and the optimal solutions of many of these instances are still unknown. For example, the optimal solution of one of these instances defined by 50 jobs and 20 machines has been obtained after 25 days of computation using an average of 328 CPU cores [Mezmaz 2007a, Mezmaz 2007b].

Employing the method defined in [Mezmaz 2007a, Mezmaz 2007b] allows to obtain a random list L of subproblems such that the resolution of L lasts $Tcpu$ minutes when the serial B&B algorithm is initialized by (1) this list L and (2) the optimal solution of L. If these two conditions are met, then, for all exploration strategies, (1) the serial B&B algorithm always explores the same subproblems, and (2) the resolution time of this serial algorithm is always the same regardless of the used strategy. To ensure that the subproblems explored by the GPU and CPU B&B versions are exactly the same, we initialize the pool of our GPU-based B&B with the same list L of subproblems used in the serial version. If we assume the resolution of the GPU-based B&B lasts $Tgpu$ seconds, the reported speedup of the algorithm will be equal to $Tcpu/Tgpu$.

In summary, to find the speedup associated to an instance, we:

- compute, using the approach defined in [Mezmaz 2007a, Mezmaz 2007b], a list L of subproblems such that the resolution of L lasts $Tcpu$ minutes with a serial B&B,
- initialize the serial B&B with the subproblems of this list L,
- explore the subproblems of L with the serial B&B and get the number of explored subproblems $Ncpu$,
- initialize the GPU-based B&B with the subproblems of the list L,
- explore the subproblems of L with the GPU B&B,
- get the GPU resolution time $Tgpu$ and the number of explored subproblems $Ngpu$,

- check that $Ngpu$ is exactly equal to $Ncpu$,
- and finally compute the speedup associated to this instance by dividing $Tcpu$ on $Tgpu$ (i.e. $Tcpu/Tgpu$).

Instance (No. of jobs x No. of machines)	20×20	50×20	100×20	200×20
Sequential resolution time (minutes)	10	50	150	300

Table 3.6: The serial resolution time of each instance according to its number of jobs and machines

Table 3.6 gives, for each instance according to its number of jobs and its number of machines, the used resolution time $Tcpu$ with a serial B&B. For example, the serial resolution time of each instance defined with 20 jobs and 20 machines is approximately 10 minutes. Let us recall that the computation time of the lower bound of a subproblem defined with 20 jobs and 20 machines is on average less than the computation time of the lower bound of a subproblem defined with 50 jobs and 20 machines. Therefore, as shown in Table 3.6, the serial resolution time increases with the size of the instance in order to be sure that the number of subproblems explored is significant for all instances.

In the following section, an experimental study is presented with the objective to evaluate the performance impact of the GPU-based parallel evaluation of the lower bounds, the thread divergence reduction mechanisms and the data access optimization. For each, we present the objectives of the experiments and report the obtained results.

Two parameters are considered: the problem instances ($n \times m$) (as rows in the tables) and the size of the pool of subproblems to be evaluated (as columns in the tables). The first parameter gives information on the granularity of the thread computations. As the complexity of the computation of the lower bound is $O(m^2$ n. log n), for large problem instances (i.e. large values of n and m) the grain size of the kernel executed by each thread is much higher. The second parameter is useful to get information on the time cost of the data transfer between CPU and GPU and on the total number of threads to be triggered on GPU. For each couple of values associated to the two former parameters (the problem instances and the size of the pool), each table/graphics reports the average speedup corresponding to each group of 10 instances defined by the same number of jobs and the same number of machines and to each pool size.

3.6.3 Performance Evaluation of the GB&B

In this section, we experiment the effectiveness of the parallelization of the bounding operator in a B&B algorithm on top of a GPU device. The objective here is to demonstrate that the use of GPU for evaluating in parallel a selected pool of subproblems allows one to significantly accelerate the execution of the B&B.

The obtained experimental results are reported in Table 3.7. The results show that evaluating in parallel the bounds of a selected pool, allows one to significantly speedup the execution of the B&B. Indeed, an acceleration factor up to 71.69 is obtained for the 200 × 20 problem instances using a pool of 262.144 (1024 blocks × 256 threads per blocks) subproblems. The results show also that the parallel efficiency grows with the size of the problem instance. For a fixed number of machines (here 20 machines) and a fixed pool size, the obtained speedup grows accordingly to the number of jobs. For instance, for a pool size of 262.144, the acceleration factor obtained with 200 jobs (×71.69) is almost the double of the one obtained with 20 jobs (×38.40). This behavior is due to the complexity of the computation of the lower bound which is $O(m^2.n.logn)$. For large problem instances (i.e. large values of n and m) the grain size of the kernel executed by each thread is much higher which increases the GPU throughput.

Problem instance Pool size	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	43.83	58.23	59.68	61.21	66.75	68.30	**71.69**
100×20	42.59	57.18	58.53	59.95	60.52	65.70	**65.97**
50×20	41.57	**56.15**	55.69	55.49	55.39	55.27	55.14
20×20	38.74	**46.47**	45.37	41.92	39.55	38.90	38.40
Total average speedup	41.68	54.5	54.81	54.89	55.5	57.04	57.80

Table 3.7: Speedups for different problem instances and pool sizes.

As far the pool size tuning is considered, we could notice that whatever the FSP instance is, the pool size has an important impact on the performance of a GPU-based B&B applied to FSP. Indeed, for a fixed number of machines and a fixed number of jobs, the obtained speedup differ accordingly with the size of the pool being off-loaded to the GPU. For instance for a pool size of 262.144, the acceleration factor obtained with 200 jobs is about 40% higher than the speedup reached with a pool size of 4096 subproblems. The results show also that this parameter (the pool size) depends strongly on the problem instance being solved. For example, for the instances with 50 jobs and 20 machines, the best speedup is obtained with a pool size of 8192 subproblems while the best acceleration is reached with a pool size of 262.144 nodes for the instances with 100 jobs over 20 machines. The best size of the pool is thus hard to be fixed *a priori* and so has to be tuned dynamically with respect to the problem being solved.

3.6.4 Performances of the thread reduction approaches

The objective of the experimental study presented in this section is to evaluate the performances of the proposed techniques (the data reordering method and the branch refactoring mechanism) for reducing the thread divergence.

3.6.4.1 Impact of the data reordering technique

In the following, we study the impact of sorting the pool to be transferred to the device on the performance of the GPU-accelerated B&B. The results, reported in Table 3.8, show that reordering data makes the kernel running fast with a homogeneous pool than with an unordered pool. Indeed, the approach improves the GPU acceleration compared to the results reported in Table 3.7 whatever the instance and the pool size are. This is expected since assembling the subproblems according to their level in the tree and thus their set of unscheduled jobs (see Section 3.4.2) allows one to reduce the impact of the loop instructions that depend on these values.

Problem instance Pool size	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	44.04	59.67	60.13	63.10	68.94	71.23	**74.20**
100×20	43.93	57.93	59.01	60.95	62.47	66.30	**66.66**
50×20	42.58	**57.26**	56.81	56.73	56.54	56.28	55.93
20×20	39.92	**48.58**	47.53	44.72	41.14	40.63	40.59
Total average speedup	42.61	55.86	55.88	56.37	57.27	58.61	59.35

Table 3.8: Speedups for different problem instances and pool sizes using a sorted pool.

3.6.4.2 Evaluation of the branch refactoring mechanism

The objective here is to demonstrate that the thread divergence reduction mechanism based on branch refactoring has an impact on the performance of the GPU-accelerated B&B and to evaluate how this impact is significant. Table 3.9 shows the experimental results obtained using the refactoring approach presented in Section 3.4.2. The results show that the refactoring-based optimizations accentuate the GPU acceleration reported in Table 3.7 and Table 3.8 and obtained without thread divergence reduction. For example, for the instances of 200 jobs over 20 machines and a pool size of 262.144, the average reported speedup is ×77.46 while the average acceleration factor obtained without thread divergence management for the same instances and the same pool size is ×74.20 which corresponds to an improvement of 4.21%. Such considerable but not outstanding improvement is predictable, as claimed in [Han 2011], since the factorized part of the branches in the FSP lower bound is very small.

Problem instance Pool size	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	46.63	60.88	63.80	67.51	73.47	75.94	**77.46**
100×20	45.35	59.49	60.15	62.75	66.49	66.64	**67.01**
50×20	44.39	**58.30**	57.72	57.68	57.37	57.01	56.42
20×20	41.71	**50.28**	49.19	45.90	42.03	41.80	41.65
Total average speedup	44.52	57.23	57.72	58.46	59.84	60.35	60.64

Table 3.9: Speedups for different instances and pool sizes using thread divergence management.

In order to better investigate the impact of the thread divergence reduction, we draw in Figure 3.4 the number of divergent branches within a warp measured using the Nvidia Compute Visual Profiler [NVIDIA Corporation 2008] for the instances of 20 jobs over 20 machines. Counter values obtained from the Compute Visual Profiler are not the same as numbers obtained by inspecting kernel code. They are best used to identify relative performance differences between un-optimized and optimized code. These performance counter values represent events within a thread warp; they do not correspond to individual thread activity. Indeed, the divergent branch counter, we plot, is incremented by one at each point of divergence in a warp: if at least one thread in a warp diverges via a data-dependent conditional branch, the counter is incremented.

Figure 3.5 shows the time elapsed for executing the instructions contained in the divergent branches also for the instances of 20 jobs over 20 machines. For measuring the latter execution time we used the time function *clock*() which once executed in the device function returns the value of a per-multiprocessor counter that is incremented every clock cycle. Sampling this counter at the beginning and at the end of all conditional instructions, taking the difference of the two samples and recording the result provides a measure of

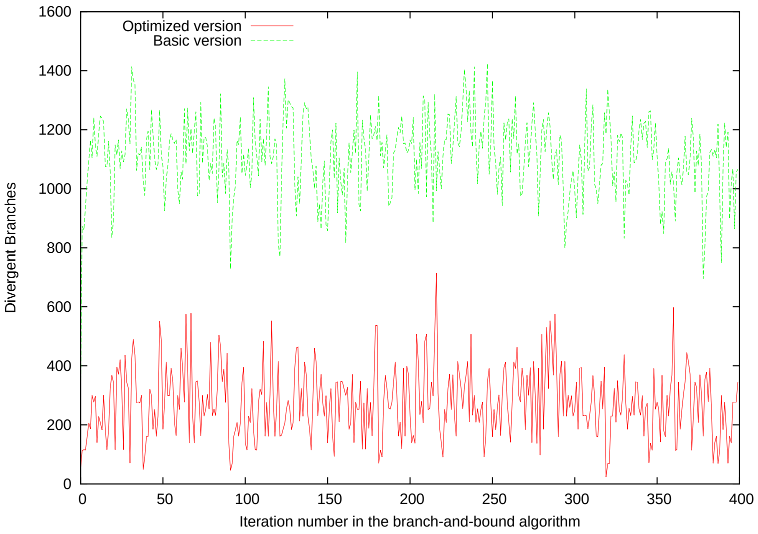


Figure 3.4: Number of divergent branches with and without thread divergence reduction.

the number of clock cycles taken by the device to completely execute these divergent instructions.

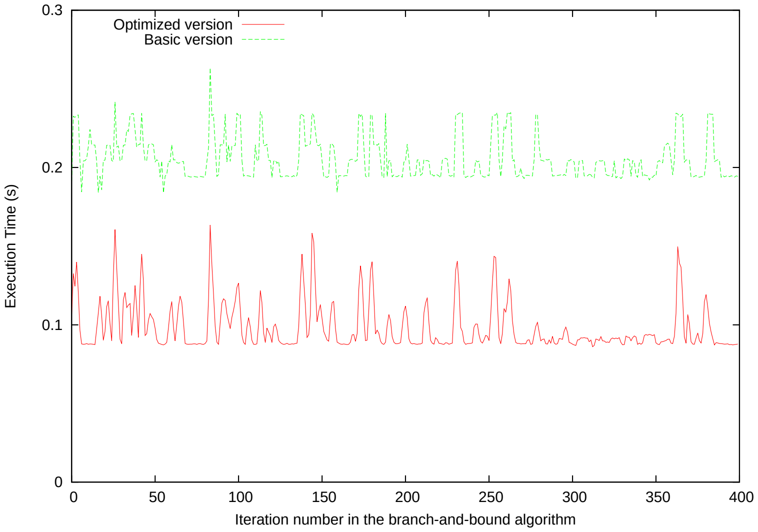


Figure 3.5: Elapsed time by the branches with and without thread divergence reduction.

The reported results in both figures show that the number of divergent branches measured using the code optimization we proposed is on average three times less than the number measured without code optimizations. However, the difference between the measured elapsed time for executing conditional instructions with and without code optimization is very tiny (on average around 0.12s). As claimed above, this little difference in execution time is due to the factorized part of the branches in the FSP lower bound which is very small and which explains the weakness of the obtained improvement.

Comparison of the branch refactoring technique with other software-based methods

As an additional enhancement of the proposed techniques, the branch refactoring method has been applied to the Monte Carlo simulation for Multi-Layered media (MCML) problem. MCML is a real-world medical application that models the scattering and absorption of photons in the tissue. This application is highly parallelizable, where a large number of photons are propagated independently, but according to identical rules and different random number sequences. The parallel nature of this special type of Monte Carlo simulation renders it highly suitable for execution on a GPU. This problem has been chosen in order to compare the proposed contribution with the work proposed in [Han 2011].

In [Han 2011], the authors also intervene at code level and introduce some software-based optimizations for reducing branch divergence in GPU programs: iteration delaying and branch distribution. Iteration delaying improves the utilization of execution units in the presence of a divergent branch within a loop. Branch distribution aims to reduce the divergent portion of a branch by factoring out structurally similar code from the branch paths. In our work, the focus is on transforming the *if-then-else* conditional instructions which is more akin to the branch distribution method rather than the iteration delay approach that targets the loop instructions.

Number of Photons	10000	50000	100000	500000	1000000	5000000
Percentage of improvement (%)	10.16 %	12.56 %	16.91 %	22.83 %	25.615 %	29.27 %

Table 3.10: Improvement obtained for the MCML problem using the branch refactoring method.

For experimentation, the GPU implementation proposed in [Alerstam 2010] has been used and tested on the same GPU device used in [Han 2011] namely a GTX 480 card. MCML has one kernel where each thread is assigned a number of photons to be simulated. Paths of the *if-then-else* instructions for which our transformations are applied, contain on average 80 fused multiply add instructions. Table 3.10 reports the improvement percentage obtained when applying the branch refactoring method compared to the original *if-then-else* instruction. The results show that the improvement grows accordingly to the number of simulated photons. The acceleration achieved by our refactoring method varies from 10% to 29% while the acceleration achieved by the branch distribution proposed in [Han 2011] varies from 5.6% to 16.1%.

3.6.5 Performances of the data access optimizations

The objective of the experimental study presented in this section is to find the best mapping of the six data structures of the lower bound LB kernel on the memories of the GPU device. As quoted in Section 3.5.1, such mapping depends on the sizes and access latencies/frequencies of these data structures and the GPU memories. The focus of our study is put on the global and shared memories.

Taking profit from the configurable storage space provided in the new Fermi-based devices such as the Tesla C2050 we used in our experiments, the 64 KB of storage was split between the shared memory and the L1 cache according to the experimented scenario.

- For the scenario were the data structures are put on the shared memory, the 64 KB of available storage are split on 48 KB for shared memory and 16 KB for L1 cache.
- For the scenario where the data sets are put on global memory, we used 16 KB for shared memory and 48 KB for L1 cache.

Problem instance Pool size	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	54.03	67.75	68.43	72.17	82.01	88.35	**90.51**
100×20	52.92	66.57	66.25	71.21	76.63	79.76	**83.01**
50×20	49.85	**65.68**	64.40	62.91	59.57	58.36	58.09
20×20	43.94	**58.10**	52.18	51.06	49.78	45.22	43.78
Average Speedup	50.18	64.52	62.56	64.33	66.99	67.92	68.84

Table 3.11: Speedups for different problem instances and pool sizes obtained with data access optimization. PTM, RM, QM and MM are placed in the GPU shared memory. JM and LM are copied to the global memory.

Table 3.11 reports the behavior of the speedup averaged on the different problem instances (sizes) as a function of the pool size when the matrices PTM, RM, QM and MM are put on the shared memory and JM and LM copied to the global memory. The table shows that the calculated acceleration grows on average with the growing of the size of the instance (the number of jobs) in the same way as in Table 3.7. For example, for the largest problem instance and a pool size of 262.144, the acceleration factor obtained is about ×90.51 faster than a single core-based execution while the best speedup for the instance with 20 jobs is ×58.10.

Problem instance Pool size	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	63.01	79.40	81.40	84.02	93.61	96.56	**97.83**
100×20	61.70	77.79	79.32	81.25	86.73	87.81	**88.69**
50×20	59.79	**75.32**	72.20	71.04	70.12	68.74	68.07
20×20	49.00	**60.25**	55.50	52.88	50.47	49.11	47.82
Average Speedup	58.37	73.19	72.11	72.29	75.23	75.56	75.61

Table 3.12: Speedups for different problem instances and pool sizes obtained with data access optimization. JM, RM, QM and MM are placed in the GPU shared memory. PTM and LM are copied to the global memory.

Compared to the speedups calculated in Table 3.7, which correspond to the scenario where all data structures are placed on the global memory, putting the matrix of processing times PTM on the shared memory allows improvements whatever the instance and the pool size are. For the instances of 100 jobs × 20 machines, the best speedup when all matrices are on global memory is calculated with the pool size 262.144 and is about ×67.01. For the same group of instance and the same pool size, the speedup in this scenario is ×83.01 which corresponds to an enhancement of 19%.

Table 3.12 reports the behavior of the speedup averaged on the different problem instances (sizes) as a function of the pool size when the matrix JM, RM, QM and MM are put on the shared memory and PTM and LM copied to the global memory. The results show that placing JM on shared memory accelerate the execution of the B&B by about 7% compared to the scenario where PTM is placed on shared memory and enables acceleration of ×97.83 compared to a serial B&B.

Table 3.13 reports the behavior of the speedup averaged on the different problem instances (sizes) as a function of the pool size when the matrix PTM and JM are placed together in shared memory and all others are placed in global memory. The results show that placing JM and PTM on shared memory accelerates the execution of the B&B by about 23% compared to the results in Table 3.7 where no data access optimization is considered.

Problem instance Pool size	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	66.13	87.34	88.86	95.23	98.83	99.89	**100.48**
100×20	65.85	86.33	87.60	89.18	91.41	92.02	**92.39**
50×20	64.91	**81.50**	78.02	74.16	73.83	73.25	72.71
20×20	53.64	**61.47**	59.55	53.39	52.40	50.03	49.37
Average Speedup	62.63	79.16	78.51	77.99	79.11	78.79	78.73

Table 3.13: Speedups for different problem instances and pool sizes obtained with data access optimization. PTM and JM are placed together in shared memory and all others are placed in global memory.

3.6.6 Overhead characterization of the GPU-accelerated parallel bounding operator

In order to further analyze the obtained speedups and to characterize the overhead induced by the host-device communications, we performed a temporal analysis of the steps involved in the GPU-based parallel evaluation of bounds:

- branching the subproblems on CPU.
- copying the pool from the host to the device.
- launching the kernel.
- copying the bounds from the device back to the host.
- assigning the bounds values to the corresponding subproblems.

To obtain an accurate assessment of the overhead, we used a high-level profiling as we computed the percentage of time consumed by each of the former steps for different FSP Taillard's problem instances. The obtained results are reported in Table 3.14.

The results show that the pre-treatment (branching the subproblems and copying them to GPU) and the post-treatment (copying the resulting bounds and assigning them to the corresponding subproblems) of the pool transfered to the device consume the major part of the time necessary for the evaluation of bounds on GPU. This observation is often encountered in GPU applications. Indeed, generally speaking, host-device synchronization is known to be a bottleneck that impacts the overall performance of GPU-accelerated applications. Therefore, as a further enhancement of our approach, the optimization of

(Nb. jobs × Nb. machines)	Branching on_CPU	Copy_Pool (H2D)	Kernel	Copy_bounds (D2H)	Assignment of_bounds
200×20	39.11%	47.92%	0.0031%	5.31%	7.65%
100×20	39.75%	43.96%	0.0033%	5.07%	11.21%
50×20	42.41%	33.89%	0.0045%	4.52%	19.18%
20×20	40.62%	24.59%	0.0062%	2.75%	32.04%

Table 3.14: Percentage of time consumed by each step of the parallel bounding approach.

the transfer of the subproblems and their associated lower bound values between CPU and GPU should be addressed.

3.7 Conclusion

In this chapter, we have presented the design of the GPU-accelerated parallel bounding operator we proposed to accelerate the execution of B&B algorithms. We particularly identified two main challenges that arise when revisiting the FSP lower bound kernel on GPUs. Indeed, because bounding is a problem-specific operator, it leads to a strongly input-dependent program behavior that impacts the accesses to the memory hierarchy of the GPU and conducts to thread divergence which harms the global throughput of the algorithm.

- **Designing a GPU-accelerated B&B based on the parallel evaluation of bounds**: In the proposed GPU-based approach (GB&B), the selection, branching and pruning of the subproblems are performed on CPU and the evaluation of their lower bounds (bounding operation) is executed on the GPU device. Pools of subproblems are off-loaded from CPU to GPU to be evaluated by blocks of threads. After evaluation, the lower bounds are returned to the CPU. The experimental results show that accelerations up to ×71.69 can be obtained especially for large problem instances and large pools of subproblems.

- **Mechanisms for reducing the number of divergent threads within a warp**: We proposed two mechanisms that aim to reduce the number of divergent threads during the execution of the FSP lower found function on GPU. The first thread-data reordering method target the divergence that is induced by the loop constructs. The second approach we called branch refactoring aims at decreasing the number of divergent branches that result from the *if-then-else* instructions. The experimental results show that the proposed techniques improve the speedup over a serial version of the B&B up to ×77.46.

- **Memory access optimizations**: We proposed a data access optimization approach that takes into account both the characteristics of the tackled instance and the memory constraints of the GPU device. The proposed data access pattern is based on a preliminary analysis of the FSP lower bound function. Such analysis allowed us to identify six data structures for which we have proposed a complexity analysis in terms of memory size and access frequency. Due to the limited size of the shared memory the matrices do not fit in all together. According to the complexity study, the recommendation is to put in the shared memory the Johnson's and the processing time matrices (JM and PTM) if they fit in together. Otherwise, the whole or a part of the Johnson's matrix has to be put in priority in the shared memory. The other data structures are mapped to the global memory. Such recommendation has been confirmed through extensive experiments. The optimizations obtained with the proposed approaches allowed us to achieve accelerations up to more than ×100.

Although, the proposed GPU-accelerated parallel bounding allows good speedups compared to a serial B&B, further improvements of the obtained results could be reached by optimizing the management of the pool of subproblems off-loaded to the device. First, the size of the pool should be tuned dynamically in respect to the problem being solved and to the used hardware configuration. Second, the transfer latency of the subproblems and their associated lower bound values between CPU and GPU should be minimized.

Chapter 4

GPU-based parallel tree exploration

Contents

Main publications related to this chapter

I. Chakroun and N. Melab.
Operator-level GPU-accelerated Branch and Bound algorithms. International Conference on Computational Science, ICCS 2013. Barcelona, Spain, June 5-7, 2013.

