Multi-level Parallel Branch-and-Bound Algorithms for Solving Permutation Problems on GPU-accelerated Clusters

Mohand Mezmaz

École Doctorale Sciences pour l'Ingénieur
Université Lille Nord de France

HABILITÉ À DIRIGER DES RECHERCHES
DISCIPLINE : INFORMATIQUE

présentée et soutenue publiquement,
le 18 septembre 2020, par

Mohand Mezmaz

Multi-level Parallel Branch-and-Bound Algorithms for Solving Permutation Problems on GPU-accelerated Clusters

Algorithmes Parallèles Multi-niveaux de Séparation et Évaluation pour la Résolution de Problèmes de Permutation sur des Clusters Accélérés par GPUs

Acknowledgments

Je tiens tout particulièrement à exprimer ma profonde gratitude à Nouredine Melab, Garant de mon HDR: tannemirt! C'est une grande chance d'avoir continué, après ma thèse, de bénéficier de la rigueur de Nouredine, de sa motivation et de ses conseils.

Mes remerciements sincères vont également à Enrique Alba, Frédéric Saubion et Pierre Sens de m'avoir fait l'honneur de rapporter cette HDR, ainsi qu'a Pierre Boulet, Pascal Bouvry et Amir Nakib d'avoir accepté de faire partie de mon jury. Leur grande implication dans cette soutenance m'a permis de présenter mon HDR, même dans des conditions particulières.

Je voudrais aussi adresser un grand merci à Daniel Tuyttens, Chef du Service MARO de l'Université de Mons. Par sa confiance et ses conseils, Daniel a beaucoup contribué à mes recherches depuis toutes ces années.

Mes remerciements sincères vont également à Rudi Leroy et Jan Gmys, que j'ai co-encadré durant leurs thèses. Leurs contributions constituent les ingrédients secrets du succès de cette HDR.

Merci à tous les collègues et anciens collègues du Service MARO d'avoir participé à cet esprit d'équipe formidable.

Mes derniers remerciements et non les moindres vont à ma famille, et en particulier à mes parents. Leurs efforts, leurs patiences et leurs encouragements sont déterminants pour cette réussite.

Contents

CHAPTER 1

Main research activities since my PhD

The research works done in my PhD thesis, defended at the end of 2007, focused on the parallelization of combinatorial optimization algorithms. During this thesis, we mainly presented a new approach, called B&B@Grid [MMT07b], for the parallelization of Branch-and-Bound (B&B) algorithms.

The results of our experiments show that the coding of B&B@Grid divides the size of the communicated information by an average of 766. These experiments have also shown that the coding of the communicated information, performed using B&B@Grid, is on average 92 smaller than the coding of the token approach of PICO [EPH00], and 465 smaller than the coding of the variable approach published in [IF00]. These experiments demonstrate the advantage of using B&B@Grid in clusters to reduce the size of the communicated information, and therefore the communication delays.

On the other hand, despite the high number of CPU cores in our clusters, about 1111 on average, the experiments, carried out with the *master-worker* paradigm, showed that the master exploits its processor at 1.29% on average, and that workers spend an average of 99.94% of their time in computing. These two percentages are good indicators of the quality of B&B@Grid's load balancing strategy and its ability to scale up.

The B&B@Grid approach is used to solve an instance of the flowshop problem, known as *Ta056* (50 jobs and 20 machines), and published in [Tai93] which has never been optimally solved before. In terms of the used computing power, the *Ta056* resolution ranks second among the large-scale challenges tackled in combinatorial optimization field before 2007. On average, 328 processors were used for more than 25 days, and a peak of about 1200 CPU cores was recorded during this resolution.

In the recent years, GPU-powered clusters appeared to be as more and more interesting alternatives for the parallelization of some algorithms. GPU accelerators are often used in today's largest high performance computing systems for regular and data parallel applications. However, B&B algorithms are highly irregular in terms of workload, control flow and memory access patterns. The research works, presented in this document, are a continuation of B&B@Grid to adapt B&B algorithms, having an irregular structure, to GPU-powered clusters, having accelerators with a highly regular architecture.

Before presenting our contributions, in the parallelization of B&B algorithms, this chapter describes our main industrial and academic research activities, done after my PhD and which are not presented in this manuscript. This industrial and academic research is presented in Subsection 1 and Subsection 1, respectively.