I. Chakroun and N. Melab.
An Adaptive Multi-GPU based Branch and Bound. A Case Study: the Flow-Shop

Scheduling Problem. 14th IEEE International Conference on High Performance Computing and Communications, HPCC'12. United Kingdom, Liverpool, June 24-27, 2012.

4.1 Introduction

In this chapter, we extend the design and implementation of the former presented GPU-accelerated B&B based on the parallel evaluation of bounds (GB&B). The proposed optimizations target the management of the pool of subproblems that is off-loaded to the GPU. Indeed, the experimental results obtained in Section 3.6.6 and the temporal analysis performed in Section 3.6.3, have shown that the pre-treatment and the post-treatment of the pool transfered to the device consume the major part of the time necessary for the evaluation of bounds on GPU and that the size of this pool has an important impact on the overall throughput of the algorithm.

Having in mind these observations, we first introduce an adaptive version of the (GB&B) where the selection operator is revisited so that the size of the selected pool is tuned dynamically according to the problem being solved and to the targeted hardware configuration. For dealing with this issue, we propose an empirical heuristic for parameters auto-tuning at runtime. As a second optimization, we tackle the CPU-GPU communication bottleneck that results from the transfer of the pool of subproblems and their associated lower bound values between the host and the device. Indeed, even if the bounding operator is highly time-consuming (97% to 99% of the total execution time), there is no guarantee that its GPU-based acceleration will significantly improve the performances of the B&B. This parallel bounding-based algorithm requires, in fact, some additional tasks which induce a notable overhead: the preparation of the pool of subproblems on which the bounding operator is applied, the transfer of the pool from CPU to GPU, and the transfer of the lower bounds from GPU to CPU. Therefore, the second contribution of this chapter is to extend the GB&B approach to minimize such overhead.

For achieving optimized CPU-GPU communications (see Section 2.4.3), our idea is to revisit on GPU the parallel tree exploration model which is reflected by the parallelization on GPU of the branching and pruning operators as well even if they consume less time than the bounding operator such as demonstrated in Section 3.3. Since they allow to reduce the cost of the data transfer between CPU and GPU, higher performances should be achieved. Indeed, knowing that the peak bandwidth between the internal device memory is much higher than the peak bandwidth between host memory and device memories, it is intelligible to minimize data transfer between the host and the device even if that means running kernels on the GPU that do not apparently demonstrate great speed-up compared with running them on the host CPU. Therefore, we investigate two different approaches based on the parallel tree exploration paradigm for performing full B&B operators on GPUs. The first multiple-nodes driven approach consists in exploring in parallel different sub-spaces of the tree. Selected parent nodes from the tree are assigned to different GPU threads which locally execute their own B&B. Each GPU thread locally performs the *branching*, *bounding* and *pruning* operators on multiple nodes and returns back to the host the list of promising nodes that would be explored in the following iterations. The second single-node driven approach consists in transforming the unpredictable and irregular workload associated to the B&B search tree (see Section 2.2.3) into data-parallel kernels optimized for the SIMD-based execution model of GPUs. All GPU threads compute in parallel the same amount of work on a single tree node. Using persistent data structures, the different operators are applied in parallel on a pool of unexplored nodes. At each iteration, parent nodes are selected and assigned to the GPU threads. First, the *branching* operator is applied in parallel: each thread generates a unique child and inserts it into a global pool. The underlying pool is used by the *bounding* operator which assigns a lower bound to each tree node. Finally, *pruning* is applied by each thread to decide whether to delete the assigned node or to turn it back to the host. Both approaches are experimented and their associated performances are compared to each other.

An efficient GPU-accelerated B&B algorithm is not only akin to a regular data-parallelism application but should also provide optimal values of algorithmic parameters which represent different variations and configurations of the algorithm. The size of the selected pool of subproblems is considered as an algorithmic parameter of our GPU-based B&B since it does not change the result of computations but has an impact on the overall performance. The empirical determination of the optimal values of the number of blocks and number of threads blocks presented in Section 3.6 was our first contribution towards an efficient measurement of the size of the pool. As a additional improvement of the size tuning, we propose an empirical heuristic for parameters auto-tuning at runtime. The heuristic dynamically adjusts the size of the pool to be off-loaded to the GPU according to the tackled problem and to the used hardware configuration.

The remainder of this chapter is structured as follows: Section 4.2 introduces the adaptive selection operator based on a proposed auto-tuning heuristic. Section 4.3 presents the multiple-nodes driven GPU-accelerated approach we investigate for accelerating the traversal of the tree search of the B&B. In Section 4.4, the single-node driven GPU-accelerated B&B is described and details about the parallelization of each operator are provided. Finally, in Section 4.5 details about the performed experimental study are given and the obtained results are discussed.

4.2 An adaptive selection operator based on a dynamic parameter tuning heuristic

One of the challenging concerns we considered with the aim to make efficient the GPU-based B&B is supplying the device with a large pool of subproblems. Indeed, experiments presented in Section 3.6 show that the proposed parallel bounding model is efficient when large pools (thousands of subproblems) are considered whatever the size of the FSP instances being tackled is. As a solution for the problem, we come up with a new selection strategy. Rather than selecting a single pending node as in traditional B&B algorithms, our approach assumes that a pool of nodes is selected from the *pending nodes* list (see Figure 4.1).

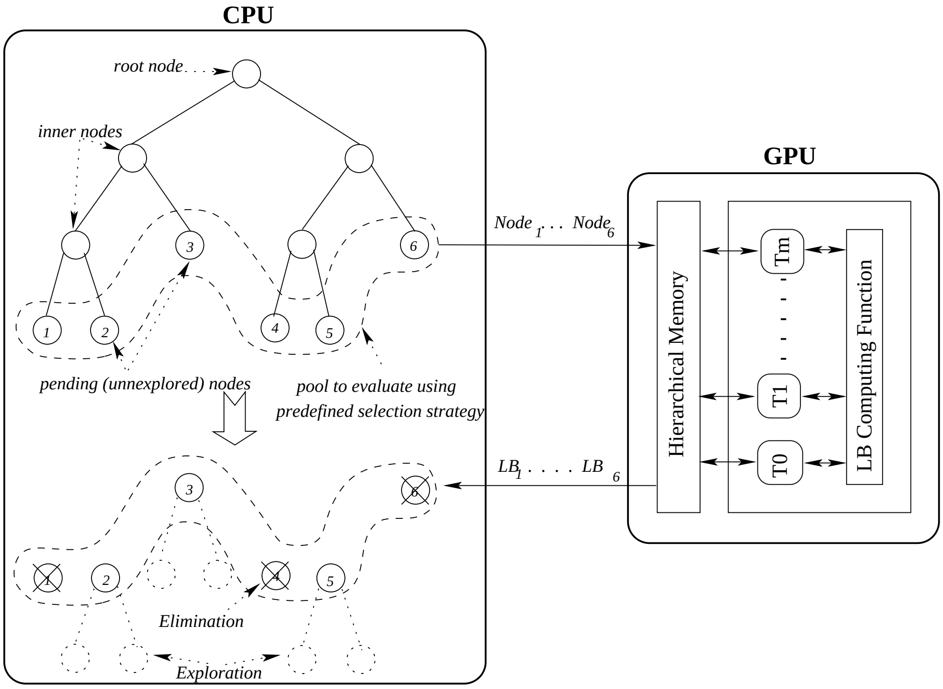


Figure 4.1: The overall architecture of the GPU-accelerated B&B algorithm based on the parallel evaluation of bounds. The approach introduces two main adaptations compared to a traditional B&B : selection of thousand of nodes and evaluation in parallel.

At each iteration of the algorithm, a pool of unexplored nodes is selected from the search tree according to their depth. Deepest pending nodes are the first selected for being branched. As explained before, that pool of subproblems, corresponding to the generated tree nodes and resulting from the branching operation, is off-loaded from CPU to GPU to be evaluated by blocks of threads.

As identified in Section 3.6, the size of the pool to be off-loaded to the GPU has an important impact on the performance of the algorithm. This parameter depends strongly on the problem instance being solved. It is thus hard to be fixed *a priori* and so has to be tuned dynamically depending on the problem. For dealing with this issue, we propose an empirical heuristic, we called it *Adaptive Selection Heuristic (ASH)*, for parameters auto-tuning at runtime. Algorithm 3 gives the general template for this heuristic. The main idea of this approach is to send the pending subproblems using different-sized pools to the GPU device during the first iterations of the B&B algorithm. For each iteration, the efficiency of the used pool is computed and the size of the pool to off-load to the GPU is doubled. After a fixed number of trials, the better efficiency overall selected configurations is used for the remaining iterations of the algorithm.

Since tunning the size of the pool to submit to the GPU is equivalent to adjusting the number of threads to be triggered, *ASH* first identifies the characteristics of the used hardware. It determines the maximum configuration that can be used, namely the maximum number of threads and blocks that can be run in parallel over the GPU card. Indeed, in some cases, when a thread block allocates more registers than are available on a multiprocessor, the execution of the kernel fails. During all the tuning process, the number of threads per block is set using the occupancy calculator tool provided by NVIDIA which allows the programmer to easily calculate the best thread block size based on register and shared memory usage of a kernel. Regarding the number of blocks per grid, our primary concern when choosing this parameter was keeping the entire GPU busy. Indeed, the number of blocks in a grid should be larger than the number of multiprocessors so that each of them has at least one block to execute. Thus, the number of blocks is first initialized with the number of multiprocessors detected on the device. This number is doubled repeatedly after a certain number of iterations (fixed experimentally) until the number of threads per block × the number of blocks doesn't exceed the maximum number of active threads allowed on the device.

So far, our empirical search for the best efficiency is coarse-grained. Indeed, doubling the size in every step, and stopping when the efficiency is no longer improved, or when the limits of the GPU have been reached might find an imprecise upper bound of the performance. For this reason and in order to make the tuning more thorough, we considered to also perform a binary search around the best pool size found so far. When the maximum number of active threads is reached, the iterative doubling process terminates and returns the best found configuration parameters. Thereafter, the heuristic computes a downwards and an upwards search around the best pool size found so far. The best efficiency overall selected configurations is used for the remaining iterations of the algorithm.

Algorithm 3 Template of the Adaptive Selection Heuristic (ASH).

```
Data: nb_iterations;
Result: best_number_of_threads
max_nb_threads = Detect_GPU_Characteristics();
nb_threads = Use_Cuda_Occupancy_Calculator();
nb_blocks := Get_Number_Of_Multiprocessors();
while not_empty_tree() do
    while pool_size ≤ nb_threads × nb_blocks do
        take_sub_problem();
    end
    Iteration pre-treatment on host side;
    Kernel evaluation on GPU;
    Iteration post-treatment on host side;
    if ( iteration % nb_iterations = 0 ) and ( (nb_threads × nb_blocks) ≤ max_nb_threads) then
        if Is_best_pool_improved() then
            best_number_of_threads = nb_threads × nb_blocks ;
        end
        nb_blocks := nb_blocks * 2 ;
    end
    else
        Compute_Binary_Search_Around_Best_Pool() ;
    end
    iteration := iteration + 1 ;
end
```

As a second optimization towards an efficient management of the pool of subproblems off-loaded to the GPU, we tackled the CPU-GPU communication bottleneck that results from its transfer between the host and the device. The first approach we investigate is presented in the following section.

4.3 The multiple-nodes driven GPU-accelerated approach

The first multiple-nodes driven approach we proposed for designing efficient B&B on GPUs, where the main operators are parallelized, consists in dividing the global search space into disjoint sub-spaces that are explored in parallel by the GPU threads. The approach is an extension of the model proposed in [Carneiro 2011]. As illustrated in Figure 4.2, in the considered GPU-based schema, a set of root nodes is selected from the *pending nodes* list according to their depth: deepest pending nodes are selected first.

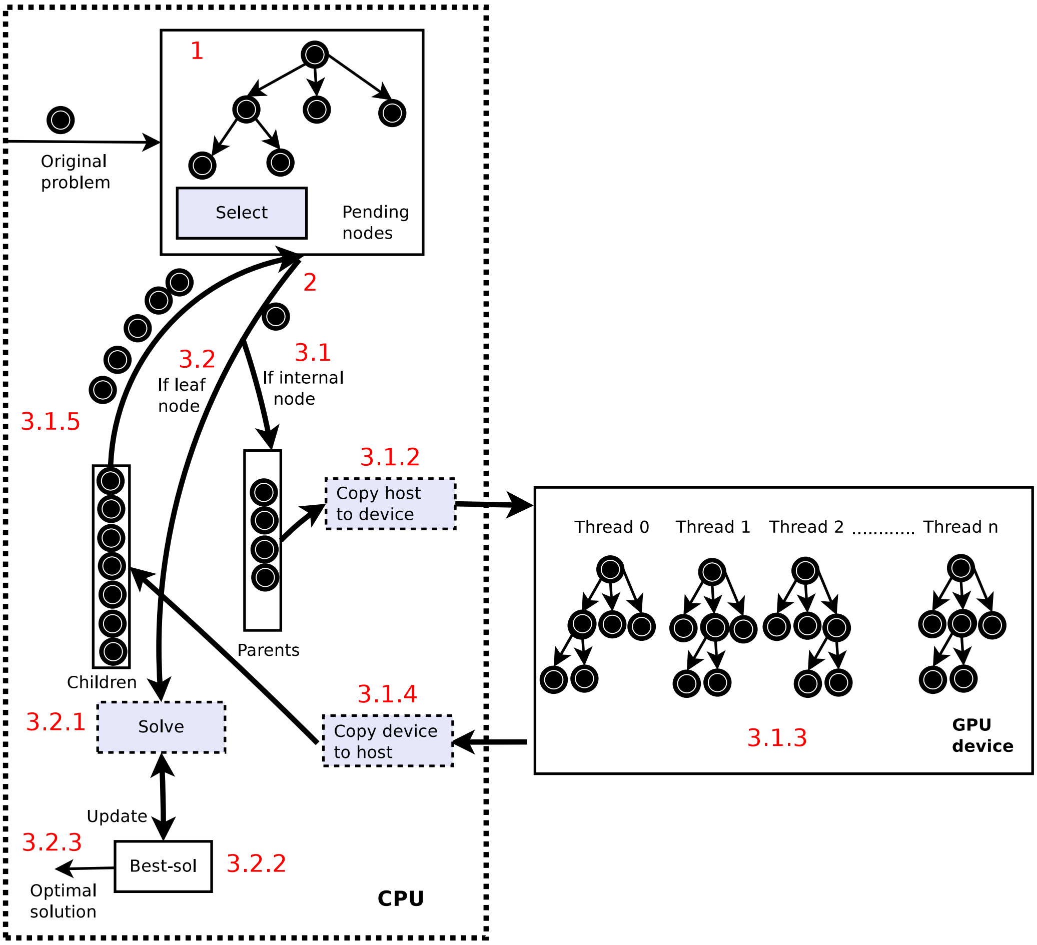


Figure 4.2: The overall architecture of the multiple-nodes driven GPU-accelerated B&B algorithm.

The selected pool of nodes is off-loaded to the GPU where to each thread is assigned a node. Each thread builds its own local search tree by applying the *branching*, *bounding* and *pruning* operators to the assigned node. The resulting nodes are moved back to the host where the promising nodes are inserted into the *pending nodes* set. The non promising nodes are kept on the device memory and deleted there. Figure 4.2 illustrates the overall architecture of the multiple-nodes driven GPU-accelerated B&B algorithm.

4.3.1 Branching Operator

Because all threads in a grid execute the same kernel function, they rely on unique coordinates to distinguish themselves from each other and to identify the appropriate portion of the data to process. As a remainder, these threads are organized into a two-level hierarchy using unique coordinates *blockIdx* (for block index) and *threadIdx* (for thread index) assigned to them by the CUDA runtime system. Therefore, defining an appropriate mapping mechanism is a critical step since it defines for each thread its assigned role, assigns specific input and output positions and determines the work unit to perform.

Mapping strategy

In our case, each node (subproblem) from the selected set is mapped to a thread to ensure that each sub-space of the solution space is evaluated concurrently and is disjoint from others. As detailed in *Algorithm* 4, thread i branches the node i, thread $i+1$ branches the nodes $i+1$, and so on. For each node i from the pending set, the number of subproblems to generate is calculated and an output pool where the generated nodes are written is allocated accordingly. Each thread writes the nodes it generates in the allocated range. In particular, thread i writes the generated nodes according to the number of children of the thread $i-1$. For instance, if thread 1 generates 3 children, and thread 2 generates 5 children then thread 2 starts writing in the output array from the position 3 and thread 3 starts writing from the position 8, etc. This pattern of writing in global memory locations leads to uncoalesced memory accesses which significantly penalize the throughput of the kernel execution.

4.3.2 Pruning Operator

The Johnson's algorithm used for computing the FSP lower bound assumes to assign jobs at the beginning and at the end of a partial schedule associated with a subproblem (see Section 3.2). Therefore, regardless of its level in the tree, each internal node has two pools of children: the first pool of children is obtained by scheduling jobs at the beginning of the partial schedule while the second results from scheduling jobs at the end of the partial schedule. In the schedule presented in Figure 4.3, jobs 1 and 2 are scheduled at the beginning, jobs 9 and 10 are scheduled at the end, and the other jobs are not yet scheduled.

The two pools of generated nodes are called *Begin* and *End*. As detailed in *Algorithm* 4, if the generated child corresponds to a schedule of a job at the beginning of the partial permutation, the child is inserted is the pool *Begin* otherwise it is put in the pool

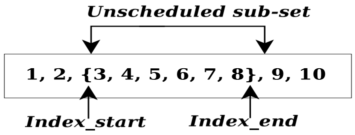


Figure 4.3: Representation of a partial schedule associated with a subproblem. The indexes between brackets correspond to unscheduled jobs.

End. Then, thread i computes the lower bound value for each child of the node i and writes the resulting nodes into the output pool in the position specified by its index. For pruning nodes, thread i compares the value of the bound of each node of the pool *Begin* and *End* to the best solution found so far *best-sol* and decides which pool to move back to the CPU and which pool to delete.

Algorithm 4 Kernel of the multiple-nodes driven GPU-accelerated B&B.

```
Data: Fathers = Parents nodes.
Result: Children = Nodes to be explored in the next iterations.
thread_idx = Get thread_id() ;
father = fathers[thread_idx];
output_position = estimated_children[thread_idx - 1];
for  j ∈ [ father.index_start + 1 , father.index_ end ] do
    child_begin = Generate_child_Begin(j);
    child_end = Generate_child_End(j);
    Evaluate_Lower_Bound(child_begin);
    Evaluate_Lower_Bound(child_end);
end
if  Choose_Begin_End() == Begin then
    Prune_End();
    Write_Begin_in_Children(output_position);
end
else
    Prune_Begin();
    Write_End_in_Children(output_position);
end
```

4.3.3 Synthesis

Using the above described approach, where each thread generates all the children of its root node, not only leads to uncoalesced memory accesses (see Section 2.4.2) but also conduct to an unbalanced workload between threads. As demonstrated in Section 2.2.3, during the exploration of the B&B tree, the number of new generated nodes and the number of promising nodes are variable and depend on the level of the tree being explored and on the best solution found so far *best-sol*. Therefore, due to such unstructured and unpredictable nature of the search tree, some threads might stay idle while other threads are overloaded. For instance, let us consider the example below which is extracted from the template of *Algorithm* 4 and used in the implementation of the branching operator. Let us suppose here that for 32 threads (*father.index_end* - *father.index_start*) is equal to 100 for the first thread and to 10 for the other 31 threads. In this case, all threads will finish the first 10 iterations together. Two passes will be used to execute each of the 90 following iterations, one pass for those that take the iteration and one for those that do not. All this extra-time (compared to an optimized execution) elapsed because of thread divergence (see Section 2.4.1), decrease the performances of the GPU-accelerated parallel tree exploration approach.

```
for $j \in [\,father.index\_start + 1\ ,\ father.index\_end\,]$ do
    child_begin = Generate_child_Begin(j);
    child_end = Generate_child_End(j);
    Evaluate_Lower_Bound(child_begin);
    Evaluate_Lower_Bound(child_end);
end
```

Compared to the approach proposed in [Carneiro 2011] which assumes that the pruning operator is performed on the CPU side, in the schema we suggest the pruning operator is also applied on the GPU device. Moreover, in [Carneiro 2011] the size of the pool to be off-loaded to the GPU is determined statically without taking into consideration neither the instance of the problem nor the underlying hardware configuration. In our approach, the size of the pool to be transfered to the device is calculated dynamically at runtime depending on the instance being solved and the used GPU configuration using the heuristic *ASH* described in Section 4.2.

In the following section, the second proposed template for the GPU-based parallel tree exploration is detailed. While the multiple-nodes driven approach is akin to an irregular computation-based model, the idea of the second proposed approach is to transform the unpredictable and irregular workload associated to the exploration of the B&B tree (see Section 2.2.3) into a sequence of regular data-parallel kernels applied to a set of nodes and optimized for the SIMD-based execution model of GPUs.

4.4 The single-node driven GPU-accelerated B&B

The second GPU-accelerated B&B we propose consists in launching consecutive data-parallel kernels that perform in parallel B&B operators. The main asset of this approach is that it transforms an irregular and unpredictable tree traversal into regular even tasks to be performed in parallel. All GPU threads compute in parallel the same amount of work on a single node.

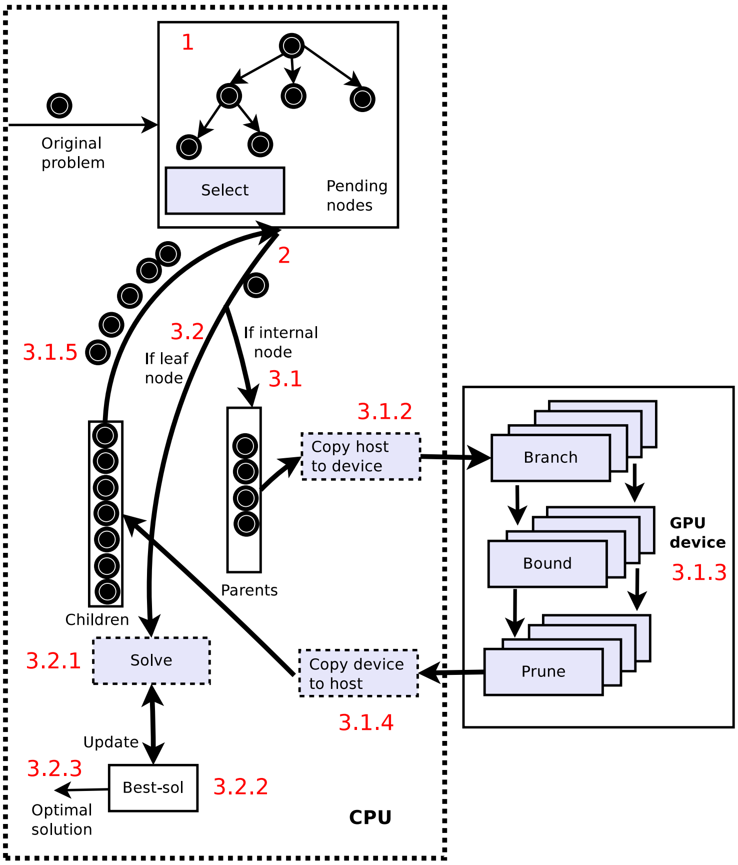


Figure 4.4: The overall architecture of the parallel single-node driven GPU-accelerated B&B algorithm.

As illustrated in Figure 4.4, the proposed algorithm proceeds as follows: at each iteration, a pool of root subproblems is selected on CPU host (according to the strategy described in Section 4.2) from the tree and off-loaded to the GPU where the branching operator is applied first: each thread generates a unique child and inserts it into a global pool. Here, it is important to highlight that we take care of using persistent data structures in order to minimize the data transfers between the CPU and the GPU.

Algorithm 5 Template of the single-node driven GPU-accelerated B&B based on the parallelization of the branching, bounding and pruning operators.

```
Create the initial problem;
Insert the initial problem into the tree;
Set the Upper_Bound to $+\infty$;
Set the Best_Solution to $\emptyset$;

GPU_Pool_Size = Run_Heuristic_For_Tuning_Pool_Size();
while not_empty_tree() do
    Sub_Problem = Take_sub_problem();
    if Is_leaf ( Sub_Problem ) then
        Upper_Bound = Cost_Of( Sub_Problem );
        Best_Solution = Sub_Problem;
    end
    else
        if Pool_Of_Fathers.size() < GPU_Pool_Size then
            Pool_Of_Fathers.push(Sub_Problem);
        end
        else
            Copy_Fathers_Pool_To_GPU();
            Copy_Number_Estimated_Children_Pool_To_GPU();

            Branching_Kernel<<<>>>;
            Bounding_Kernel<<<>>>;
            Pruning_Kernel<<<>>>;

            Copy_Promising_Children_Pool_From_GPU();
            Insert_Promising_Children();

        end
    end
end
```

Hence, the generated pool of children is kept in the device memory and used by the second kernel which implements the parallel evaluation of bounds and where each thread assigns a lower bound to a node. Then, the evaluated pool of children is again kept in the device memory (so not moved back to the CPU) where the pruning operator is run in parallel to decide which nodes should be moved back to the CPU and which nodes should be deleted. *Algorithm* 5 gives the general template of the single-node driven GPU-accelerated B&B.

4.4.1 Branching Operator

In order to ensure that all threads execute exactly the same amount of work, we redefined the serial branching operator so that each active thread generates only one of the children of its root node.

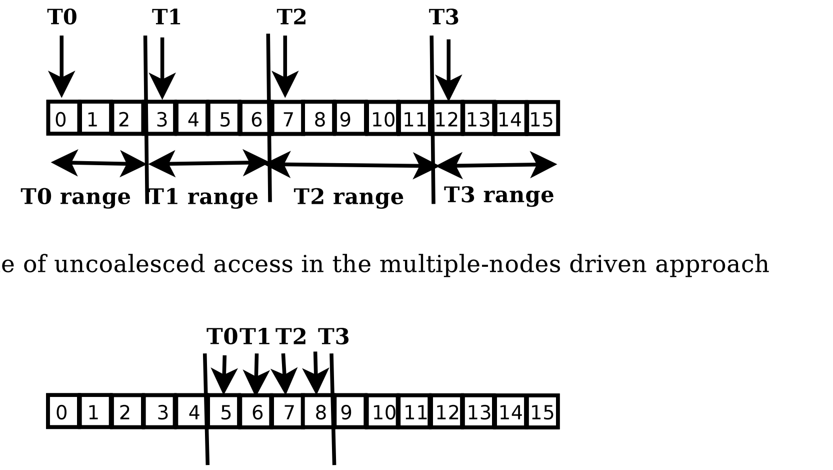
Mapping strategy

While in the multiple-nodes driven method, the thread i generates all the children of its root node, in this approach thread i only generates one child of its root node according to its unique identifier. Apart from the pool of root nodes, a pool containing the number of children of each root node is off-loaded to the device. As illustrated in *Algorithm* 6, using its unique identifier, each thread identifies its root node, which child it should generate and where to write the newly generated node in the global output structure stored in the device global memory. Compared to the multiple-nodes driven method, this way of applying in parallel the branching operator prevents from the thread divergence phenomenon explained in Section 2.4.1 since no data-dependent loop instructions occur and all threads execute exactly the same flow of instructions.

Algorithm 6 Kernel of the parallel branching operator on GPU.

```
Data: Fathers = Parents nodes.
Result: Children = Nodes to be explored in the next iterations.
thread_idx = Get_thread_id();
father = Get_father(thread_idx,Fathers);
generated_child = Generate_child(thread_idx,father);
Write_Generated_Children(thread_idx,generated_child,Children);
```

Apart from ensuring an even workload distribution among running threads, another major asset of the considered approach is that it prevents from the uncoalesced accesses to the global memory of the GPUs since its memory accesses constitute a contiguous range of addresses. Indeed, thread i writes the generated child node i in the position i (thread with $idx = 1$ generates one child and writes it in the position 1, thread with $idx = 2$ generates one child and writes it in the position 2, etc..). Figure 4.5 exhibits an example of the coalesced access to the output structure performed in the branching kernel of the second approach and the scattered access in the branching operator performed in the multiple-nodes driven approach.



Example of uncoalesced access in the multiple-nodes driven approach

Example of a contiguous and coalesced access in the single-node driven approach

Figure 4.5: Comparison of memory location accesses in the multiple-nodes driven and single-node driven GPU-based branching operator.

4.4.2 Pruning operator

In order to reduce the overhead induced by bringing the pool back and forth on GPU, the pruning operator is performed on top of GPU. This way, the time of transfering the resulting pool from the GPU to the CPU is reduced since the non promising subproblems are kept in the GPU memory and deleted there. To do so, the pool of bounded children is kept in the device memory where the elimination operator is applied by each thread in parallel to decide which nodes should be moved back to the CPU and which nodes should be deleted.

Since the Johnson's algorithm (used for computing the lower bound of the FSP) proceeds iteratively by assigning jobs at the beginning and at the end of a partial schedule (see Section 3.2), we defined two pools *Begin* and *End* where threads write the generated and evaluated children according to their indexes. As illustrated in *Algorithm* 7, if the generated child corresponds to a schedule of a job at the beginning of the partial permutation the node is written in the pool *Begin* else in the pool *End*. To each thread are assigned the two pools of children *Begin* and *End* corresponding to a same parent node. Using the value of *best-sol*, threads estimate which of the pools *Begin* and *End* are able to produce more promising nodes. The best pool is moved back to the CPU and inserted into the *pending nodes* list, the remaining pool is pruned in the GPU. This way of eliminating subproblems on the GPU level alleviates the overhead induced by bringing data back to CPU.

Algorithm 7 Kernel of the parallel pruning operator on GPU.

```
Data: Bounded_nodes = Nodes returned by the bounding kernel.
Result: Promising_nodes = Nodes to be explored next.

thread_idx = Get_thread_id() ;
parent_node = Get_father(thread_idx);
begin_pool = Get_begin_pool(thread_idx, parent_node, Bounded_nodes);
end_pool = Get_end_pool(thread_idx, parent_node, Bounded_nodes);

if choose_pool(begin_pool, end_pool) == begin_pool then
    write(begin_pool, Promising_nodes);
end
else
    write(end_pool, Promising_nodes);
end
```

4.4.3 Synthesis

A similar design of this approach is proposed in [Lalami 2012] where a GPU-accelerated B&B based on a parallel evaluation of bounds model coupled with a parallel tree exploration model where only the branching operator is parallelized. The algorithm is applied to the knapsack problem which is solved using a binary search tree: at each level of the tree, a parent node has only two children. This characteristic implies that the workload computed by each thread is the same and no irregular task balancing occurs. However for FSP, as demonstrated in Section 2.2.3, applying a B&B induces a highly irregular workload due on the one hand to the unpredictable number of branches pruned by the algorithm and on the other hand to the representation of FSP. At each level of the tree, the number of new generated nodes and the number of promising nodes are variable and depend on the level of the tree being explored and on the best solution found so far. As exploration strategy, the author uses a breadth-first search strategy which means that the pool of nodes that is off-loaded to the GPU are from the same tree level unless from successive levels. Compared to a depth-first selection strategy, using breadth-first one emphasizes the regular amount of the work flow that is assigned to each thread. In [Lalami 2012], the size of the pool off-loaded to GPU is statically determined and the pruning is performed on the CPU side.

4.5 Experiments

In the following, the performances of the *ASH* heuristic and of both multiple-nodes and single-node driven approaches are evaluated.

4.5.1 Performance evaluation of the *ASH* heuristic

The objective of the experimental study presented in this section is to demonstrate that the use of the adaptive selection operator is efficient and that it returns the best pool size that allows to take the most benefit from the use of the GPU.

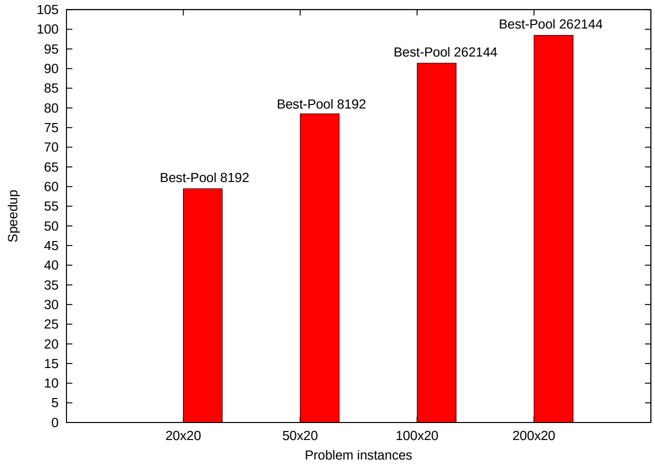


Figure 4.6: The speedups and corresponding used pools obtained using the auto-tuned algorithm.

Figure 4.6 depicts the speedups obtained for the different problem instances using the GB&B. For each problem instance we report the best pool returned by the proposed dynamic parameter tuning heuristic. To validate the obtained results, we run several experiments using different pre-fixed pool sizes. The corresponding results are reported in Table 4.1. The rows correspond to the problem instances defined by (Number of jobs × Number of machines) and the columns correspond to the size of the pool of subproblems evaluated in parallel.

The reported results confirm that the best speedups measured when varying the sizes of the pool are obtained with the same pool sizes returned by *ASH* (see Figure 4.6). For example, the best speedup for the 200 × 20 instances is obtained with a pool size of 262.144 which is the best pool size the proposed heuristic calculated for the same instances.

Problem instance	4096	8192	16384	32768	65536	131072	262144
(No. of jobs × No. of machines)	Average speedup for each group of 10 instances						
200×20	66.13	87.34	88.86	95.23	98.83	99.89	**100.48**
100×20	65.85	86.33	87.60	89.18	91.41	92.02	**92.39**
50×20	64.91	**81.50**	78.02	74.16	73.83	73.25	72.71
20×20	53.64	**61.47**	59.55	53.39	52.40	50.03	49.37

Table 4.1: Parallel speedup measured for different problem instances and pool sizes without using the *ASH* heuristic.

4.5.2 Performance evaluation of the proposed GPU-based approaches

The objective of the experimental study presented in this section is to evaluate and compare the performances of both the multiple-nodes driven GPU-accelerated B&B and the single-node driven GPU-based B&B. The adaptive selection operator is used for both approaches.

Table 4.2 reports the speedups obtained for the different problem instances using the two approaches presented in Section 4.4 and Section 4.3. The rows correspond to the used approach while the columns correspond to the experimented problem instance defined by (Number of jobs × Number of machines). The reported results show that executing the operators of the B&B in parallel allows to significantly speedup the execution of a B&B and that it is by far more efficient than exploring the B&B tree in parallel on GPU.

Number_of_jobs × Number_of_machines	20×20	50×20	100×20	200×20
Multiple-nodes driven approach	42.94	37.12	27.59	12.94
Single-node driven approach	79.42	128.41	144.13	160.41

Table 4.2: Speedups reported for the two approaches of the GPU-based B&B.

Compared to a single core CPU-based execution, the single-node driven approach allows significant accelerations reaching up to (×160.41) for the 200 × 20 problem instances. The same behavior observed for the GB&B approach (see Section 3.3) is perceived here since the obtained speedup with the single-node driven approach grows with the size of the problem instance. For a fixed number of machines, the speedup grows accordingly with the number of jobs. For instance, the speedup calculated with 200 jobs (×160.41) is higher than the one calculated with 100 jobs (×144.13), 50 jobs (×128.41) and 20 jobs (×79.42). This property is mainly due to the complexity of the computation of the lower bound which is $O(m^2.n.log(n))$. Indeed, for large problem instances the grain size of the kernel executed by each thread is much higher which increases the GPU throughput.

Compared to the multiple-nodes driven GPU-accelerated B&B approach, the single-node driven approach is by far much more efficient. For example, while the latter approach reaches speedup of ×160.41 for the instance with 200 jobs on 20 machines, a speedup of only ×12.94 is obtained with the multiple-nodes driven approach. Moreover, on the contrary of the single-node approach, in the multiple-nodes driven GPU B&B the speedups decrease when the problem instance becomes higher. Remember here that while in the single-node approach all threads compute only one node each whatever the permutation size is. In the multiple-nodes driven approach, each thread branches all the nodes of its root node. Therefore, the bigger the size of the permutation is, the bigger the amount of work performed by each thread is and the bigger the difference between the workloads is. Indeed, let us suppose that for the instance with 200 jobs, the thread 0 handles a node from the level 2 of the tree and the thread 100 handles a node from the level 170 of the tree. In this case, the thread 0 generates and evaluates about 6 times more nodes than the thread 100. The problem in this example is that the kernel execution would last until the thread 0 finishes its work while the other threads might have ended their works and stayed idle.

4.5.3 Impact of the parallelization of each operator of the single-node driven approach

In order to further detail the analysis of the performance of the single-node driven approach, we plot the impact of the parallelization of each of the operators of the B&B algorithm. Figure 4.7 shows that whatever the problem instance is, the three GPU models based on the parallel regular execution of the operators of B&B behaves better than the multiple-nodes driven approach.

For further details, Table 4.3 reports the speedups obtained for the three different single-node driven GPU-accelerated parallel B&Bs: (1) only the bounding operator is parallelized, (2) the branching and bounding operators are computed on GPU and (3) bounding, branching and pruning are executed on GPU. The results shows that computing all the branching, bounding and pruning operators on GPU gives the best accelerations. The results show also that carrying out the branching operator on the GPU device exhibits improvements ranging from about 14% for the smaller instances (with 20 jobs) to about 20% for the larger ones (with 100 jobs). Parallelizing the pruning operator on top of GPU allows an enhancement of about 29% for the instances with 200 jobs, 20% for the instances with 100 and 50 jobs and 10% for those with 20 jobs.

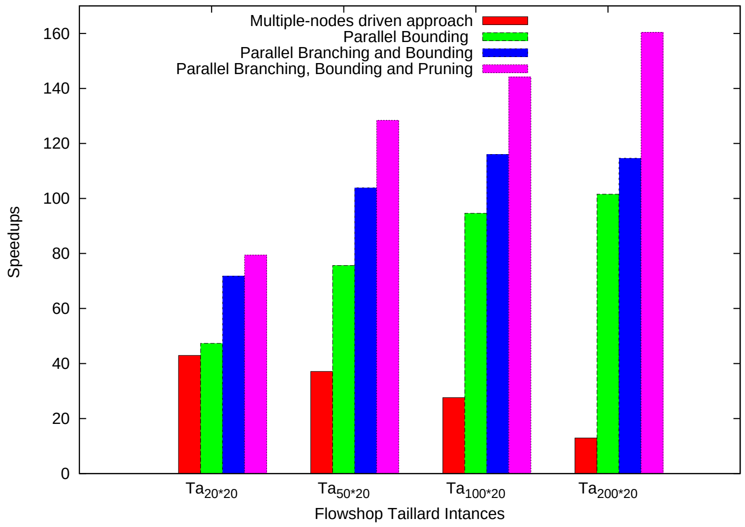


Figure 4.7: Comparison of the speedups obtained with different GPU accelerated versions of the B&B.

(Nb. jobs × Nb. machines)	Bounding Only on GPU	Branching and Bounding on GPU	Branching, Bounding and Pruning on GPU
200×20	100.48	114.63	160.41
100×20	92.39	116.00	144.13
50×20	81.50	103.90	128.41
20×20	61.47	71.73	79.42

Table 4.3: Speedup calculated with the parallelization of each operator.

To explain the enhancement resulting from each operator, we report in Table 4.4 the amount of data transfers exchanged between the CPU and GPU when each operator is parallelized. The results show that the average amount of the data transfer largely differs from an instance to another and becomes exorbitant for large instances with 100 and 200 jobs. The results demonstrate also that performing all branching, bounding and pruning operators on GPU allows one to reduce by about 50% the average amount of data exchanged between the host and the device compared to the data transfered when only branching and bounding are parallelized. This observation assets that in order to achieve best throughputs for GPU applications, one should strive to minimize the data transfers between the host and the device because those transfers have much lower bandwidth than internal device data transfers. Our recommendation here, is to avoid big data transfer by simply recomputing them whenever it is needed.

(Nb. jobs × Nb. machines)	Bounding Only	Branching and Bounding	Branching, Bounding and Pruning
200×20	181.29 MB	180.91 MB	98.42 MB
100×20	101.39 MB	100.45 MB	65.45 MB
50×20	1865.90 KB	1860.96 KB	840.10 KB
20×20	916.72 KB	926.26 KB	384.18 KB

Table 4.4: Comparison of the amount of data transfer with the different parallelization approaches.

Regarding the execution of the branching operator on GPU, one could notice that the amount of data transfered is almost the same as the amount exchanged when only the bounding operator is on GPU. Indeed, the pool of subproblems is no more transfered from the CPU to the GPU to be evaluated but it is generated on the device, evaluated and moved back from the GPU to the CPU which preserve almost the same amount of data transfer. However, the speedups calculated with the GPU-accelerated B&B where subproblems are generated on GPU are better than ones calculated when subproblems are decomposed on the host side. This is accomplished because more code is moved from the host to the device.

4.6 Conclusion

In this chapter, we have rethinked the design and implementation of the GPU-accelerated B&B based on the parallel tree exploration model. The management of the pool of subproblem exchanged between the CPU and the GPU is particularly addressed. We first investigated an efficient tuning of the size of the pool that is selected at each iteration. Second, we explored the optimization of the transfer of this pool and its associated lower bounds from the host to device. More exactly, the parallel B&B algorithm is extended with the parallelization on GPU of the branching and pruning operators which allows to reduce the cost of the data transfer between CPU and GPU.

- **An adaptive selection operator based on a dynamic parameter tuning heuristic.** Because of the size of the pool to be off-loaded to the GPU strongly depends on the problem instance being solved, we proposed an empirical heuristic for parameters auto-tuning at runtime that adjusts the size of the pool dynamically according to the problem being solved and to the used hardware configuration. The experiments show that using the adaptive selection operator is efficient and that it returns the best pool size that allows to take the most benefit from the GPU.

- **The multiple-nodes driven GPU-accelerated approach.** The first investigated approach consists in exploring in parallel different sub-spaces of the tree. Selected parent nodes from the tree are assigned to different GPU threads which locally execute their own B&B. Each GPU thread locally performs the *branching*, *bounding* and *pruning* operators and returns back to the host the list of promising nodes that would be explored in the following iterations.

- **The single-node driven GPU-accelerated B&B.** The second proposed template for GPU-accelerated B&B transforms the irregular workload into regular data-parallel kernels optimized for the SIMD-based execution model of GPUs. Compared to the multiple-nodes driven approach, thread divergence and uncoalesced memory accesses are considered in the optimization process. Compared to a serial execution, the single-node approach allows very significant acceleration reaching up to (×160.41) for the 200×20 problem instances. Compared to the multiple-nodes driven GPU-accelerated B&B approach, the single-node approach is by far much more efficient.

Because it is massively data-parallel and more fine-grained, the single-node driven approach is the most efficient approach for rethinking on GPU the parallel tree exploration model. However, further speedups could be reached if the multiple CPU cores available on nowadays ressources are judiscioulsy used. In the next chapter, this approach is extended for heterogeneous platforms where multiple CPU cores and multiple GPU devices are provided. In the rest of the document, the term *LL-GB&B* for Low-Latency GPU B&B refers to the GPU single-node driven approach since its major asset is to hide the latency induced by data transfers between CPU and GPU.

Chapter 5

Parallel Heterogeneous B&B combining GPU accelerators and multi-core processors

Contents

Main publications related to this chapter

I. Chakroun, N. Melab, M. Mezmaz and D. Tuyttens.

Combining multi-core and GPU computing for solving combinatorial optimization problems. Journal of Parallel and Distributed Computing (JPDC) - Elsevier (Under revision).

I. Chakroun and N. Melab.
Towards an heterogeneous and adaptive parallel Branch and Bound algorithm. Journal of Computer and System Sciences - Elsevier (Under revision).

5.1 Introduction

Heterogeneous computing systems combining GPU devices and multi-core CPUs, provide an opportunity to impressively increase the computational power for solving challenging problems. Therefore, designing heterogeneous parallel algorithms has been an active research area over last decade [Lastovetsky 2009, Brodtkorb 2010, Buchty 2012]. Nevertheless, none of the existing works on GPU-accelerated B&B algorithms [Lalami 2012, Carneiro 2011] has investigated the conjunction of the multi-core and the many-core processors for reducing the execution time of B&B algorithms. In this chapter, we introduce a prior work on designing an heterogeneous CPU-GPU accelerated multi-core B&B algorithm.

Although the GPU-accelerated B&B algorithms proposed in the previous chapters, allows one to significantly reduce the execution time needed to explore the B&B search tree, further speedups could be reached if the multiple CPU cores available on the underlying platforms are judiscioulsy used. Nevertheless, revisiting these algorithms for heterogeneous architectures requires a complete redesign to fit in together different hardware configurations. Indeed, the heterogeneity and incompatibility of CPU and GPU resources in terms of programming models makes parallelizing search algorithms very challenging and imposes one to judiscioulsy distribute data and computations workload between them. While some strategies give advantages to CPU cores, by making some host routines to be executed asynchronously with the GPU computations (for example, some iterations of the B&B algorithm are performed on the multiple CPU cores while other iterations are performed in parallel on GPU), others assume to use the CPU cores for only handling data input and output of GPU devices.

To be relevant to the growing number of heterogeneous computing systems, we have studied the combination of multiple CPU cores with one GPU and with several GPUs. For achieving this, we have proposed the following contributions:

- Rethink the B&B algorithm for multi-core machines endowed with multiple processing cores without GPUs.

- Propose a multi-core CPU-GPU accelerated B&B by investigating two patterns for combining multiple CPU cores and a single GPU.
- Redesign the CPU-GPU accelerated B&B for multi-GPU enabled configurations.

The remainder of this paper is organized as follows. Section 5.2 introduces a multi-core design and implementation of a B&B algorithm where no GPU device is used. In Section 5.3, the first approach for combining multi-core and GPU is presented. In Section 5.4, the second approach consisting in overlapping multi-core and GPU computing is detailed. In Section 5.5, the multi-GPU version of the algorithm is described. Section 5.6 details and discusses the performances for each of the proposed scenarii. Conclusions are drawn in Section 5.7.

5.2 Multi-core B&B *(MC-B&B)*

The scenario studied in this section concerns multi-core machines endowed with multiple processing cores that can be used concurrently (nCPU-0GPU). Several approaches to parallel programming for multi-core CPUs exist, ranging from low-level multi-tasking or multi-threading to high-level libraries that provide abstractions and features that attempt to simplify software development. For the proposed multi-core B&B algorithm, we have adopted a library-based approach using the standard POSIX thread library for implementation [Bradford 1996].