2.1 Industrial research

Our industrial research is mainly carried out in the railway field. These research works allowed us to have several contributions, such as the development of a train simulator, the development of a train autopilot, reducing the energy consumption of a rail network, or automatic learning of the technical parameters of a train. In general, our work for the railway industry is not intended for scientific publications. One of our main industrial partners, Alstom, prefers to keep confidentiality for promising results.

G-Drive : an automatic train operation

- **Problem description:** The cost of energy, for the traction of passenger trains, is estimated at $\sim$100 million euros during 2006 in Belgium, and at $\sim$850 million euros during 2008 in France. Some studies show that, in the same journey and the same driving conditions, a train can consume up to 50 % more energy than another train. Therefore, the cost of the wasted energy for train traction is relatively high.

- **Conventional approach:** To reduce this cost, railway operators use two types of approaches. The first approach is to give the driver a list of expected times at some track positions during journey. And the second is to equip trains with a driver assistance system.

- **Our approach:** In collaboration with Alstom, we developed an automatic train control system. This system automatically controls and drive train by optimizing energy consumption and respecting the constraints.

- **Obtained results:** Simulator tests show that our system saves from 15% to 25% energy compared to drivers of the National Railway Company of Belgium.

- **Publications:** We decided to do not publish the used optimization method because Alstom plans to market this system in the coming months. An international patent is under submission. In this patent, I am one of the authors of the developed system.

Energy-efficient railway traffic control

- **Problem description:** In modern railway system, most of the energy required by trains is supplied by the electric network. In recent years, the reduction of energy consuming has become one of the main concerns of the railway managers.

- **Conventional approach:** Although railway traffic control is an important topic in the modern railway management system, few results about the studies on railway traffic control, especially real-time energy-efficient traffic control for general railway system, have been published in the literature. Most of the studies simplify the problem by using average velocities or constant speeds to estimate the energy consumption of the trains along their journeys.

- **Our approach:** Our method proposes suitable driving profiles for the trains that run in the same railway network in order to reduce the total energy consumption of the whole railway network. To minimize the total energy consumption of a targeted railway network, it is important to not only propose ecodriving driving profile for each train but also to synchronize the operations of different trains. This synchronization introduces a high reutilization rate of the energy regenerated by dynamic braking operations of the trains.

- **Obtained results:** Experimental data used in to evaluate our approach are supplied by the Belgian railway infrastructure manager.

- **Publications:** This work is published in Mathematical Problems in Engineering journal [TFMJ13]. According to the experimental results, our proposed method can generate energy-efficient traffic control solution, where the driving profile of a train is defined by a suitable mono-train driving strategy that can be different from the driving strategy used by another train part in the same railway network.

Learning the parameters of a train

- **Problem description:** In some algorithms, it is important to modelize the train. The obtained train model is defined by a set of parameters. These parameters are related to: (1) the train traction system; (2) its braking system; (3) its aerodynamic; (4) the different masses of the train; and (5) its length; In total, any realistic train model is defined by more than 100 parameters.

- **Conventional approach:** For each train, it is important to know the values of these parameters. To find these values, the conventional approach is (1) to contact the manufacturer of the train to get the values of some parameters, and (2) to measure the values of the other parameters. However, this approach has two disadvantages: (1) measuring the value of a parameter can be an expensive operation; and (2) some parameters, such as the total mass of a train, are constantly changing;

- **Our approach:** Our learning algorithm deduces the values of the parameters by only observing the behavior of the train. The train is seen as a system which receives input commands. These commands can be traction, braking, or no control. A command changes the state of the train. This state is defined by the position of train, its velocity, and its acceleration. The state of the train can be measured by an odometer, and this state is measured with a certain noise. The role of the learning algorithm is to deduce the parameters of the train using a great number of pair of values. Each pair is defined by (1) an input command and (2) the obtained state.

- **Obtained results:** With our method, we are able to learn the exact values of all parameters when the states of train is known without noise. If the states contain noise, the algorithm is able to learn these values with an average error of less than 2%.

- **Publications:** These results are obtained in January 2017. Next weeks, we will discuss with our partners the possibility of publishing this work.

Developing a train simulator

- **Problem description:** In the optimization algorithm of the railway sector, the cost function is sometimes based on the simulation of a train.