The proposed multi-core approach consists in partitioning the exploration of the B&B tree among running CPU threads (thread-based parallel tree exploration model). As illustrated in Figure 5.1, the algorithm starts by creating a number of threads that explore in parallel the B&B search tree. The number of running threads is a platform parameter that is fixed according to the used machine. The number of concurrent threads do not exceed the total number of computing cores. Threads cooperate by updating information stored in global shared variables. At any time during the exploration process, these two variables *pending-nodes* (the global pool of pending subproblems) *best-sol* (the best solution found so far) describe the current state of the B&B algorithm.

When using such shared memory between threads, the recommendation is to use locks as a synchronization mechanism since they enforce mutual exclusion and guarantee the coherence of the data. Thus, if two concurrent CPU threads try at the same time to pick or to insert a subproblem from or into the *pending-nodes* list, one of them is forced to wait until the lock is released by the other thread. At each iteration, a concurrent CPU thread

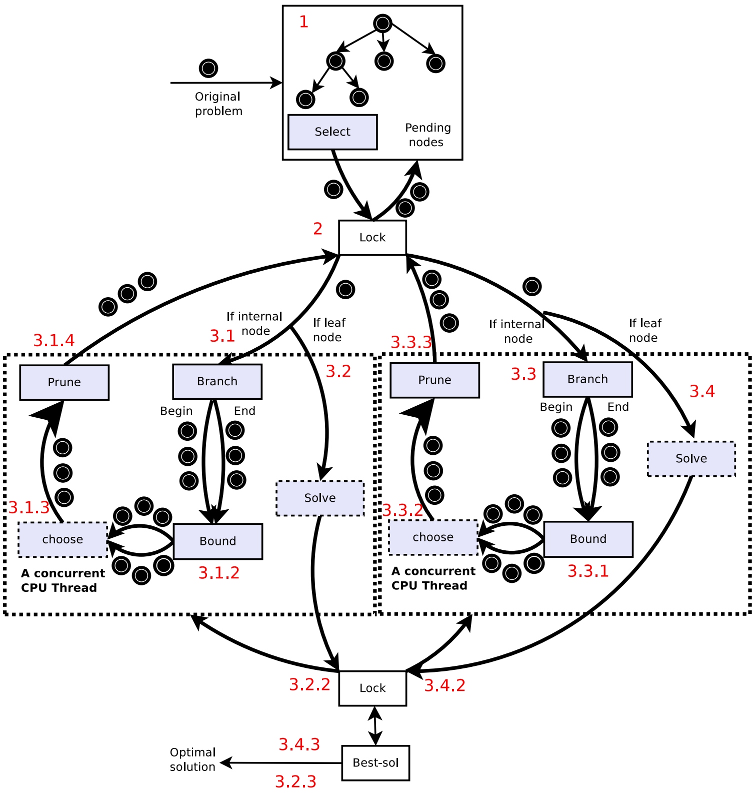


Figure 5.1: Illustration of the multi-core B&B algorithm.

tries to select a subproblem from the *pending-nodes* list. If no other thread is locking the pool, it picks the deepest subproblem having the smallest lower bound. Otherwise, this thread waits until the lock is free. If the selected subproblem is a leaf of the tree search, the cost of the solution of this subproblem is calculated and compared to the cost of the best solution found so far. If the cost of the best solution is improved, the thread puts a lock on the shared variable which stores the best solution found, and updates it with the new solution. Otherwise, the subproblem is deleted. If the selected subproblem is an internal node of the tree, it is decomposed and the lower bound function is applied to each of the generated children. The pruning operator eliminates each new generated subproblem having a bound greater than the cost of the best solution found so far. Finally, the non-eliminated subproblems are inserted into the *pending-nodes* list after locking the access to it. The concurrent CPU threads repeat the described process until this list is empty which corresponds to the termination of the algorithm.

As quoted above, the POSIX library is used for implementation. POSIX threads are native threads of processing that run within a single process/application and can share access to resources and memory at a fine-scale. The programmer explicitly creates and manages threads, with each thread inheriting its parent's access to resources. The programmer can synchronize threads and protect critical sections, such as shared memory locations in data structures and access to input/output resources, via mutual exclusion locks. These support three operations: lock, unlock, and try, a non-blocking version of lock where a thread either succeeds at acquiring the lock, or resumes execution without the lock. Condition variables suspend a thread until an event occurs that wakes up the thread. These variables in conjunction with mutex locks allow one to create higher-level synchronization events such as shared-memory barriers. In a threaded code, the programmer can then rely on coherency protocols to update shared memory locations.

5.3 ConcuRrent multi-core Low-Latency GPU-accelerated B&B *(RLL-GB&B)*

The scenario exposed in this section involves a large number of heterogeneous platforms where multi-core processors are combined with many-core processors. However, most of the heterogeneous applications that aim to bring in together multi-core processors and GPUs use multi-threading approaches to control independent GPU devices. Using multiple CPU cores for handling multiple GPUs is straightforward since one device could easily be assigned to one CPU core avoiding load balancing and concurrent access problems that might occur when only one GPU is shared by multiple CPU cores. In this latter case, more challenges related to the computation and data partitioning arise.

In order to combine using shared memory multi-core architecture and GPU, one of our investigated approaches consists in partitioning the exploration of the B&B tree among the CPU cores and the GPU device. For achieving this, a multi-threaded B&B is designed and implemented using the POSIX [Bradford 1996] standard.

As illustrated in Figure 5.2 and detailed in *Algorithm* 8, the algorithm starts by creating a fixed platform parameter number of threads that explore in parallel the B&B search tree. These threads are called concurrent CPU threads. In addition, the algorithm creates one special CPU thread called concurrent GPU thread to which the highest priority is assigned. Its role is to exploit the computing power of the GPU. The concurrent CPU threads and the GPU thread have a shared access to the *pending-nodes* list and to the *best-sol* variable. At any time during the exploration process, these two variables describe the current state of the B&B algorithm. The access to this couple of shared variables is handled using the same synchronization mechanism described in the previous section.

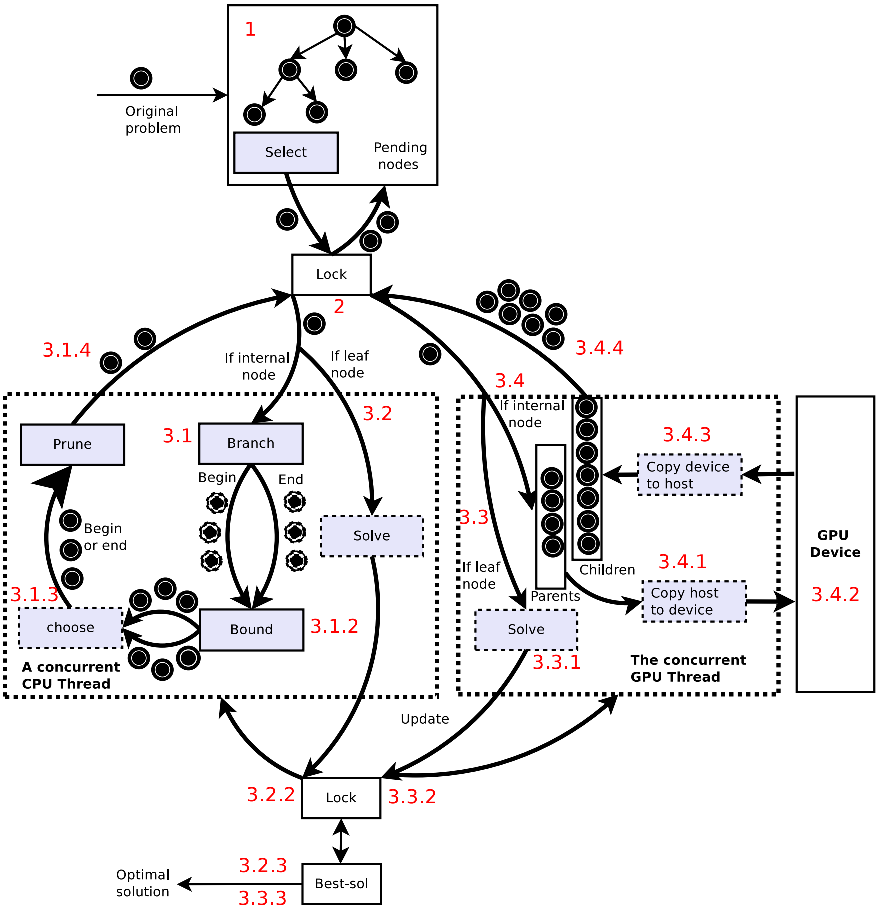


Figure 5.2: Illustration of the ConcuRrent multi-core Low-Latency GPU-accelerated B&B.

Algorithm 8 Template of the ConcuRrent multi-core Low-Latency GPU-B&B.

```
Create the initial problem;
Insert the initial problem into the tree;
Set the Cost_of_best_solution to $+\infty$;
Set the Best_Solution to $\emptyset$;

pthread_create(NULL,NULL,Concurrent_GPU_Branch_and_Bound_thread,NULL);
for $j \in [\,1\,,\ Number\_of\_threads\,]$ do
    pthread_create(NULL,NULL,Concurrent_CPU_Branch_and_Bound_thread,NULL);
end
```

5.3.1 Concurrent GPU thread

Algorithm 9 Template of a Concurrent_GPU_Branch_and_Bound_thread.

```
GPU_Pool_Size = Run_Heuristic_For_Tuning_Pool_Size();
while not_empty_tree() do
    Lock_shared_pool();
    Sub_Problem = Take_sub_problem();
    UnLock_shared_pool();

    if Is_leaf ( Sub_Problem ) then
        Lock_best_solution();
        Cost_of_best_solution = Cost_Of( Sub_Problem );
        Best_Solution = Sub_Problem;
        UnLock_best_solution();
    end
    else
        if Pool_Of_Fathers.size() < GPU_Pool_Size then
            Pool_Of_Fathers.push(Sub_Problem);
        end
        else
            Copy_Fathers_Pool_To_GPU();
            Copy_Number_Estimated_Children_Pool_To_GPU();

            Branching_Kernel<<>>;
            Bounding_Kernel<<>>;
            Pruning_Kernel<<>>;

            Copy_Promising_Children_Pool_From_GPU();

            Lock_shared_pool();
            Insert_Promising_Children();
            UnLock_shared_pool();

        end
    end
end
```

As shown in *Algorithm* 9, the concurrent GPU thread proceeds in several iterations.

At each iteration, it tries to pick a subproblem from the *pending-nodes* list. If no other thread is locking the pool, it selects the deepest subproblem having the smallest lower bound. Otherwise, this thread waits until the lock is free. If the selected subproblem corresponds to a leaf of the tree search, the cost of the solution of this subproblem is calculated and compared to the cost of the best solution found so far. If the best cost is improved, the concurrent GPU thread puts a lock on the *best-sol* shared variable and updates this variable with the new solution. Otherwise, the found solution is deleted.

If the selected subproblem is an internal node, the GPU thread inserts this subproblem in the *pending-nodes* list. The concurrent GPU thread continues selecting subproblems until the estimated number of children of the selected subproblems (see Section 4.4.1) exceeds the size threshold of the pool to off-load to the GPU returned by the *ASH* tuning heuristic. Once this size reached, the pool of the selected root nodes and the pool containing the number of children of each parent are copied to the device. After the branching, bounding and pruning kernels are finished, the pool of newly generated subproblems is inserted into the *pending-nodes* list. In order to insert these new nodes, the concurrent GPU thread waits until the other CPU threads free the lock of this shared list. The concurrent GPU thread repeats the described process until the shared pool is empty which corresponds to the termination of the algorithm.

5.3.2 Concurrent CPU threads

In the meanwhile, the other concurrent CPU threads execute a sequential version of the B&B as detailed in *Algorithm* 10. At each iteration, a concurrent CPU thread tries to select a subproblem from the *pending-nodes* list. If no other thread is locking the pool, the concurrent CPU thread picks only one subproblem. This thread uses the same selection strategy as that used by the concurrent GPU thread. In other words, the deepest subproblem having the smallest lower bound is the first to be selected. Once the subproblem is chosen, the thread frees the lock of the shared pool. As for the concurrent GPU thread, if the selected subproblem corresponds to a leaf of the tree search, the cost of its solution is calculated and compared to the cost of the best solution found so far. If the cost of the best solution is improved, the thread puts a lock on the shared variable which stores the best found solution. Then, this variable is updated with the new found solution. Otherwise, the subproblem is deleted.

However, if the selected subproblem is an internal node of the tree, it is decomposed into sub-sequent subproblems. The lower bound function is then applied to each generated node. The pruning operator eliminates each new generated subproblem having a bound greater than the cost of the best solution found so far. Finally, the non-eliminated subproblems are inserted into the *pending-nodes* list after locking the access to this variable.

Algorithm 10 Template of Concurrent_CPU_Branch_and_Bound_thread.

```
while not_empty_tree() do
    Lock_shared_pool();
    Sub_Problem = Take_sub_problem();
    UnLock_shared_pool();

    if Is_leaf ( Sub_Problem ) then
        Lock_best_solution();
        Cost_of_best_solution = Cost_Of( Sub_Problem );
        Best_Solution = Sub_Problem;
        UnLock_best_solution();

    end
    else
        Lower_Bound = compute_lower_bound(Sub_Problem);
        if Lower_Bound $\leq$ Cost_of_best_solution then
            Branch(Sub_Problem);
            Lock_shared_pool();
            Insert child sub problems into the tree;
            UnLock_shared_pool();

        end
        else
            Prune (Sub_Problem);
        end
    end
end
```

5.4 CooPerative multi-core Low Latency GPU-accelerated B&B *(PLL-GB&B)*

In order to avoid the synchronization issue and the overhead induced in the *RLL-GB&B* approach, the parallel algorithm proposed in this section is based on a more collaborative approach. Moreover, this approach allows one to further minimize the data transfer latency from CPU to GPU.

5.4.1 Overlapping data transfers and kernel calls

The principle of this approach is to make the GPU computations (i.e. the kernel calls) interleaved and overlapped with the data transfer operations. These operations are executed in parallel by the different threads. Since the communications between the host and the GPU device induce a considerable overhead on the performance of CPU-GPU accelerated applications, the aim of this proposed algorithm is to hide the latency of these operations by executing them asynchronously with the kernel calls.

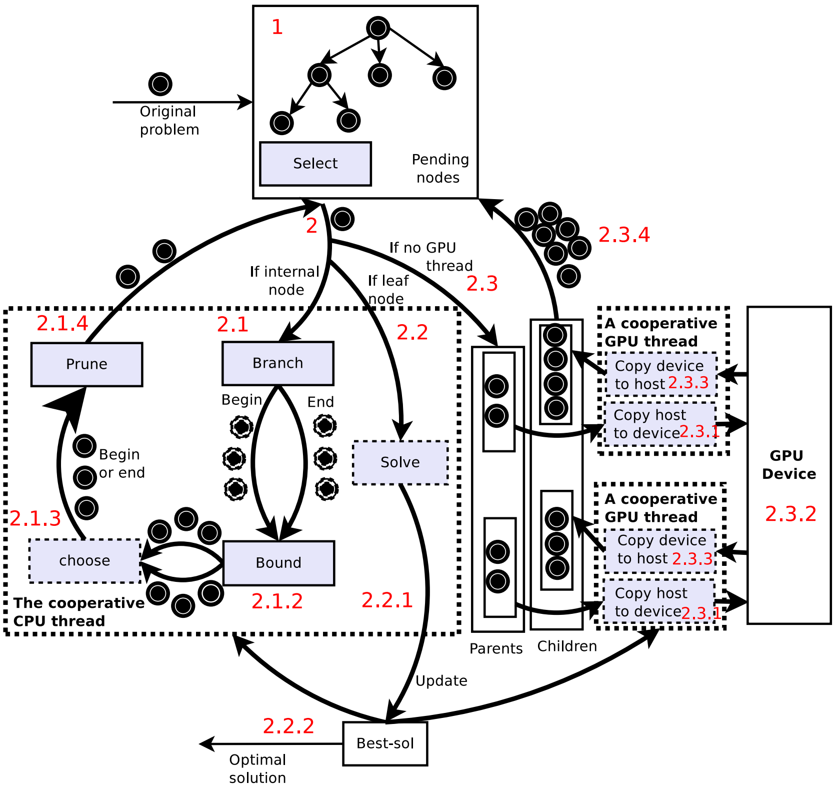


Figure 5.3: Illustration of the cooperative multi-core low latency GPU-accelerated B&B *PLL-GB&B*.

As illustrated in Figure 5.3, at each iteration of the algorithm, a thread, called collaborative CPU thread, picks a subproblem from the *pending-nodes* global list. This subproblem is inserted into the pool which contains the root subproblems to be transferred to the GPU. The collaborative CPU thread stops adding nodes to this pool when its size reaches the threshold fixed by the adaptive selection heuristic *ASH*. Then, the collaborative CPU thread creates a fixed platform-parameter number of threads, called collaborative GPU threads. The created new collaborative GPU threads execute in parallel a stream of ordered operations where the use of the GPU is interleaved and shared among these threads. The collaborative CPU thread waits for all the collaborative GPU threads to finish their executions. Then, the collaborative CPU thread inserts results of the collaborative GPU threads into the *pending-nodes* list.

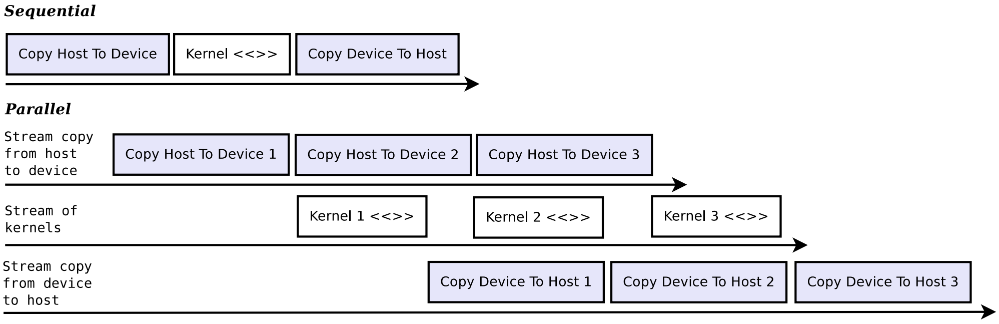


Figure 5.4: Sequential and concurrent operations performed on GPU devices with compute capability 2.0. Two copy engine and a kernel engine enables concurrent transfer operations and kernel execution.

A stream of ordered operations is associated with each collaborative GPU thread to perform its execution on the GPU. This is achieved using CUDA-enabled devices with compute capability 2.0, where a sequence of operations that are executed in issue-order on GPU is introduced. As illustrated in Figure 5.4, a compute (kernel) engine and two copy engines are provided: one for uploading from host to device and one for downloading from device to host. Each engine is equipped with a queue that stores pending data and kernels that will be processed by the engine shortly. Compared to classical execution on GPU where sequentially the data are pushed to the GPU, the kernel is launched and the results are retrieved, CUDA operations of different streams could overlap one with others. For example, as shown in Figure 5.4, the kernel launched by the host thread 2 is executed concurrently in parallel in stream 2 with the data copy (from host to device) performed in stream 1 by host thread 3.

5.4.2 Cooperative GPU threads

Each cooperative GPU thread handles a part of the pool of root nodes that is equally split. As illustrated in *Algorithm* 11, using a stream identifier, each thread (1) performs asynchronous transfers of its partition of the input pool to the device, (2) calls the kernels of branching, bounding and pruning operators that are processed only on its input partition, then (3) copies its portion of the output pool (the generated nodes of its assigned portion of subproblems) back to the CPU host side. In this approach, each collaborative GPU thread uses asynchronous data copy functions which are non-blocking. Using these functions ensures that the control is returned back to the collaborative CPU thread immediately and before the device has completed the requested task. Therefore, there is no synchronization between the cooperative GPU threads. Only the collaborative CPU thread must wait the end of all the collaborative GPU threads in order to insert the obtained new subproblems into the *pending-nodes* list.

Algorithm 11 Template of a Cooperative_GPU_Branch_and_Bound_thread.

```
Copy_Fathers_Pool_To_GPU(stream_id);
Copy_Number_Estimated_Children_Pool_To_GPU(stream_id);

Branching_Kernel<< stream_id >>;
Bounding_Kernel<< stream_id >>;
Pruning_Kernel<< stream_id >>;

Copy_Promising_Children_Pool_From_GPU(stream_id);
```

5.4.3 Cooperative CPU thread

As sketched in *Algorithm* 12, the main role of the collaborative CPU thread is to initialize the program and to create the collaborative GPU threads. In addition, this thread explores some pending subproblems from the global list while the collaborative GPU threads are performing parallel operations on the GPU. In a naive approach, the collaborative CPU thread has to wait until all the output data are brought back from the device to the host.

In the proposed approach, since the global *pending-nodes* list is not used in the meanwhile, the collaborative CPU thread picks a subproblem from the this list, and tests if the selected subproblem is a leaf. In the case this subproblem is a leaf, the collaborative CPU thread updates the *best-sol* variable if the solution of the leaf has a better cost than the existing one.

If the subproblem is an internal node of the tree, the collaborative CPU thread (1) decomposes this subproblem into a pool of subproblems, (2) bounds all the subproblems of the pool, (3) eliminates the nodes which are non-promising, and (4) inserts the promising subproblems into the global pool list. When all collaborative GPU threads finish their executions, the collaborative CPU thread continues processing the next iteration of the B&B. The algorithm stops when the global pool list becomes empty.

Algorithm 12 Template of Cooperative_CPU_Branch_and_Bound_thread.

```
GPU_Pool_Size = Run_Heuristic_For_Tuning_Pool_Size();
while not_empty_tree() do
    Sub_Problem = Take_sub_problem();
    if Is_leaf ( Sub_Problem ) then
        Cost_of_best_solution = Cost_Of( Sub_Problem );
        Best_Solution = Sub_Problem;
    end
    else
        if Pool_Of_Fathers.size() < GPU_Pool_Size then
            Pool_Of_Fathers.push(Sub_Problem);
        end
        else
            Split_ Pool_Of_Fathers();
            for j ∈ [ 1 , Number_of_threads ] do
                pthread create(...,Cooperative_GPU_Branch_and_Bound_thread,....);
            end
            while GPU_threads_running() = True do
                Sub_Problem = Take_sub_problem();
                if Is_leaf ( Sub_Problem ) then
                    Cost_of_best_solution = Cost_Of( Sub_Problem );
                    Best_Solution = Sub_Problem;
                end
                else
                    Lower_Bound = compute_lower_bound(Sub_Problem);
                    if Lower_Bound ≤ Cost_of_best_solution then
                        Branch(Sub_Problem);
                        Insert child sub problems into the tree;
                    end
                    else
                        Prune (Sub_Problem);
                    end
                end
            end
            Insert child subproblems returned by the GPU threads;
        end
    end
end
```

5.5 Low Latency Multi-GPU B&B algorithm *(LL-MultiGB&B)*

Nowadays, the trend in general-purpose computing on graphics processing units is to use multiple GPUs on a given system, like using multiple cores on CPU-based systems. The objective is to improve the performances by exploiting larger degrees of parallelism using multiple (parallel) GPUs. In the following section, details are given on the multi-GPU B&B algorithm. This algorithm is selected when the used hardware is composed by multiple CPU cores and at least two GPU devices (nCPU-nGPU).

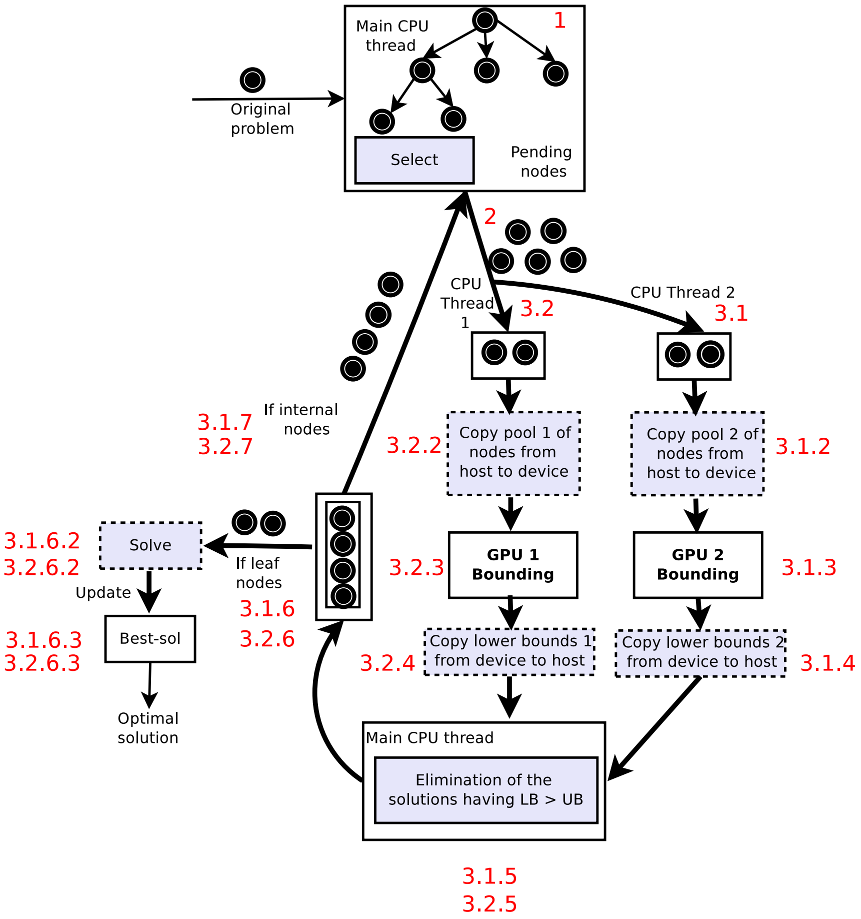


Figure 5.5: Illustration of the multi-GPU B&B algorithm where only the bounding kernel is on GPU.

The first step toward a multi-GPU design is to determine how many GPUs will be used and how each GPU will be exploited. In [Chakroun 2012], we have proposed a multi-GPU design of a B&B algorithm, where only the bounding operator is parallelized on GPU.

The aim is to use multiple GPUs to speedup the kernel execution rather than using each GPU differently. Consequently, the only concern in that case was to define a workload distribution between the used GPUs in order to make all the available devices compute the same work in parallel without need of synchronization. Since the approach ensures that the decomposed subproblems are different and independent from each other and since the used lower bound function is problem-dependent, we opted for simply splitting the pool of subproblems among the selected GPUs. Each pool is then evaluated in parallel and independently from other pools. After each GPU has finished computing the bounding kernel function, the outputs from each device have to be merged to get final results. The process is illustrated in Figure 5.5.

A main CPU thread selects a pool of from the *pending-nodes* list according to their depth. That pool of subproblems is equally split into as many pools as there are devices. In order to ensure full concurrency between the bounding computations, as many CPU threads as GPUs to be used are created. To each CPU thread is assigned an individual GPU using the NVIDIA CUDA “cudaSetDevice()” method [NVIDIA Corporation 2011a], which gives the possibility to select which device to execute the kernel on. Each created CPU thread copies its pool of subproblems from the CPU to its affiliated GPU, executes the kernel, and copies the resulting bounds back to the CPU. The main CPU thread waits for all other CPU threads to complete and merges the results into one.

In the newer optimized version of the algorithm, three kernels are executed separately on GPU. Therefore, the idea of the new multi-GPU based B&B is to split the kernels across the available GPUs and to profit from the time during which the kernels run in parallel to proceed with the next iterations of the algorithm. Concurrent CPU threads are used to decompose tasks among the multiple GPUs and deal with the challenges of data communication and synchronization.

As illustrated in Figure 5.6 and detailed in *Algorithm* 13, the main CPU thread selects a pool of unexplored nodes from the search tree. The branching kernel is executed first on one GPU device. Once the execution finished, the resulting pool of children is moved to the memory of a second GPU device using the peer-to-peer memory access feature explained in Figure 5.7. Peer-to-peer memory copies between two devices no longer need to be staged through the host and are therefore faster. As sketched in *Algorithm 14*, a second CPU thread launches then the bounding and the pruning kernels on the second device while the first CPU thread prepares the pool of the next nodes to be explored. When the second CPU thread finishes, i.e. when the bounding and pruning kernels end, the first CPU thread unlocks the shared pool where the resulting pool of children is pushed.

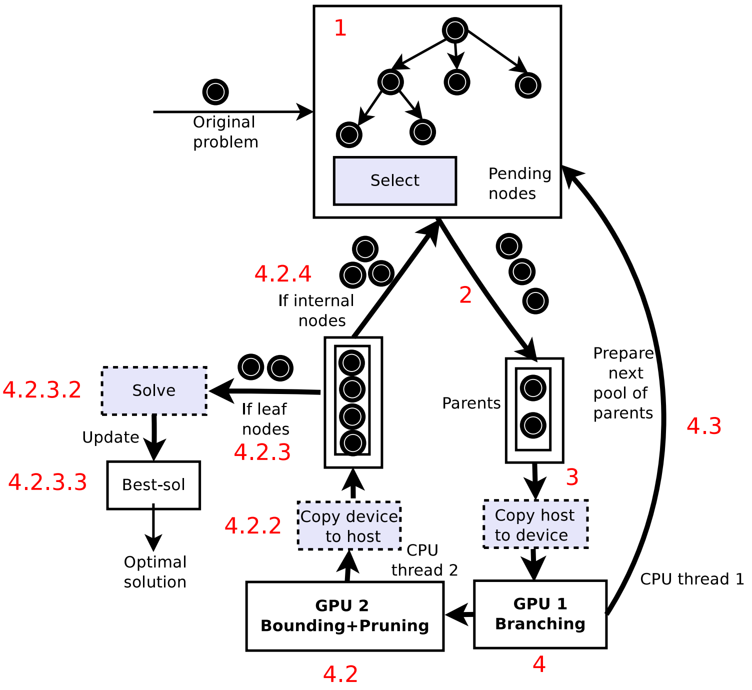
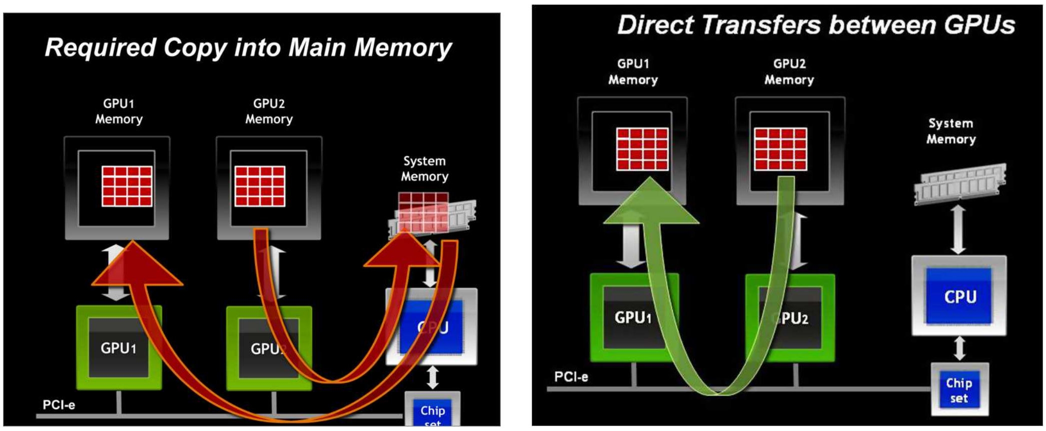


Figure 5.6: Illustration of the Low Latency Multi-GPU B&B algorithm *(LL-MultiGB&B)*.



(a) (b)

Figure 5.7: Data transfer without Peer to Peer direct transfer memory (via CPU memory) (a) with Peer to Peer direct transfer memory (b) (direct between GPUs) [NVIDIA Corporation 2011b].

If four GPU devices are provided, the idea is to combine two levels of parallelism. The first two CPU threads launch the branching kernels on two devices. To each thread CPU is assigned an individual GPU using the NVIDIA CUDA Runtime API "cudaSetDevice()" method. The selected pool of subproblems is equally split between both devices. There-

Algorithm 13 Template of Low Latency Multi-GPU B&B algorithm *(LL-MultiGB&B)*.

```
Create the initial problem;
Insert the initial problem into the tree;
Set the Cost_of_best_solution to $+\infty$;
Set the Best_Solution to $\emptyset$;
GPU_Pool_Size = Run_Heuristic_For_Tuning_Pool_Size();
while not_empty_tree() do
    Sub_Problem = Take_sub_problem();
    if Is_leaf ( Sub_Problem ) then
        Cost_of_best_solution = Cost_Of( Sub_Problem );
        Best_Solution = Sub_Problem;
    end
    else
        if Pool_Of_Fathers.size() < GPU_Pool_Size then
            Pool_Of_Fathers.push(Sub_Problem);
        end
        else
            Branching_Kernel<< device_0 >>;
            cudaMemcpyPeer (...,device_0,...,device_1,....);
            pthread create(...,Multi-GPU_Branch_and_Bound_thread,...);
            while Multi-GPU_thread_running() = True do
                Sub_Problem = Take_sub_problem();
                if Is_leaf ( Sub_Problem ) then
                    Cost_of_best_solution = Cost_Of( Sub_Problem );
                    Best_Solution = Sub_Problem;
                end
                else
                    if Pool_Of_Fathers.size() < GPU_Pool_Size then
                        Pool_Of_Fathers.push(Sub_Problem);
                    end
                end
            end
            Insert child subproblems returned by the Multi-GPU threads;
        end
    end
end
```

Algorithm 14 Template of a multi-GPU_Branch_and_Bound_thread.

```
cudaSetDevice(device_1);
Bounding_Kernel<<<>>>;
Pruning_Kernel<<<>>>;
```

after, the pool of subproblems generated by the branching kernel is copied respectively to the memory of remaining devices using the peer-to-peer memory copy where the bounding and the pruning kernels are executed. Each CPU thread copies the resulting pool of children produced by its affiliated GPU back to the CPU where they are merged into the *pending-nodes* list. In the meanwhile, the first two CPU threads select a pool from this list for the next iterations of the algorithm.

5.6 Experiments

The different approaches we have proposed in this chapter have been implemented using C-CUDA 4.0. The experiments have been carried out using an Intel Xeon E5520 bi-processor coupled with four GPU devices. The bi-processor is 64-bit, quad-core and has a clock speed of 2.27GHz. The GPU devices are an Nvidia Tesla C2050 with 448 CUDA cores (14 multiprocessors with 32 cores each), a clock speed of 1.15GHz, a 2.8GB global memory, a 49.15KB configurable shared memory, and a warp size of 32. The same experimental protocol as defined in 3.6.2 is used and the speedups are calculated relatively to the same serial B&B deployed on a single CPU core.

5.6.1 Performance of the multi-core B&B

The objective of the experimental study presented in this section is to evaluate the performance of the approach based on concurrent parallel exploration of the search tree using multi-core CPUs.

Table 5.1 reports the obtained speedups using the multi-core based B&B algorithm on different problem instances. The columns correspond to the number of concurrent CPU threads. The rows correspond to the problem instances defined by the number of jobs and the number of machines. Reported results show that the speedup obtained grows on average with the growing of the number of used CPU processing cores. For example, for the instances of 200 jobs on 20 machines, an acceleration factor of ×5.71 is calculated using 6 concurrent CPU threads while only a speedup of ×1.96 is calculated using 2 CPU threads running on 2 distinct CPU cores.

Number of concurrent CPU threads	2	3	4	5	6
200×20	1.96	2.86	3.81	4.76	5.71
100×20	1.96	2.87	3.82	4.77	5.73
50×20	1.95	2.86	3.81	4.76	5.70
20×20	1.91	2.86	3.80	4.75	5.20

Table 5.1: Obtained speedups using the *(MC-B&B)* approach where no GPU is used.

The results exhibit also that the slope is linear and that the acceleration factor is independent from the tackled instance. In fact, a speedup of on average ×5 is reported with 6 CPU threads and on average ×3.8 with 3 CPU threads whatever the number of jobs is.

Compared to the performances of the *LL-GB&B* approach reported in Section 4.5, the speedups obtained using only multi-core CPUs are by far less important. For example, for the instances of 100 jobs on 20 machines, using the GPU-based parallel B&B performs almost 26 times faster than the 6 cores-based B&B. However, the aim of the scenario considered here, is to demonstrate that with machines where multiple CPU cores are provided, the proposed approach allows accelerations compared to a serial B&B.

5.6.2 Performance of the *RLL-GB&B* approach

The objective of the experimental study presented in this section is to evaluate the performance of the approach presented in Section 5.3 and based on concurrent parallel tree exploration between the multi-core CPU and the GPU device.

Table 5.2 reports the speedup of the parallel CPU-GPU concurrent B&B averaged on the different problem instances. The columns correspond to the number of concurrent CPU threads. In this experiment, the number of concurrent GPU threads is always equal to 1. The rows correspond to the problem instances defined by the number of jobs and the number of machines.

The obtained results show that not only using CPU-GPU concurrent B&B approach decreases the speedup obtained with the *LL-GB&B* approach reported in Section 4.5, but also that the more the number of cores is, the worst the speedup is. For example, for the instances with 50 jobs over 20 machines, the acceleration factor is about ×123 with two CPU threads while it is about ×114 with five concurrent CPU threads and one GPU thread which corresponds to 7% of performances decrease.

Number of concurrent CPU threads	2	3	4	5
Number of concurrent GPU threads	1	1	1	1
200×20	155.02	151.16	146.45	143.42
100×20	142.67	141.18	139.49	137.73
50×20	123.17	120.09	116.15	114.74
20×20	79.29	78.64	77.91	76.96

Table 5.2: Obtained speedups using the *RLL-GB&B* approach with a single GPU.

To explain this behavior, we report in Table 5.3 the average normalized waiting times spent by the concurrent GPU thread for accessing global data structures. For each row, the waiting times are normalized and divided by the values obtained when only one concurrent GPU thread is used (no concurrent CPU threads). The reported results prove that when the GPU finishes its computation, the GPU thread is forced to wait for the lock to be free in order to access to the global pool (insert generated subproblems and take new subproblems to explore) even though the highest priority is assigned to it. This waiting time increases when the number of used concurrent CPU threads increases which explains the decrease in speedup when the number of concurrent CPU threads increases.

Concurrent CPU threads	0	1	2	3	4	5
Concurrent GPU threads	1	1	1	1	1	1
20×20	1	1.13	1.15	1.16	1.17	1.19
50×20	1	1.16	1.2	1.23	1.30	1.44
100×20	1	1.22	1.26	1.32	1.41	1.56
200×20	1	1.94	2.12	2.28	2.42	2.9

Table 5.3: Average normalized waiting times spent by the concurrent GPU thread when accessing global data structures.

Moreover, the results previously reported in Table 5.1 correspond to a CPU-GPU concurrent B&B approach where no concurrent GPU thread is used. As quoted before and unlike the results observed in Section 5.2, the speedup obtained when no concurrent GPU thread is used grows on average with the growing of the number of used computing CPU cores. This further demonstrates that the limited behavior of the CPU-GPU concurrent B&B algorithm is not due to the concurrent CPU threads but to the under-utilization of the GPU by the concurrent GPU thread.

5.6.3 Performance of the *PLL-GB&B* approach

In this section, the performance of the CPU-GPU cooperative parallel B&B algorithm introduced in Section 5.4 and based on the collaborative computation between the multi-core CPU and the GPU device is evaluated.

Number of cooperative CPU threads	1	1	1	1
Number of cooperative GPU threads	2	3	4	5
200×20	161.28	162.95	164.41	168.67
100×20	148.64	153.46	155.38	160.91
50×20	140.55	148.21	149.87	153.92
20×20	109.27	111.08	113.11	122.31

Table 5.4: Obtained speedups using the *PLL-GB&B* approach where the cooperative CPU thread does not perform the exploration of subproblems.

Table 5.4 reports the speedups obtained by the CPU-GPU cooperative B&B approach on the different problem instances. The columns correspond to the number of cooperative GPU threads. The number of cooperative CPU threads is always equal to 1. In these experiments, the used cooperative CPU thread does not explore nodes. The rows correspond to the problem instances defined by the number of jobs and the number of machines. The results show that the obtained speedups increase according to the instance size and to the number of used cooperative GPU threads. For a same instance, for example the instance defined by 20 jobs and 20 machines, the speedup obtained using two cooperative GPU threads is ×109.27 while it is ×122.31 using five cooperative GPU threads.

Number of cooperative CPU threads	1	1	1	1
Number of cooperative GPU threads	2	3	4	5
200×20	164.69	165.99	167.52	170.69
100×20	150.12	155.29	156.59	162.27
50×20	144.72	150.81	151.16	156.70
20×20	112.22	114.86	117.53	124.10

Table 5.5: Obtained speedups using the *PLL-GB&B* approach where the collaborative CPU threads explores nodes in parallel to the GPU execution.

Table 5.5 reports the impact of making the cooperative CPU thread exploring in parallel the search tree while the other cooperative GPU threads are busy sharing the use of the GPU. The results show that exploring in parallel some subproblems on the CPU side, while the GPU is computing and when no concurrent access to the shared queue occurs, allows accelerations up to 36% compared to the *LL-GB&B* approach. For example, for the instances with 20 jobs over 20 machines, the speedup obtained with the *LL-GB&B* is ×79.42 (see Table 4.2) while it is ×124.10 with the *PLL-GB&B* (see Table 5.5).

5.6.4 Performance of the *LL-MultiGB&B* approach

In this section, we experiment the use of the parallel B&B algorithm with multiple GPUs. The objective here is to evaluate the impact of the *LL-MultiGB&B* approach proposed in Section 5.5.

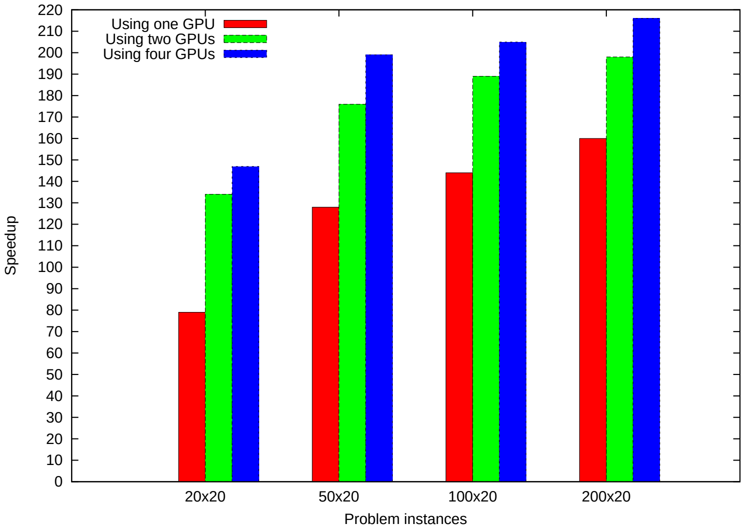


Figure 5.8: Comparing the speedup for different problem instances using a single / multiple GPUs.

Figure 5.8 compares the computed speedups obtained for the different problem instances using respectively 1, 2 and 4 GPU(s). The reported results show that the speedup grows accordingly to the number of used GPUs. For instance, an acceleration factor up to ×216.92 is obtained with 4 GPUs for the 200 × 20 problem instances while a speedup of ×198.55 is obtained for the same instances using 2 GPUs and ×160.41 with only one device. However, the improvement is not linear and the slop decrease as long as the number of the used GPUs raises. This is explained by the synchronization overhead induced by the use of CPU threads. Let us recall here that, unlike the scenario where 2 GPUs are used, the pool of selected nodes is split into as many pools as used devices and copied to the memory of each GPU. The CPU threads have also to copy the resulting pools of promising nodes (evaluated) produced by its affiliated GPU back to the CPU and merge them into the *pending-nodes* list.

5.7 Conclusion

In this chapter, we have investigated the design and implementation of an heterogeneous CPU-GPU accelerated multi-core B&B algorithm. Our first contribution was to propose a new template for a heterogeneous multi-CPU single-GPU accelerated B&B. As a second contribution, we rethink the CPU-GPU accelerated B&B for multi-GPU enabled configurations.

- **Multi-core based B&B algorithm** *(MC-B&B)* This first parallel B&B concerns multi-core machines and consists in partitioning the exploration of the B&B tree among CPU threads. Threads cooperate by updating the *pending nodes* and the *best-sol* shared variables. The results exhibit that the speedup grows with the number of used CPU cores, that the slope is linear and that the acceleration factor is independent from the tackled instance. Compared to *LL-GB&B*, the speedups obtained using only multi-core CPUs are by far less important. However, the aim of the scenario considered here, is to demonstrate that with machines where multiple CPU cores are provided, the proposed approach allows accelerations compared to a serial B&B.

- **ConcuRrent multi-core Low Latency GPU-accelerated B&B** *(RLL-GB&B)* In this approach where a GPU and multi-core CPUs are bringing in together, the computations performed by the GPU and the multi-core are concurrent. Each of the CPU threads and the GPU device explores in parallel the search space. The experiments show that using a concurrent exploration of the B&B tree between GPU and CPU threads is not efficient compared to a single core CPU-GPU execution since the GPU is forced to wait for shared memory spaces to be free which lead to its under-utilization.

- **CooPerative multi-core Low Latency GPU-accelerated B&B** *(PLL-GB&B)* In this approach, CPU threads and GPU cooperate together in order to avoid synchronization issues. Towards a latency hiding strategy, the algorithm assumes that the threads overlap the memory copies of the data that are off-loaded to the device with the kernel executions on the GPU. The second asset of this approach is to add further degree of concurrency by making the main CPU thread exploring some pending subproblems while other threads are busy with the GPU. The reported results show that the *PLL-GB&B* approach enables accelerations up to 36% compared to *LL-GB&B*.

- **Low Latency Multi-GPU B&B algorithm** *(LL-MultiGB&B)* The idea of the multi-GPU based B&B is to split the kernels across the available GPUs and to profit from the time where the kernels run in parallel to proceed with the next iterations of the algorithm. The branching kernel is executed first on one GPU device and the resulting pool of subproblems is moved to the memory of a second GPU device using the peer-to-peer memory copy. A second CPU thread launches then the bounding and the pruning kernels on the second device while the first CPU thread prepares the pool of the next nodes to be explored. Compared to a serial B&B, accelerations up to ×216.92 are reached for large problem instances with the multi-GPU based approach and the more GPU devices are used, the better the speedups are.

Chapter 6

Towards a grid-enabled GPU-accelerated Branch and Bound

Contents

To be relevant to the arrival of GPU accelerators and the advent of multi-core in clusters and computational grids, we propose in this chapter a large-scale distributed version of the heterogeneous multi-core GPU-accelerated Branch and Bound algorithm. The targeted execution environment is composed of a set of heterogeneous computing nodes provided through a computational grid. Each computing node is either a single multi-core processor or multi-core processors coupled with one or several GPU(s). For achieving this, we propose B&B meta-algorithm coupled with the B&B@GRID approach proposed in [Mezmaz 2007a]. Indeed, while B&B@GRID allows one to efficiently partition the B&B tree search among distant computing nodes, the meta-algorithm explores assigned subtrees using the parallel B&B algorithm that best fits the hardware configuration of the underlying execution nodes.