- **Conventional approach:** At the beginning of a project, we received a professional simulator, where the simulation is based on a time discretization. However, we observed that the obtained simulations are not sufficiently accurate, and some values do not match those of the train's mathematical model.

- **Our approach:** So we developed a train simulator, based on a time discretization, and with a better accuracy. This simulator takes into account, not only, the control system, the traction and braking systems, and the aerodynamic of the train, but also, the track topology, namely gradients and curves.

- **Obtained results:** With negligible maximum errors in the positions, velocities and accelerations, the obtained simulations correspond to those of the train's mathematical model.

- **Publications:** This simulator is more a development work than a research one. As the scientific contribution is not important, we do not plan to publish this work.

Multi-objective optimized railway timetable

- **Problem description:** A timetable can be improved according to different criteria, such as robustness, which measures the sensitivity of a timetable to disturbances, and total energy consumption. As the Belgian network is relatively dense, it is important to improve the timetable by taking into account these two criteria.
- **Conventional approach:** To the best of our knowledge, there is no work in the literature which optimize the timetable according to the robustness and the energy consumption, using a realistic energy model. In addition, some works try to optimize the timetable by building a new timetable from scratch. However, the users do not like having lot of changes in the timetable.
- **Our approach:** The objective of our algorithm is not to generate a new timetable but to improve the current one, transparently to the user. Between two successive stations, a timetable defines the moments at which the train must pass at certain intermediate positions of the track (i.e. between two stations). Our algorithm does not modify the arrivals and departures times at the stations, but updates only the times of these intermediate positions.
- **Obtained results:** The evaluation of the energy consumed by a network requires the use of a special simulator. In this project, our role was not to develop this simulator, but only the optimization algorithm. Our optimization software was provided to our partners approximately six months before the end of the implementation of the simulator.
- **Publications:** This work is not published.

Energy-aware railway timetable re-scheduling during disturbances

- **Problem description:** Disturbance management can be seen as a three-step process. The first step is the detection of a disturbance. Then, a decision is made to manage this event. And finally, this decision is implemented.
- **Conventional approach:** In our work, we assume that another system is responsible for detecting disturbances. Decision-making, such as the removal of a train, its deviation, or its slowdown, is a complicated process, which is hard to automate. When a

decision is taken to change the path of a train, the Belgian railway operator, Infrabel, manually generates a local timetable for this path.

- **Our approach:** The software, which we have developed, allows to generate this local timetable automatically by optimizing at the same time the tardiness and the energy consumption.
- **Obtained results:** The evaluation of the energy consumed by a network requires the use of a special simulator. In this project, our role was not to develop this simulator, but only the optimization algorithm. Our optimization software was provided to our partners approximately six months before the end of the implementation of the simulator.
- **Publications:** This work is not published.

2.2 Academic research

Simultaneously with industrial research, we also had contributions in the academic research. This section presents only our main academic works which are not related to B&B algorithms. These contributions belong to three different domains, namely combinatorial optimization, cloud computing and machine learning.

Hyper-heuristic GRASP

- **Problem description:** GRASP is a well-known two-phase metaheuristic. First, a construction phase builds a complete solution iteratively component by component by a greedy randomized algorithm. After that, a local search phase improves this solution. The basic GRASP configuration is defined by (1) a cost function, (2) a probabilistic parameter of greediness, and (3) a neighborhood structure.
- **Conventional approach:** Traditionally, the choice of a GRASP configuration (greediness, cost function, and neighborhood) is guided by the theoretical studies or by manually testing the different parameter values. However, hyper-heuristic framework exists to design and adapt automatically heuristics to solve hard computational search problems. The feature of this framework is a protocol (also called a high-level heuristic or mechanism) that manages with the low-level heuristics (tabu search, genetic algorithms, etc.) looking for the best configuration of their parameters.
- **Our approach:** Therefore, in our work we propose a hyper-heuristic that runs GRASP configurations in accordance with several predefined configurations. Each

configuration manages a set of one-iteration GRASPs with different parameter. The goal is to automatically test multiple GRASPs and collect performance measures in order to detect the leading configurations for given test instances. We consider 7 well-known neighborhood structures, 9 greediness values, and 5 cost functions (including a cost function based on a bounding which is integrated for the first time in GRASP).