The remainder of this chapter is structured as follows: Section 6.1 presents the overall design of the heterogeneous B&B for computational grids. The comprehensive description includes details about the B&B meta-algorithm and the used B&B@GRID approach. In Section 6.2 details about the used experimental platform and the experimented problem instances are given and the obtained results are discussed.

6.1 Parallel heterogeneous B&B for computational grids : joining two levels of parallelism

In this section, the overall design of the proposed heterogeneous GPU-enabled B&B for computational grids is detailed.

6.1.1 Overall design of the distributed heterogeneous B&B (HB&B@GRID)

Our approach to further reduce the exploration time consumed by B&B algorithms for solving challenging COPs, is to extend our work to use a large number of computational-powered ressources. As claimed in Section 2.6, such significant computing power may be provided through a computational grid which is a collection of geographically-distributed heterogeneous computing resources. As illustrated in Figure 6.1, the proposed approach consists in hierarchically combining two levels of parallelism by (1) dividing the B&B tree among multiple distributed grid nodes and (2) exploring in parallel each sub-tree.

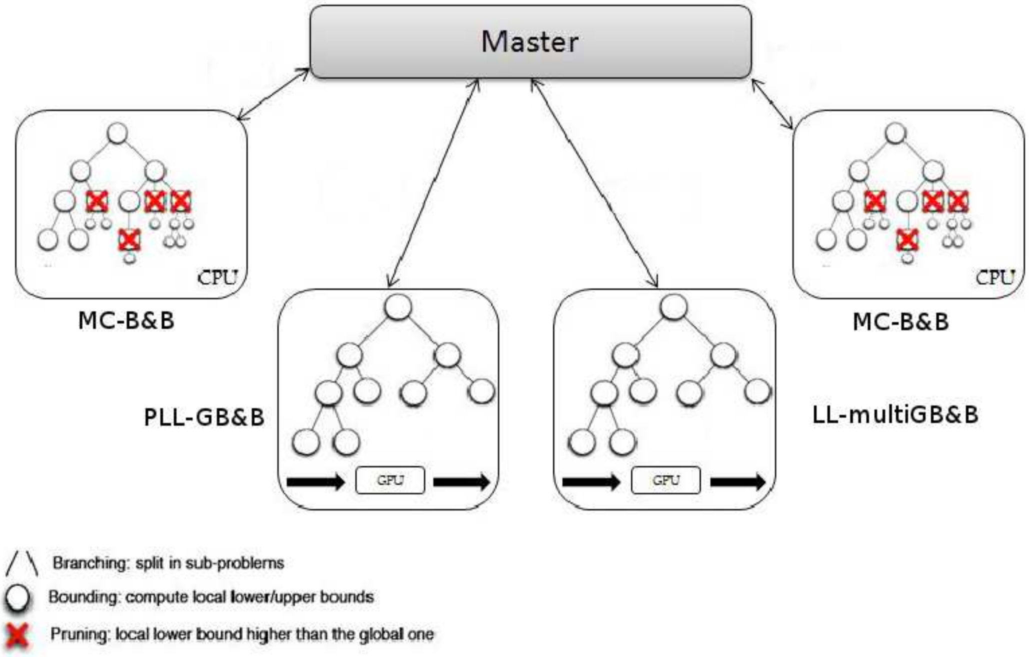

Master
CPU
MC-B&B
PLL-GB&B
GPU
GPU
LL-multiGB&B
CPU
MC-B&B
Branching: split in sub-problems
Bounding: compute local lower/upper bounds
Pruning: local lower bound higher than the global one

Figure 6.1: Overview of the distributed heterogeneous B&B (HB&B@GRID).

The first level of parallelism, carried out by the B&B@GRID master-worker approach, is based on the parallel tree exploration model presented in Section 2.3.1. It consists in launching a master process to control the distributed exploration and several worker processes to explore simultaneously different paths of the same tree. Each worker applies the B&B algorithm using a depth first exploration strategy. As soon as a new best solution for the problem being solved is found, it is communicated to other workers through the master.

The second level of parallelism is carried out using a B&B meta-algorithm that automatically selects the parallel B&B to be deployed according to the hardware configuration of the underlying grid node. As illustrated in Figure 6.2, four hardware configuration scenarii have been considered: single CPU core coupled with a single GPU (1CPU-1GPU), multi-core CPU without GPUs (nCPU-0GPU), multi-core CPU coupled with a single GPU (nCPU-1GPU), multi-core CPU coupled with multiple GPUs (nCPU-nGPU).

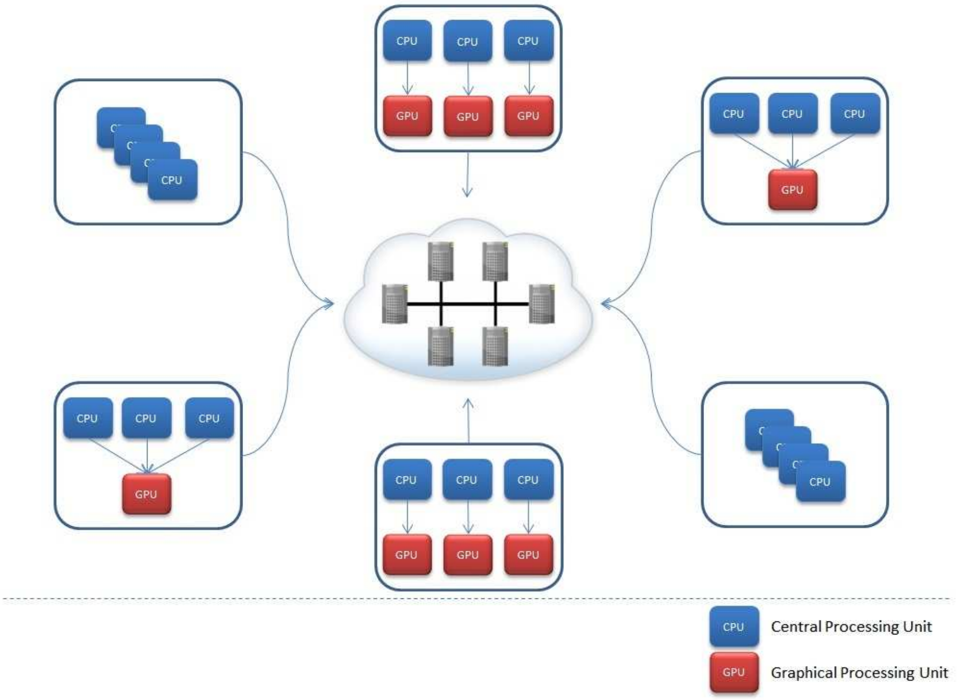


Figure 6.2: A simplified representation of a cluster/grid that contains interconnected heterogeneous ressources with single/multiple CPUs and single/multiple GPUs.

6.1.2 The B&B meta-algorithm

On heterogeneous platforms where processing elements have different performance characteristics, the portability and the ability to automatically tune algorithms to any hardware platform is a major challenge. With the aim to design a portable and adaptive heterogeneous parallel B&B algorithm, we propose a new meta-algorithm. This meta-algorithm selects the parallel B&B algorithm to be deployed according to the number of CPU cores and GPU devices in the target hardware configuration. As illustrated in *Algorithm 15*, the meta-algorithm proceeds by detecting the number of provided CPU cores and GPU devices. If the underlying grid node does not contain GPU devices, the multi-core B&B presented in Section 5.2 is used. If only a single CPU core coupled with a single GPU device is available, the B&B meta-algorithm runs the LL-GB&B algorithm presented in Section 4.4. When multiple CPU cores are coupled with a single GPU device, the PLL-GB&B algorithm is selected (see Section 5.4). Finally, if more GPUs are available, the LL-MultiGB&B algorithm is deployed (see Section 5.5).

Algorithm 15 Template of the proposed meta-algorithm.

```
max_nb_devices = Detect_GPU_Characteristics();
max_nb_cores = Get_CPU_Characteristics();

if  $max\_nb\_devices = 0$  then
    Run_MCB&B_algorithm();
end
else if  $max\_nb\_devices = 1$ && $max\_nb\_cores < 2$  then
    Run_LLGB&B_algorithm();
end
else if  $max\_nb\_devices = 1$ && $max\_nb\_cores >= 2$  then
    Run_PLLGB&B_algorithm();
end
else if  $max\_nb\_devices > 1$  then
    Run_LL-MultiGB&B_algorithm();
end
```

For each of the studied hardware configurations, three groups of parameters have been identified [Lastovetsky 2009] and tuned: problem parameters, algorithmic parameters and platform parameters.

- The problem parameters are those of the problem to be solved. For our B&B algorithm, these parameters correspond to the data related to the instance of the problem being solved and are defined accordingly. For example, for FSP, the instance of a problem determines the size of the matrices used to compute the lower bound, the size of the permutation representing the solution, etc.

- The algorithmic parameters represent different variations and configurations of the algorithm. These parameters do not change the semantics of the algorithm but can have an impact on its performance. For the proposed B&B, these parameters correspond to the size of the pool of nodes that are selected from the search tree and off-loaded to the GPU. The pool size depends strongly on the problem being solved and the underlying used GPU. Hence, it has to be tuned at runtime.

- The platform parameters are related to the execution heterogeneous platform such as the number of CPU cores, the number of GPU accelerators, etc. For B&B, according to the target hardware, a specific scenario is selected. Moreover, if a GPU device is available, the number of used blocks and threads are tunned according to the GPU configuration.

6.1.3 The B&B@Grid approach

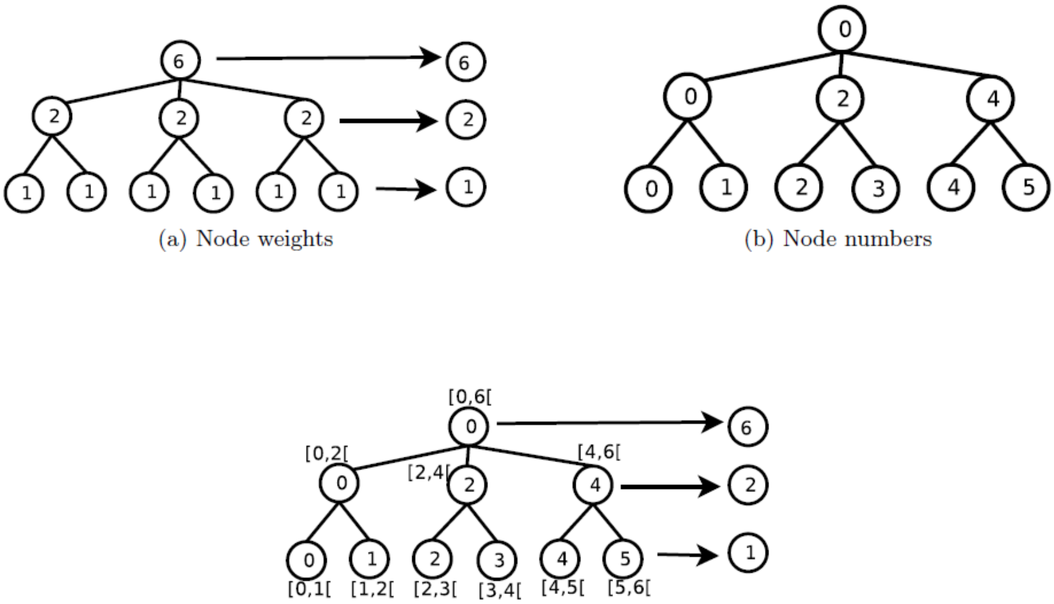
In [Mezmaz 2007a], the authors rethought the representation of the search space through an efficient encoding of work units to minimize the cost of information flowing in the network. The approach also includes efficient load sharing, fault tolerance and termination detection mechanisms. In particular, this approach was successfully applied to find the optimal solution of an unsolved flowshop hard instance, namely the Ta056 instance which belongs to the group of instances of Taillard with 50 jobs to be scheduled on 20 machines.

6.1.3.1 Tree encoding

The principle of the approach is constructed upon the assignment of a number to each pending node of the tree. The tree is labeled in such a way a sub-set of nodes is encoded by an interval: the numbers of any set of nodes always form an interval. The approach thus defines a pairing between the list of pending nodes and intervals. Thanks to its reduced size, the interval is used to optimize communication and check-pointing operations, while the list of pending nodes is used for exploration.

In order to retrieve a set of nodes from a given interval and *vice versa*, the approach defines two additional operators: the *fold* operator and the *unfold* operator. The fold operator deduces an interval from a list of pending nodes, and the unfold operator deduces an unique and minimal list of pending nodes from an interval. To define these two operators, three concepts are introduced: node's weight, node's number and node's range.

As illustrated in Figure 6.3 (a), the weight(p) of a node p corresponds to the number of leaves of the sub-tree generated from p. The number(p) is assigned to p according to the value of path(p) which is the set of nodes from the root to the node p, including both the root and p and to the rank of p which is the position of the node p among its



(a) Node weights

(b) Node numbers

(c) Intervals

Figure 6.3: The tree-based representation where each node has a unique number and contiguous nodes are represented by intervals.

sibling nodes. During the generation of the children of a given node, the rank of the first generated node is 0, the rank of the second generated node is 1, and so on. An example of how numbers are assigned is shown in Figure 6.3 (b). The range(p) of a node p, as a result, defines the interval that contains all the nodes of which the node p is the root node. As shown in Figure 6.3 (c) which is an example for coding a one-permutation tree of size n = 3, any set of contiguous nodes can be represented by an interval. Assuming that n is the size of the permutations, the size of the search space is S = n! and the whole search space can be represented by the global interval I = [0, n![. In Figure 6.3 (c), the global interval is equal to [0, 6[.

6.1.3.2 Master-Slave tree exploration

As quoted above, the B&B@GRID is based on a master-worker exploration model (see Section 2.6). The approach assumes that each worker process explores an interval and manages the local best solution while the master keeps a copy of all the not yet explored intervals in the *intervals* list and manages the global best solution found so far stored in the *solutions* variable.

The master continuously updates *intervals* by removing the subintervals that are al-ready explored and distributing others. Each time a worker process requests an interval (when it joins the calculation for the first time or when it finishes the exploration of its interval), the master assign an interval to it by partitioning an existing one into two parts. It also ensures efficient solution sharing by updating the *solutions* variable each time a worker informs that its best local solution is improved. The master is also responsible for notifying workers of the algorithm termination detection which occur when the list *intervals* of remaining sub-intervals becomes empty.

When distributing tree search works over multiple CPUs and GPUs, a major constraint have to be considered: GPU are substantially faster in evaluating tree nodes than a CPU. This observation has been confirmed in our previous chapters. In Section 4.5, for example, we report that our GPU-accelerated exploration of an interval that corresponds to a set of sub-problems is almost 160 times faster than the serial (single-core based) exploration of the same interval. Thereby, one may think that a fair partitioning between workers should grant GPU-endowed resources with longer intervals and consequently more nodes to explore which is not always valuable. Indeed, one should have in mind here that no assumption can be done about the amount of tree nodes that is explored in each interval. Besides, computers with multi-core CPUs might be more available in a grid than GPU-enabled ones which must be taken into consideration as well. The solution here is to assume that load sharing occurs according to the rate heterogeneous resources asks for works. Hence, if GPUs run out of work more frequently than CPUs does, they will ask the master for work more often and will be served correspondingly. In contrast, if a larger number of CPUs are provided than GPUs, no loss in performance will occur since the CPUs will be granted intervals that have been taken off from GPUs.

6.2 Experiments

In this section, we will introduce the experimental platform used for evaluating the performances of the distributed heterogeneous B&B and present the results of the experimentations conducted on standard FSP benchmarks.

6.2.1 Experimental platform

Our experiments have been conducted on the French nation-wide Grid'5000 Experimental Grid [Bolze 2006]. Grid'5000 is a scientific instrument launched in 2003 and designed to support experiment-driven research in all areas of computer science related to parallel, large-scale or distributed computing and networking. Grid'5000 inter-connects 10 sites

[0] Lille, Reims, Orsay1, Nancy, Rennes, Bordeaux, Toulouse, Grenoble, Lyon, Sophia-Antipolis.

in France via RENATER (the French academic network) and one site in Luxembourg via RESTENA (the luxembourgian academic network) (see Figure 6.4). The interconnection network is a Virtual Private Network (VPN) built on top of the RENATER network, composed of 10 Gbps network links using optical fibers. In May 2013, Gird'5000 comprises about 8.150 computational cores and more than 100 Tb of non-volatile storage capacity. All the physical machines used feature multi-core processors (varying from 2 to 24 depending on the machine). These resources are managed by different institutions and have to be reserved before being used. A reservation (commonly called a job) can be made under three modes: normal, deploy (where one can deploy his/her own operating system on the grid nodes) and best-effort (where some of the reserved resources can be reassigned to another job during the reservation).

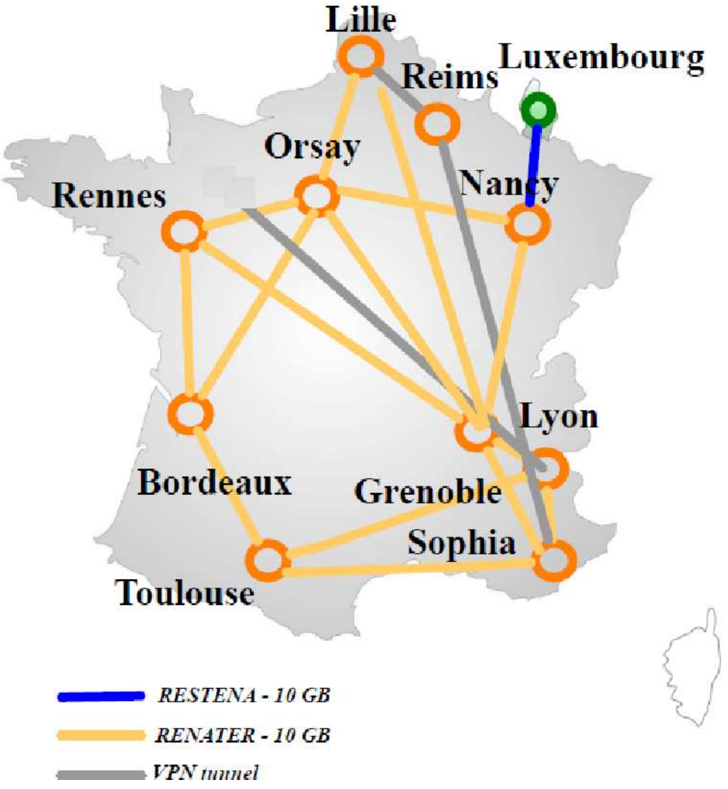


Figure 6.4: The experimental computational grid Grid'5000 [Gri 2003].

Grid'5000 is built in such a way to facilitate the deployment of user applications, and the retrieval of the results in a transparent way for the end user. For this, many management systems, middlewares and services were developed by different collaborating laboratories. The main advantage of Grid5000 is its degree of reconfigurability. This functionality, allows researchers to deploy and install the exact software environment they need for their experiments, making the platform the ideal tool for real-life experimentation. The reconfiguration mechanism allows the deployment of the constructed environment on the number of nodes that are requested for the experiments. Taking advantage of this capability, a user can very easily deploy his/her own cluster or grid upon the Grid'5000 platform.

6.2.2 Performance Evaluation

In the prospect of a thorough performance evaluation of the proposed template for running distributed heterogeneous B&B algorithms on computational grid, it is interesting to compare its performance with a same multi-level architecture that exploits distributed multi-core machines without GPU accelerators.

6.2.2.1 Experimental settings

In order to perform a fair comparison between the throughput of each considered approach, the different used architectures must have a same computational power in terms of theoretical peak of floating-point operations per second (FLOPS). FLOPS is a common measure of a computer's performance, especially in fields of scientific computing that make heavy use of floating-point calculations. Because it measures the computing ability, FLOPS is used as a benchmark indicator for rating the speed of supercomputers such as TOP500 [TOP500].

For the different machines used for the experiments, the number of potential GFLOPS is calculated from the theoretical ones provided by constructors [Intel Corporation 2011b, Intel Corporation 2011a, Intel Corporation 2011c]. The different workstations have been chosen according to the used GPU configuration i.e. in agreement with their computational power. As quoted in Section 3.6, the experiments have been carried out on using an Nvidia Tesla C2050. According to its constructor [NVIDIA Corporation 2010], the theoretical double precision floating-point performance peak of this GPU device is about 515 GFLOPS. Therefore, we used the set of machines described in Table 6.1 which has a double precision floating-point performance peak equivalent of greater than the NVIDIA Tesla's one. The objective is, indeed, to investigate the value-added of using GPU accelerators compared to an equivalent horsepower.

The theoretical FLOPS provided in [Intel Corporation 2011b, Intel Corporation 2011a, Intel Corporation 2011c] are given per CPU core. There-

Architecture	Configuration	
	Machines	GFLOPS
GPU	Tesla M2050	515
Grid	20 Intel Xeon E5520 (Edel cluster)	20 nodes * 8 cores/node * 36 GFLOPS = 5760
	10 Intel Xeon E5420 (Genepi cluster)	10 nodes * 8 cores/node * 40 GFLOPS = 3200
	20 Intel Xeon E5620 (Chimint cluster)	10 nodes * 8 cores/node * 38 GFLOPS = 3040
	25 Intel Xeon X5570 (Parapide cluster)	25 nodes * 8 cores/node * 46 GFLOPS = 9200

Table 6.1: Configurations of the distributed machines used for the experiments on Grid'5000.

fore for an octo-core workstation, such as the ones considered in our experiments, the total number of FLOPS is multiplied by eight i.e. the number of CPU cores provided. For example, from the Edel cluster (available in Grenoble site) we used 20 Intel Xeon E5520 nodes having each 8 CPU cores, the total GFLOPS of the workstations used is as a result equal to 5760 since the GFLOPS of an Intel Xeon E5520 is 36 GFLOPS.

6.2.2.2 Experimental results

The FSP benchmarks used to evaluate the performances of the HB&B@GRID algorithm are the Taillard's instances ranging from Ta021 to Ta030, aiming at scheduling 20 jobs on 20 machines. Table 6.2 reports the sequential resolution times for these instances obtained by using a single core of an Intel Xeon E5520 processor. These instances allow us to study the performances of our approach along the entire resolution process. For all experimented instances (Ta021-Ta030), the proposed heterogeneous GPU-accelerated B&B and the serial B&B lead to the same best known solutions reported in [Taillard 1993b].

Table 6.3 reports the execution times in seconds for a grid-enabled parallel B&B executed on several CPU cores. The rows correspond to the experimented instance ranging from Ta021 to Ta030. The columns correspond to the number of used CPU cores and the corresponding theoretical peak in GFLOPS. The results shows that whatever the used computational resource is, the execution times significantly vary from an instance to another even though they belong to the same group of instances with 20 jobs and 20 machines. For example, for the 15 CPU cores-based execution the resolution time is 9173.00 (s) for the instance Ta021 and about 29674.34 (s) for the instance Ta023. Therefore, our analysis of the former results will be based on the average of the obtained values.

The results show that on average when increasing the number of CPU cores involved

Instance	Resolution Time (Seconds)
Ta021	98056
Ta022	120806
Ta023	384300
Ta024	93544
Ta025	376094
Ta026	212135
Ta027	165082
Ta028	31444
Ta029	97699
Ta030	17257

Table 6.2: Sequential resolution times (seconds) for the instances Ta021-Ta030 corresponding to the group of instances with 20 jobs and 20 machines.

in the computations, the execution times necessary for solving the considered instances decrease accordingly. However, the slope is not linear. For example, when using 100 CPU cores the average execution time over all the instances is 2160 (s) while it is 1266 (s) when using 200 CPU cores. The expected theoretical execution time with 200 CPU cores should be twice less than the one registered with 100 CPU cores, while it is only 40% less in the performed experiments. This behavior is a characteristic of grid-based computations which are closely related to the underlying communication time. In such environments, higher accelerations can be reached as long as the computing time is not too much dominated by the communication time. Particularly, in our case, using 200 CPU cores obviously lead to upper communications time than in experiments where only 100 CPU cores are running which explains the drift away from the theoretically attainable performances.

Table 6.4 reports the execution times in seconds for the grid-enabled parallel B&B using several distributed GPUs. The rows correspond to the experimented instance ranging from Ta021 to Ta030. The columns correspond to the number of used GPU devices and the corresponding theoretical peak in GFLOPS. It is important to highlight here that the GPU devices are located on distinct machines. As noticed for the CPU-based executions, the resolution times significantly vary from an instance to another. Therefore, our analysis of these results will be also based on the average of the obtained values. On average, the results show that when increasing the number GPU devices involved in the computations, the execution times necessary for resolving the considered instances decrease accordingly.

Figure 6.5 exhibits the comparison between the average speedups for the instances 20

Used computational resources	15 CPU cores	50 CPU cores	100 CPU cores	200 CPU cores	500 CPU cores
Average Performances in GFLOPS	570	1900	3800	7600	19000
Ta021	9173.00	3149.70	1753.581	1106.254	762.633
Ta022	9885.30	3281.12	1958.009	912.393	767.208
Ta023	29674.34	12593.98	4984.879	2337.605	1691.613
Ta024	8118.44	2725.63	1260.54	762.863	592.873
Ta025	24719.96	8539.17	3787.595	2317.965	1735.098
Ta026	18317.21	5606.03	3937.5	2215.491	1233.828
Ta027	15975.82	5195.50	2225.421	1782.05	980.038
Ta028	2580.20	741.87	510.573	388.148	326.61
Ta029	4077.29	1177.69	881.006	562.789	447.666
Ta030	1372.82	368.30	309.163	282.771	235.494
Average Execution time	12389.44	4337.90	2160.83	1266.83	877.31

Table 6.3: Execution times (seconds) for the instances Ta021 to Ta030 using different scales of the distributed CPU-based version of the B&B.

machines × 20 jobs (compared to the serial execution reported in Table 6.2) obtained with the distributed GPU-based B&B and the distributed CPU-based version of the algorithm. For a same computational power, our approach for designing B&B algorithms on top of GPU accelerators is much more efficient than a large-scale distributed CPU-based execution. Indeed, for a computational power around 515 GFLOPS, the average acceleration calculated when using the distributed GPU-based B&B is ×88.52. For a same computational power (around 15 CPU cores) the speedup of the distributed CPU-based B&B is on average ×13.32. Even when using up to 100 CPU cores, using a single GPU is more efficient although the theoretical peak in GFLOPS of 100 CPU cores is almost 7 times more than the GPU device. Moreover, the GPU-based B&B executed on top of 3 distant GPU devices (1545 GFLOPS) runs on average 5 times faster than the CPU-based B&B executed on top of 50 CPUs which has an equivalent computational power (1900 GFLOPS). For the upper computational powers, the experimental results show that using 3 GPU devices is efficient enough to perform as good as using 500 distributed CPU cores and that using 5 GPU devices allows accelerations twice higher than those obtained using 500 CPU cores.

Used computational resources	1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs
Average Performances in GFLOPS	515	1030	1545	2060	2575
Ta021	993.41	567.08	411.10	322.93	302.68
Ta022	1405.13	852.67	455.82	385.69	335.79
Ta023	4913.19	2776.68	2207.35	1765.88	1119.04
Ta024	1017.73	598.66	439.42	351.54	288.36
Ta025	3854.63	2322.06	1702.45	1302.94	1090.92
Ta026	2631.50	1529.94	1102.75	883.55	655.04
Ta027	1842.34	1103.20	813.40	647.89	527.63
Ta028	368.97	217.04	153.73	123.56	100.46
Ta029	1058.66	619.10	445.65	376.25	297.47
Ta030	202.92	120.07	86.90	69.57	54.21
Average Execution time	1884.85	1070.65	781.86	622.98	477.16

Table 6.4: Execution times (seconds) for the instances Ta021 to Ta030 using different scales of the distributed GPU-accelerated version of the B&B.

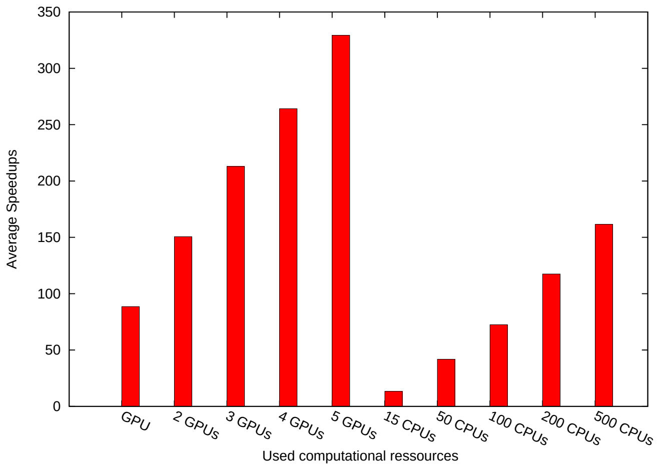


Figure 6.5: Comparison between the GPU-based Branch and Bound and the CPU-based distributed version of the algorithm.

6.3 Conclusion

In this chapter, we have presented the design and implementation of a large-scale heterogeneous B&B algorithm that uses the combined computing power of several multi-core machines and GPUs distributed within a cluster or a grid. Indeed, with the arrival of GPU resources and the advent of multi-core resources on computational grids, it is relevant to study the performances of a large-scale heterogeneous multi-core GPU-accelerated B&B algorithm and compare it with other successful large-scale algorithms.

- **Parallel heterogeneous B&B for computational grids** The approach consists in using a parallel heterogeneous B&B on top of several resources that are distributed inside one or several clusters and to make these ressources communicate efficiently. For achieving this, a B&B meta-algorithm is coupled with the B&B@GRID approach proposed in [Mezmaz 2007a]. Indeed, while B&B@GRID allows one to efficiently partition the B&B tree search among distant nodes, the meta-algorithm explores assigned sub-trees using the parallel B&B algorithm that best fits the targeted hardware configuration.

- **Performance of the distributed heterogeneous GPU-accelerated B&B algorithm** Comparison between the speedups obtained with the distributed GPU-based B&B and a distributed CPU-based algorithm, show that for a same computational power, the GPU-based approach is much more efficient. For a computational power equivalent to 50 CPU cores which is respectively equal to 3 times its computational power, our GPU-based B&B runs two times faster than the CPU-based version. Even when using up to 100 CPU cores, using a single GPU is more efficient although the theoretical peak in GFLOPS of 100 CPU cores is almost 7 times more than the GPU device. For the upper computational powers, experimental results show that using 3 GPU devices is efficient enough to perform as good as using 500 distributed CPU cores and that using 5 GPU devices allows accelerations two times higher than those obtained using 500 CPU cores.

Chapter 7

Conclusion and future works

GPU accelerators are nowadays available everywhere: in our laptops, in high performance computing workstations, in hybrid clusters and in computational grids and clouds. Such availability will increase with the arrival of exascale machines announced for 2018-2020. The challenge is, and will be in the near future, how to design and implement efficient algorithms for those GPU-enhanced computing environments. In this thesis, the focus is put on tree-based exact combinatorial optimization. Indeed, we have revisited the design and implementation of B&B algorithms for solving challenging COPs on top of GPU-enhanced heterogeneous computational platforms. Without loss of generality, the Flowshop Scheduling Problem (FSP) has been considered as a case study.

For achieving this, we have first investigated the use of a single CPU coupled with a GPU device tackling different issues related to the characteristics of GPU and the highly irregular nature of B&B: thread divergence, device memory management, adaptive sizing of transferred data and CPU-GPU data transfer optimization. Because it is data-parallel, intrinsically asynchronous and fine-grained and as it is the most time consuming operation in the B&B algorithm, our focus was first put only on the bounding operator. In the proposed GPU-based approach (GB&B), the selection, branching and pruning of the sub-problems are performed on CPU and the evaluation of their lower bounds is executed on the GPU device where each thread applies a computation function (lower bound) to a tree node. The analysis of the code implementing the lower bound function allows us to identify the thread/branch divergence sources and the data structures shared by the threads. For B&B algorithms or tree-based exploration algorithms in general, two insights came out (1) thread divergence reduction is interesting only for branches that contain long flows of instructions and (2) the use of shared memory for shared data structures allows a significant improvement of the performance. The proposed optimizations allow to reach speedups up to ×100 compared to a serial CPU-based B&B.

Even if the bounding operator represents more than 97% of the computation time of a B&B algorithm, its parallelization on GPU is not sufficient even if it is efficient. Indeed, further performance improvement can be obtained by reducing the overhead induced by the management of the pools of subproblems (tree nodes): their preparation on CPU, their transfer to GPU and the transfer of their associated lower bounds back to CPU. Therefore, our second focus in this work was to investigate the tree-based exploration model on GPU. For acheiving this, we extended the GB&B algorithm to the Low Latency GPU-accelerated B&B (LL-GB&B) algorithm which assumes to execute the branching, bounding and pruning operators on GPUs. Such parallelization requires to address the highly irregular nature of the explored search tree. Indeed, a preliminary experimental analysis has shown that even if a pool of nodes belong to the same level on the tree their branching may generate different numbers of children. To deal with such issue which arise in tree-based exploration algorithms in general be them B&B or not, the recommendation is to limit the granularity of each thread to the application of an operator (branching, bounding and pruning) to a single node. Further speedups compared to GB&B have been obtained reaching up to ×160.

With the ever-increasing demand for more computing performance, the HPC industry is moving toward a hybrid computing model, where GPUs and CPUs collaborate together to perform general-purpose computing tasks. Therefore, to be relevant to the growing number of heterogeneous computing systems, we have studied the combination of multiple CPU cores with one GPU and with several GPUs. The major issue addressed therein is mainly the repartition of pools of subproblems and their associated exploration computation between the CPU cores and the GPU device(s). When only one GPU is used, two scenarios have been explored leading to two approaches: concurrent LL-GB&B (rLL-GB&B) and cooperative LL-GB&B (pLL-GB&B). Due to synchronization overhead, even if the concurrent approach makes all the CPU cores contributing to the exploration process, it does not improve the performance compared to the single CPU core GPU-based approach. On the contrary, the cooperative approach enables an improvement of up to 36% compared to LL-GB&B. Therefore, as a general recommendation, one can say that when using this parallelization for tree-based algorithms, the exploration process including all the operators should be performed on GPU and that CPU cores should participate to the exploration process but and mainly to the preparation and transfer of the pools of tree nodes to GPU. We also suggest to use the CUDA data streaming to further reduce the cost of data transfer between CPU and GPU. When several GPUs are available, one efficient method is to split the kernels across these devices and to profit from the time where the kernels run in parallel to proceed with the next iterations of the algorithm. In addition, the direct memory access between different devices through the peer-to-peer memory access feature is highly suggested. Indeed, over a serial B&B, accelerations up to ×216.92 are reached for large instances of FSP and the more GPU devices are used, the better the speedups are.

With the arrival of GPU accelerators and the advent of multi-core processors on cluster and computational grids, our final contribution consists in proposing a large-scale version of the heterogeneous multi-core GPU-accelerated Branch and Bound algorithm. The proposed approach (H-B&B@GRID) consists in using a B&B meta-algorithm on top of several heterogeneous resources that are distributed over one or several cluster(s) and to ensure efficient work sharing and communication through the B&B@GRID approach. The algorithm consists in hierarchically combining two levels of parallelism by (1) dividing the B&B tree exploration among multiple distributed ressources using the B&B@GRID and (2) explore in parallel each sub-tree using the heterogeneous meta-algorithm. Using this portable, heterogeneous and self-adaptive approach allows us to acheive high performances comparable and sometimes even better than 500 distributed CPU cores when using 5 GPU devices of equivalent horsepower.

As future research directions for this work, we have identified some challenging perspectives summerized in the following:

- In this thesis, we have considered the Johnson's lower bound function as a case study for FSP but other bounds [Gharbi 2013] should be investigated. We believe that some functions might be efficient on single or multi-core CPU but inefficient on GPU because their parallelization on that device is hard and *vice versa*. A further challenging improvement of the B&B meta-algorithm consists in designing and implementing a library of lower bounds for the same problem on single and multi-core CPU and on GPU. As an additional level of adaptivity, the meta-algorithm will dynamically and automatically choose the implementation that best suits to the underlying execution machine.

- In the grid-enabled heterogeneous B&B approach, the CPU-level checkpointing mechanism proposed in B&B@Grid is used to deal with failures even if the CPU is coupled with a GPU device. The mechanism should be revisited to take into account GPU-level failures.

- We believe that the conclusions drawn from the experiments are the same whatever is the shape of (how irregular is) the tree and thus for any tree-based application. In the near future, we plan to revisit, within the framework of the Ph.D thesis of Rudi Leroy, the proposed approaches for other tree-based exploration algorithms especially B&X ones, X being Cut, Price, etc.

Other challenging perspectives are related to the recent evolutions in the context of

High Performance Computing (HPC). Indeed, HPC is moving from in-house to cloud-based computing, the software is becoming energy-aware, and GPU accelerators are more and more massively parallel and CPU-independent. The underlying future evolutions of the proposed approaches are outlined in the following:

- With the arrival of GPU resources in cloud computing infrastructures, the challenge is to revisit our proposed approaches on virtualized environments. This is a first natural step towards energy saving. So far, some experimental results reported in this thesis show that a single GPU is as efficient as 100 CPU cores but much less energy-consuming. However, the energy consumption criterion should be explicitely considered in the design and implementation of our approaches.

- Recently, the GPU technology has known an evolution coming up with a new generation of devices including advanced features. For instance, Kepler-based devices [NVIDIA Corporation 2012] allow dynamic parallelism and Nvidia GPUDirect. Dynamic parallelism consists in allowing the GPU device to generate new work by itself, synchronizing on results, and controling the scheduling of that work via dedicated and accelerated hardware paths, all without involving the CPU. Nvidia GPUDirect is a capability enabling GPUs located on servers distributed across the network to directly exchange data, in a peer-to-peer way, without passing through the CPU. A future challenge will consist in revisiting the proposed GPU-aware tree-based approaches for those modern GPUs taking into account their underlying advanced features.

Bibliography

[Aida 2002] K. Aida and Y. Futakata. *High-performance parallel and distributed computing for the BMI eigenvalue problem.* In 16th International Parallel and Distributed Processing Symposium, pages 71–78, 2002. (Cited on pages 19 and 29.)

[Aida 2005] K. Aida and T. Osumi. *A Case Study in Running a Parallel Branch and Bound Application on the Grid.* SAINT'05: The 5th IEEE Symposium on Applications and the Internet, vol. 27, no. 2, pages 164–173, 2005. (Cited on page 29.)

[Alerstam 2010] E. Alerstam, W. Chun Yip Lo, T. D. Han, J. Rose, S. Andersson-Engels and L. Lilge. *Next-generation acceleration and code optimization for light transport in turbid media using GPUs.* Biomedical Optics Express, pages 658–675, 2010. (Cited on page 57.)

[Allahverdi 1999] A. Allahverdi, J.N.D Gupta and T. Aldowaisan. *A review of scheduling research involving setup considerations.* Omega, vol. 27, no. 2, pages 219–239, 1999. (Cited on page 8.)

[Allen 1997] R. Allen, L. Cinque, S. Tanimoto, L. Shapiro and D. Yasuda. *A Parallel Algorithm for Graph Matching and Its MasPar Implementation.* IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 5, pages 490–501, 1997. (Cited on page 1.)

[Anderson 2002] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky and D. Werthimer. *SETI@home: an experiment in public-resource computing.* Communications of the ACM, vol. 45, no. 11, pages 56–61, 2002. (Cited on page 29.)

[Bader 2005] D. A. Bader, W. E. Hart and C. A. Phillips. *Parallel Algorithm Design for Branch and Bound.* vol. 76, pages 5–1–5–44, 2005. (Cited on pages 1 and 19.)

[Baker 2002] M. Baker, R. Buyya and D. Laforenza. *Grids and grid technologies for wide-area distributed computing.* Software: Practice and Experience, vol. 32, no. 15, pages 1437–1466, December 2002. (Cited on page 27.)

[Barreto 2010] L. Barreto and M. Bauer. *Parallel branch and bound algorithm-a comparison between serial, openmp and mpi implementations.* In Journal of Physics: Conference Series, vol. 256, 2010. (Cited on pages 2 and 27.)

[Bendjoudi 2012] A. Bendjoudi, N. Melab and E-G. Talbi. *Hierarchical branch and bound algorithm for computational grids.* Future Generation Computer Systems, vol. 28, no. 8, pages 1168–1176, 2012. (Cited on pages 2 and 30.)

[Bendjoudi 2013] A. Bendjoudi, N. Melab and E-G. Talbi. *FTH-B&B: a Fault-Tolerant Hierarchical Branch and Bound for Large Scale Unreliable Environments.* IEEE Transactions on Computers, 2013. (Cited on page 30.)

[Bolze 2006] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O.Richard, E-G. Talbi and I. Touche. *Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed.* volume 20, pages 481–494, Thousand Oaks, CA, USA, November 2006. Sage Publications, Inc. (Cited on page 115.)

[Bonney 1976] M.C. Bonney and S.W. Gundry. *Solutions to the constrained flowshop sequencing problem.* Operational Research Quarterly, page 869, 1976. (Cited on page 8.)

[Bradford 1996] N. Bradford, B. Dick and F.J. Proulx. O'Reilly, 1996. (Cited on pages 87 and 89.)

[Brodtkorb 2010] A. R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik and O.O. Storaasli. *State-of-the-art in heterogeneous computing.* Journal of Scientific Programming, vol. 18, no. 1, pages 1–33, 2010. (Cited on page 86.)

[Buchty 2012] R. Buchty, V. Heuveline, W. Karl and J.P. Weiss. *A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators.* Concurrency and Computation: Practice and Experience., vol. 24, no. 7, pages 663–675, 2012. (Cited on page 86.)

[Carneiro 2011] T. Carneiro, A.E Muritiba, M. Negreiros and G.A. Lima de Campos. *A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU.* In 23rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2011. (Cited on pages 2, 19, 24, 69, 72 and 86.)

[Casado 2008] L. G. Casado, J.A. Martinez, I. Garcia and E. M. T. Hendrix. *Branch-and-Bound interval global optimization on shared memory multiprocessors.* Optimization Methods Software, vol. 23, no. 5, pages 689–701, 2008. (Cited on pages 2, 19 and 26.)

[Chakroun 2011] I. Chakroun, A. Bendjoudi and N. Melab. *Reducing Thread Divergence in GPU-based B&B Applied to the Flow-shop problem.* In 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011), Torun, Pologne, September 2011. (Cited on pages 20 and 24.)

[Chakroun 2012] I. Chakroun and N. Melab. *An Adaptive Multi-GPU Based Branch-and-Bound. A Case Study: The Flow-Shop Scheduling Problem.* In 14th IEEE International Conference on High Performance Computing and Communication, HPCC'12, Liverpool, United Kingdom, June 25-27, pages 389–395, 2012. (Cited on page 98.)

[Crainic 2006] T.G. Crainic, B. Le Cun and C. Roucairol. Parallel branch-and-bound algorithms, pages 1–28. John Wiley & Sons, Inc., 2006. (Cited on pages 16 and 143.)

[Diconstanzo. 2007] A. Diconstanzo. *Branch-and-bound with peer-to-peer for large-scale grids.* In PhD thesis, Ecole doctorale STIC, Sophia Antipolis, France, 2007. (Cited on pages 30 and 31.)

[Djamai 2011] M. Djamai, B. Derbel and N. Melab. *Distributed B&B: A Pure Peer-to-Peer Approach.* 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, vol. 0, pages 1788–1797, 2011. (Cited on pages 2 and 31.)

[Djerrah 2006] A. Djerrah, B. Le Cun, V. Cung and C. Roucairol. *Bob++: Framework for Solving Optimization Problems with Branch-and-Bound methods.* In 15th IEEE International Symposium on High Performance Distributed Computing, pages 369–370, 2006. (Cited on page 19.)

[Evtushenko 2009] Y. Evtushenko, M. Posypkin and I. Sigal. *A framework for parallel large-scale global optimization.* Computer Science - Research and Development, vol. 23, pages 211–215, 2009. (Cited on page 26.)

[Finkel 1987] R. Finkel and U. Manber. *DIB-a distributed implementation of backtracking.* ACM Transactions on Programming Languages and Systems, vol. 9, no. 2, pages 235–256, March 1987. (Cited on page 30.)

[Foster 1998] I. Foster and C. Kesselman. *Computational Grids*, 1998. (Cited on page 27.)

[Fung 2007] W. L. Fung, I. Sham, G. Yuan and T.M Aamodt. *Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow.* In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 40, pages 407–420. IEEE Computer Society, 2007. (Cited on pages 21, 40 and 41.)

[Garey 1976] M.R. Garey, D.S. Johnson and R. Sethi. *The complexity of flow-shop and job-shop scheduling*. Mathematics of Operations Research, vol. 1, pages 117–129, 1976. (Cited on pages 1 and 9.)

[Gendron 1994] B. Gendron and T.G. Crainic. *Parallel Branch-and-Bound Algorithms: Survey and Synthesis*. Operations Research, vol. 42, no. 6, pages 1042–1066, 1994. (Cited on pages 2, 10, 16, 143 and 144.)

[Gharbi 2013] A. Gharbi and A. Mahjoubi. *New Lower Bounds for Flow Shop Scheduling*. International Journal of Humanities and Management Sciences (IJHMS), 2013. (Cited on page 125.)

[Gri 2003] *French national gird*. In https://www.grid5000.fr, 2003. (Cited on pages viii and 116.)

[Han 2011] T. D. Han and T. S. Abdelrahman. *Reducing branch divergence in GPU programs*. In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, New York, NY, USA, 2011. ACM. (Cited on pages 40, 41, 55, 57 and 58.)

[Hejazi 2005] S. Reza Hejazi and S. Saghafian. *Flowshop-scheduling problems with makespan criterion: a review*. International Journal of Production Research, vol. 43, no. 14, pages 2895–2929, 2005. (Cited on page 8.)

[Iamnitchi 2000] A. Iamnitchi and I. Foster. *A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems*. In Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing, ICPP '00, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on page 30.)

[Intel Corporation 2011a] Intel Corporation. *Intel Xeon Processor 5400 Series*. In http://download.intel.com/support/processors/xeon/sb/xeon_5400.pdf, April 2011. (Cited on page 117.)

[Intel Corporation 2011b] Intel Corporation. *Intel Xeon Processor 5500 Series*. In http://download.intel.com/support/processors/xeon/sb/xeon_5500.pdf, April 2011. (Cited on page 117.)

[Intel Corporation 2011c] Intel Corporation. *Intel Xeon Processor 5600 Series*. In http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf, April 2011. (Cited on page 117.)

[Janakiram 1988] V. Janakiram, EF. Gehringer, DP. Agrawal and R. Mehrotra. *A randomized parallel branch-and-bound algorithm.* International Journal of Parallel Programming, vol. 17, no. 3, pages 277–301, 1988. (Cited on page 17.)

[Jang 2011] B. Jang, D. Schaa, P. Mistry and D. Kaeli. *Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures.* IEEE Transaction on Parallel and Distributed Systems, vol. 22, no. 1, pages 105–118, 2011. (Cited on page 40.)

[Johnson 1954] S.M. Johnson. *Optimal two and three-stage production schedules with setup times included.* Naval Research Logistis Quarterly, vol. 1, pages 61–68, 1954. (Cited on pages 3, 8 and 35.)

[J.R.Jackson 1956] J.R.Jackson. *An Extension of Johnson's results on Job-Lot Scheduling.* Naval Research Logistis Quarterly, 1956. 3:3. (Cited on page 35.)

[King 1980] J. R. King and A. S. Spachis. *Heuristics for flow-shop scheduling.* International Journal of Production Research, vol. 18, no. 3, page 345â357, 1980. (Cited on page 8.)

[Kirk 2010] D. B. Kirk and Wen mei W. Hwu. Programming massively parallel processors: A hands-on approach. Morgan Kaufmann Publishers Inc. 1st edition, 2010. (Cited on page 137.)

[Kumar 1984] V. Kumar and L.N. Kanal. *Parallel Branch-and-Bound Formulations for AND/OR Tree Search.* IEEE Transactions on Pattern Analysis and Machine Intelligence, no. 6, pages 768–778, 1984. (Cited on page 17.)

[Lalami 2012] M.E. Lalami. *Contribution à la résolution de problèms d'optimisation combinatoire: méthodes séquentielles et parallèles.* Université de Toulouse III - Paul Sabatier., 2012. (Cited on pages 2, 19, 24, 77 and 86.)

[Lastovetsky 2009] A. Lastovetsky and J. Dongarra. High performance heterogeneous computing. Wiley, 2009. (Cited on pages 86 and 112.)

[Lenstra 1978] J.K. Lenstra, B.J. Lageweg and A.H.G. Rinnooy Kan. *A General bounding scheme for the permutation flow-shop problem.* Operations Research, vol. 26, no. 1, pages 53–67, 1978. (Cited on pages 3 and 35.)

[L.G.Mitten 1959] L.G.Mitten. *Sequencing n jobs on two machines with arbitrary time lags.* Management Science, 1959. (Cited on page 35.)

[Luong 2011] T.V. Luong. *Métaheuristiques parallèles sur GPU.* Université des Sciences et Technologie de Lille, 2011. (Cited on pages 2 and 23.)

[Mahmoudi 2012] S.A. Mahmoudi and P. Manneback. *Efficient Exploitation of Heterogeneous Platforms for Images Features Extraction.* 10/2012 2012. (Cited on page 23.)

[Mahmoudi 2013] S.A. Mahmoudi. *Traitement efficace d'objects multimédia sur architectures parallèles et hétérogènes.* PhD thesis, Université de Mons, 2013. (Cited on pages 22 and 23.)

[Mans 1995] B. Mans, T. Mautor and C. Roucairol. *A parallel depth first search branch and bound algorithm for the quadratic assignment problem.* European Journal of Operational Research, vol. 81, no. 3, pages 617 – 628, 1995. (Cited on page 19.)

[Melab 2005] N. Melab. *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul*, 2005. (Cited on pages 16 and 143.)

[Meng 2010] J. Meng, D.Tarjan and K.Skadron. *Dynamic warp subdivision for integrated branch and memory divergence tolerance.* In Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10, pages 235–246, New York, NY, USA, 2010. ACM. (Cited on pages 21 and 40.)

[Mezmaz 2007a] M. Mezmaz. *Une approche efficace pour le passage sur grilles de calcul de méthodes d'optimisation combinatoire.* PhD thesis, Université des Sciences et Technologies de Lille 1, 2007. (Cited on pages 2, 5, 16, 29, 31, 51, 109, 113 and 122.)

[Mezmaz 2007b] M. Mezmaz, N. Melab and E-G. Talbi. *A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems.* In In Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS). Long Beach, California, March 2007. (Cited on page 51.)

[Miller 1993] Donald L. Miller and Joseph F. Pekny. *The Role of Performance Metrics for Parallel Mathematical Programming Algorithms.* vol. 5, no. 1, pages 26–28, 1993. (Cited on page 17.)

[NVIDIA Corporation 2008] NVIDIA Corporation. *COMPUTE VISUAL PROFILER User Guide.* In https://developer.nvidia.com/nvidia-visual-profiler, 2008. (Cited on pages 51 and 55.)

[NVIDIA Corporation 2010] NVIDIA Corporation. *TESLA C2050 / C2070 GPU Computing Processor (NVIDIA TESLA | DATASHEET) .* In http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10 _lores.pdf, July 2010. (Cited on pages 48 and 117.)

[NVIDIA Corporation 2011a] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide.* In http://developer.nvidia.com/nvidia-gpu-computing-documentation, June 2011. (Cited on pages 20, 43 and 99.)

[NVIDIA Corporation 2011b] NVIDIA Corporation. *Peer-to-Peer & Unified Virtual Addressing.* In http://developer.download.nvidia.com/CUDA/training/cuda _webinars_GPUDirect_uva.pdf, 2011. (Cited on pages viii, ix, 100, 138 and 140.)

[NVIDIA Corporation 2012] NVIDIA Corporation. *Kepler TM GK110 White paper.* In www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. (Cited on page 126.)

[Papadimitriou 1982] C.H. Papadimitriou and K.Steiglitz. Combinatorial optimization: algorithms and complexity. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982. (Cited on page 10.)

[Paulavicius 2009] R. Paulavicius and J. Zilinskas. *Parallel Branch and Bound Algorithm with Combination of Lipschitz Bounds over Multidimensional Simplices for Multicore Computers.* In Parallel Scientific Computing and Optimization, volume 27 of *Springer Optimization and Its Applications*, pages 93–102. Springer New York, 2009. (Cited on page 27.)

[Quinn 1990] J. Quinn. *Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer.* IEEE Trans. Comput., vol. 39, no. 3, pages 384–387, 1990. (Cited on pages 2 and 19.)

[Roucairol 1996] C. Roucairol. *Parallel processing for difficult combinatorial optimization problems.* European Journal of Operational Research, vol. 92, no. 3, pages 573 – 590, 1996. (Cited on page 16.)

[Ryoo 2008] S. Ryoo, C.I. Rodrigues, S.S. Stone, J.A. Stratton, S.Z. Ueng, S.S. Baghsorkhi and Wen mei W. Hwu. *Program optimization carving for GPU computing.* Journal of Parallel and Distributed Computing, vol. 68, no. 10, pages 1389–1401, 2008. (Cited on pages 22 and 40.)

[Stone 2010] J. E. Stone, D. J. Hardy, I. S. Ufimtsev and K. Schulten. *GPU-accelerated molecular modeling coming of age.* Journal of Molecular Graphics and Modelling, vol. 29, no. 2, pages 116 – 125, 2010. (Cited on page 137.)

[Sung 2010] I-Jui. Sung, J.A. Stratton A. and Wen-Mei W. Hwu. *Data layout transformation exploiting memory-level parallelism in structured grid many-core applications.* In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 513–522, New York, NY, USA, 2010. ACM. (Cited on page 40.)

[Taillard 1993a] E. Taillard. *Benchmarks for basic scheduling problems.* European Journal of Operational Research, vol. 64, no. 2, page 278 â 285, 1993. (Cited on pages 3, 36, 46 and 48.)

[Taillard 1993b] E. Taillard. *Taillard's FSP benchmarks.* In http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html, 1993. (Cited on pages 14, 36, 46, 48 and 118.)

[TOP500] TOP500. *TOP 500 supercomputer sites.* In http://www.top500.org/. (Cited on pages i, iii and 117.)

[Trienekens 1992] H.W.J.M. Trienekens and A. de Bruin. *Towards a Taxonomy of Parallel Branch and Bound Algorithms.* Rapport technique, 1992. (Cited on pages 16 and 143.)

[Tschöke 1995] S. Tschöke, R. Lüling and B. Monien. *Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network.* In Proceedings of the 9th International Symposium on Parallel Processing, IPPS '95, pages 182–189, Washington, DC, USA, 1995. IEEE Computer Society. (Cited on pages 2 and 19.)

[Yang 2010] Y. Yang, P. Xiang, J. Kong and H. Zhou. *A GPGPU compiler for memory optimization and parallelism management.* In Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM. (Cited on pages 22 and 40.)

[Zhang 2010] E. Z. Zhang, Y. Jiang, G. Ziyu and S. Xipeng. *Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping.* In Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, pages 115–126, New York, NY, USA, 2010. ACM. (Cited on pages 40 and 41.)

International Publications

International Journals

- I. Chakroun, N. Melab, M. Mezmaz and D. Tuyttens.
  "Combining multi-core and GPU computing for solving combinatorial optimization problems". Journal of Parallel and Distributed Computing (JPDC) - Elsevier - In Press.

- I. Chakroun, M.Mezmaz, N. Melab, and A.Bendjoudi.
  " Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm". Concurrency and Computation: Practice and Experience - vol 25, N° 8, pages 1121-1136, 2013 - John Wiley & Sons..

- N. Melab, I. Chakroun, and A. Bendjoudi.
  "GPU-accelerated Bounding for Branch-and-Bound applied to a Permutation Problem using Data Access Optimization". Concurrency and Computation: Practice and Experience - John Wiley & Sons (Accepted with minor revision).

- I. Chakroun and N. Melab.
  "Towards an heterogeneous and adaptive parallel Branch-and-Bound algorithm." Journal of Computer and System Sciences - Elsevier (Under revision).

International Conferences

- I. Chakroun and N. Melab. "Operator-level GPU-accelerated Branch and Bound algorithms". International Conference on Computational Science, ICCS'13. Barcelona, Spain, June 5-7, 2013.

- N. Melab, I. Chakroun , M. Mezmaz and D.Tuyttens. " A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem". 14th IEEE International Conference on Cluster Computing, CLUSTER'12. China, Beijin, September 24-28, 2012.

- I. Chakroun and N. Melab. "An Adaptative Multi-GPU based Branch-and-Bound. A Case Study: the Flow-Shop Scheduling Problem". 14th IEEE International Conference on High Performance Computing and Communications, HPCC'12. United Kingdom, Liverpool, June 24-27, 2012.

- I. Chakroun, A. Bendjoudi, and N.Melab. “Reducing Thread Divergence in GPU-based B&B Applied to the Flow-shop problem”. 9th International Conference on Parallel Processing and Applied Mathematics PPAM’11, LNCS. Poland, Torun, September 11-14, 2011.

Book Chapter

- I.Chakroun and N.Melab. “GPU-accelerated Tree-based Exact Optimization Methods”. Designing scientific applications on GPUs. CRC Press, Taylor & Francis Group.

Appendix A

Graphics Processing Units

A.1 State of GPU computing

Graphics Processing Units (GPUs) are at the leading edge of many-core parallel computational platforms in several research fields. For years, the use of graphics processors was dedicated to high-definition 3D graphics. Driven by the demand for high-definition 3D graphics on personal computers, GPUs have evolved into a highly parallel, multi-threaded and many-core environment endowed with great computational horsepower and a very high memory bandwidth compared to traditional CPUs. Nowadays, the massive data processing capability of modern GPUs is attracting researchers to explore mapping more general non-graphics computations onto them [Kirk 2010].

The GPU is especially well-suited for fine-grained, data-parallel computations consisting of thousands of independent threads executing the same program concurrently. It excels with programs that are executed on many data elements in parallel and with a high ratio of arithmetic operations to memory operations. Indeed, GPUs have a large number of arithmetic units with a limited cache and few control units. More of its transistors are used for data processing rather than data caching and flow control.

A GPU is organized into an array of highly threaded streaming multiprocessors (SMs). Each streaming multiprocessor contains a set general purpose arithmetic units called streaming processors (SPs) and a number of special function units (SFU) used for computing special algebraic functions not provided by the SPs. Memory load/store units (LDST), texture units (TEX), fast on-chip data caches (shared memory and constant cache), and a high-bandwidth main memory system provide the GPU with sufficient operand bandwidth to keep the arithmetic units productive. Figure A.1 [Stone 2010] depicts a simplified hardware block diagram for the NVIDIA Fermi GPU architecture.

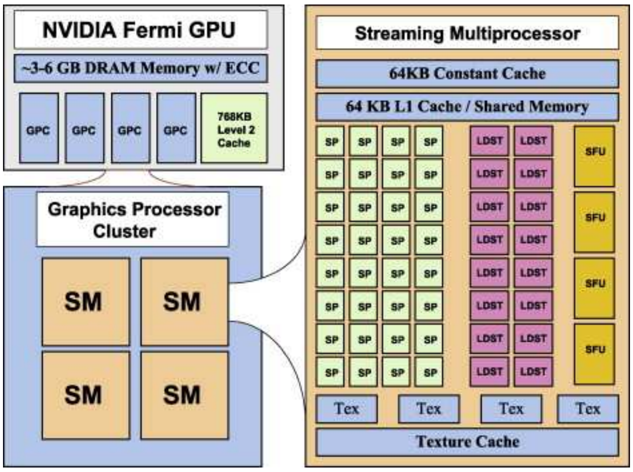


Figure A.1: A simplified hardware block diagram for the NVIDIA Fermi GPU architecture [NVIDIA Corporation 2011b].

A.2 The Compute Unified Device Architecture programming model

CUDA (Compute Unified Device Architecture) is a parallel computing environment, which provides an application programming interface for NVIDIA architectures. CUDA comes with a software environment that allows developers to use C as a high-level programming language. A CUDA program is a C-like unified source code encompassing both host and device code. The parts of the program that exhibit little or no data parallelism are implemented in host code, other parts with rich amount of data parallelism are implemented in the device code. The NVIDIA C compiler (nvcc) separates the two parts during the compilation process.

The execution starts with host (CPU) execution. When a kernel function is invoked, or launched, the execution is moved to a device (GPU), where a large number of threads are generated and execute the kernel function many times in parallel leading to a valuable data parallelism. All the threads that are generated by a kernel are organized onto thread blocks and grids of thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread identifier within its thread block, program counter, registers, per-thread private memory, inputs, and output results. Threads are partitioned into groups of 32 threads called warps which execution is scheduled following a time-sharing strategy. For each instruction of the kernel, the multiprocessor selects a warp that is ready to be run. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path. A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block identifier within its grid. A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

CUDA-enabled devices use several memory spaces with different characteristics that reflect their distinct usages. These memory spaces include global, local, shared, texture, and registers.

A.3 Device Memory Spaces

In the CUDA parallel programming model, different memory spaces are defined. As illustrated in Figure A.2, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing. Grids of thread blocks share results in global memory space after kernel global synchronization.

Global Memory Communication between the host and the GPU occurs through the global memory. This memory has the lifetime of the application and is accessible to all threads of all kernels. The global memory is the largest in size but has a high access latency. In some GPU configurations (with compute capability 1.x), this memory is not cached and its access is slow, therefore the accesses to global memory (read/write operations) need to be minimized. Reads from global memory is cached only on devices that support compute capability 2.0.

Registers Scalar variables that are declared in the scope of a kernel function are stored in register memory by default. Register variables are private to the thread. Threads in the same block will get private versions of each register variable. Register variables only exists as long as the thread exists. Once the thread finishes execution, a register variable cannot be accessed again. Register memory access is very fast, but the number of registers that are available per block is limited.

Local Memory Any variable that cannot fit into the register space allowed for the kernel will spill-over into local memory. Like registers, local memory is private to the thread. Like registers, variables in local memory have the lifetime of the thread: once the thread is finished, the local variable is no longer accessible. However, local memory is so named because its scope is local to the thread, not because of its physical location. In

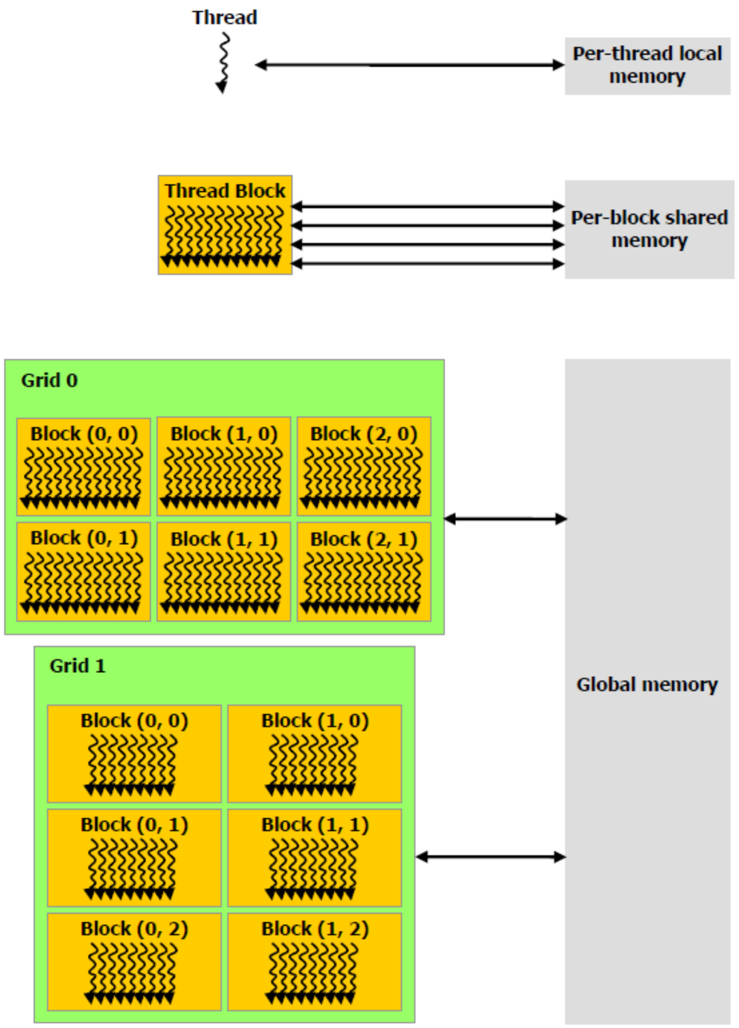


Figure A.2: CUDA hierarchy of threads, blocks and grids with corresponding per-thread private, per-block shared and per-application global memory spaces [NVIDIA Corporation 2011b].

fact, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory. Like global memory, local memory is not cached on devices of compute capability 1.x.

Shared Memory Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory. Shared memory is shared by threads of each thread block, it provides a way for threads to communicate within the same block. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.

Constant Memory Constant memory is a cached read only memory. Constant memory provides one cycle of latency when there is a cache hit even though constant memory resides in device (global) memory. The constant cache is written only by the host and is persistent across kernel calls within the same application. Access to data in constant memory can range from one cycle for in cache data to hundreds of cycles depending on cache locality.

L1 cache Devices of compute capability 2.x come with an L1 cache hierarchy that is used to cache local and global memory accesses. The same on-chip memory is used for both L1 and shared memory, and the amount dedicated to L1 versus shared memory is configurable for each kernel call. Experimentation is recommended to find out the best combination for a given kernel: 16 KB or 48 KB of L1 cache (and vice versa for shared memory) with or without global memory caching in L1 and with more or less local memory usage.

Texture Memory The read-only texture memory space is cached. Therefore, a texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache which is optimized for 2D spatial locality. On devices of compute capability 1.x, some kernels achieve a speedup when using (cached) texture fetches rather than regular global memory loads. This optimization can be counter-productive on devices of compute capability 2.x, however, since global memory loads are cached in L1 and the L1 cache has higher bandwidth than the texture cache.

Appendix B

Parallelization strategies for Branch and Bound algorithms

The parallelization of B&B is widely studied in the literature and different classifications have been proposed [Trienekens 1992, Gendron 1994, Crainic 2006, Melab 2005].

B.1 Classification of Granic *et al.*

Granic *et al.* [Crainic 2006] identified a classification with two main types of parallelization: node-based parallelization and tree-based parallelization.

- **Node-based parallelization** introduces parallelism when performing the operations on a single problem. It aims to accelerate the search by executing in parallel a particular operation, mainly associated to the subproblem: computation in parallel of lower or upper bound, evaluation in parallel of sons, and so on. This type of parallelism has no influence on the general structure of the B&B algorithm and is particular to the problem being solved.
- **Tree-based parallelization** consists in building and/or exploring the solution tree in parallel by performing operations on several sub-problems simultaneously. This coarse-grained type of parallelism affects the general structure of the B&B algorithm and yields to irregularity.

B.2 Classification of Trienekens *et al.*

Trienekens *et al.* [Trienekens 1992] have proposed two levels of parallelization for B&B algorithms: low level parallelization and high level parallelization.

- **Low level parallelization**: Only part of the serial Branch and Bound algorithm is parallelized in such a way that the interactions between the parallelized part and the other parts of the algorithm do not change. For example, the computation of the bound, the selection of the next sub-problem to be branched, the parallelization of the application of the elimination rule. Because the interactions between the various parts of the algorithm are not changed, low level parallelization does not change the semantics of the B&B algorithm as a whole. It means that the overall behavior of the created parallel B&B algorithm is similar to the behavior of the original serial Branch and Bound algorithm.

- **High Level parallelization**: For high level parallelization, the effects and consequences of the parallelism introduced are not restricted to a particular part of the B&B algorithm, but influence the semantics of the algorithm as a whole. The work performed by the parallel algorithm does not need to be the same as the work performed by the serial algorithm. The order in which the work is performed can differ, and it is even possible that some parts of the work performed by the parallel algorithm are not performed by the serial algorithm, or *vice versa*. For example, the parallel exploration of the search tree can lead to early improve the best solution and therefore allows to prune some branches of the tree that are explored in the serial version of the algorithm.

B.3 Classification of Gendron *et al.*

Gendron *et al.* [Gendron 1994] identified three main approaches for designing parallel B&Bs according to the degree of parallelism potentially provided by the search tree:

- **Parallelism of type 1** introduces parallelism when the operations are performed on generated sub-problems. For instance, executing the bounding operation in parallel for each sub-problem to accelerate the execution. This type of parallelism has no impact on the general structure of the B&B algorithm and is particular to the problem to be solved.

- **Parallelism of type 2** consists of building the search tree in parallel by simultaneously applying the B&B operators on several sub-problems. This type of parallelism may affect the design and semantics of the algorithm.

- **Parallelism of type 3** means that several search trees are generated in parallel. The trees are explored using different variants of the operations (branching, bounding, and pruning), and the information generated when building one tree can be used for the construction of another.

Another classification based on the concept of work pool, which is a memory location processes pick from find and store in their units of work, is also proposed in [Gendron 1994].

- Single work pool: only one memory location is used to store the units of work. Single pool B&B algorithms are mainly implemented on shared-memory architectures where threads access concurrently the common work pool to pick and insert subproblems. On distributed-memory architectures, this model can be implemented using the master/slave paradigm: one process called master manages the work pool and sends work units to other processes called slaves.

- Multiple work pools: in multiple pool algorithms, several memory locations are used. Several organization schemes are possible: each work pool is associated with exactly one process, each group of processes share a work pool or each process has its own work pool and there is a global pool shared by all processes.