- **Obtained results:** Our computational experiments have not revealed one leading configuration, however the winning one is a stochastic mix of several configurations.
- **Publications:** This work is published in CCPE journal [AMTM17] (Concurrency and Computation: Practice and Experience).

Optimization of precedence-constrained parallel application problem

- **Problem description:** Parallel programs can be represented by a directed acyclic graph where each node represents a task and each edge from a node A toward a node B means that the task B starts after the end of the task A. In this graph, The weight of a node A represents the computation cost of the task A, and the weight of an edge (A,B) represents the communication cost from the task A to the task B. These parallel tasks must be deployed on several computing units. Each computing unit can work with different voltage levels. For each voltage level, a computing unit has a processing speed and an energy consumption. Precedence-constrained parallel applications are one of the most typical application models used in scientific and engineering fields.
- **Conventional approach:** In the literature, usually the goal is to deploy this application in order to minimize the total processing time only.
- **Our approach:** Our objective was to minimize not only the total processing time but also the total energy consumption.
- **Obtained results:** Experiments show that our method improves on average the results of the literature for about $10,000$ standard instances. The energy consumption is reduced by about 50% and the processing time by 10%.
- **Publications:** This work is published in JPDC [MMK+11] (Journal of Parallel and Distributed Computing) which is an A*-journal. According to Google Scholar, this paper is the third most cited paper in JPDC during the last 5 years (more than 120 citations).

Solving Tuberculosis Transmission Control (TTC) problem using parallel simulation- and surrogate-assisted optimization

- **Problem description:** Tuberculosis is the most lethal infectious disease with 10 million cases of active tuberculosis recognized in 2017. The World Health Organization (WHO) proposed the End Tuberculosis Targets, with the objective to cut new cases by 90% between 2015 and 2035.

- **Conventional approach:** The AuTuMN simulator [TRDM17] implements a tuberculosis transmission dynamic model and facilitates predictions of future tuberculosis epidemic trajectories across diverse scenarios. Besides, the coupling with an economic model allows to compare and analyze different control policies, which then helps decision makers in their effort to decrease local tuberculosis burden. However, solving TTC problem using AuTuMN simulator, like many simulation-based optimization resolutions, are computationally expensive.

- **Our approach:** In the PhD thesis of Guillaume Briffoteaux, which I co-supervise, we investigate Evolution Controls (ECs) strategies that define the alternation between the simulator and the surrogate within the optimization process. The challenge is to find the best trade-off between the quality (in terms of precision) and the efficiency (in terms of execution time) of the resolution. We consider several ECs from the literature including the mechanism built on the distance-based concept of Hyper-sphere Confident Regions (HCR) and a mechanism derived from Monte Carlo dropout (MCDropout), a technique originally dedicated to quantify uncertainty in deep learning. The investigation of this latter uncertainty-aware EC is uncommon in surrogate-assisted optimization.

- **Obtained results:** We identify some characteristics common to several ECs and then derive from them different variants that are compared considering a pioneering application to the TTC problem. The reported results show that HCR- and MCDropout-based EC coupled with massively parallel computing outperforms the literature strategies.

- **Publications:** A first part of this work is published in the HPCS'2018 conference [BMMT18], the second part is under minor revision for the Future Generation Computer Systems (FGCS) journal (Impact Factor of 5.768), and the third part is submitted to the Swarm and Evolutionary Computation (SWEVO) journal (Impact Factor of 6.330).

CHAPTER 2

Introduction

This manuscript gives an overview of my research activities focused mainly on two PhD theses that I co-supervised, namely those of Rudi Leroy [Ler15a] and Jan Gmys [Gmy17] defended respectively in 2015 and 2017. Other activities conducted within the context of academic and industrial collaborations are summarized in the CV part of this manuscript (Chapter 1).

Many industrial and economic problems, like flowshop scheduling, are permutation combinatorial optimization problems. Solving these problems consists in finding an optimal permutation of elements among a large finite set of possible permutations. A wide range of these problems is known to be large in size and NP-hard to be solved. The Permutation Branch-and-Bound (Permutation B&B or PB&B) algorithms are one of the most used exact methods to solve these permutation optimization problems. These algorithms are based on an implicit enumeration of all the feasible permutations of the problem to be tackled. Building and exploring the PB&B tree are performed using four main operations: branching, bounding, selection and pruning. In PB&B algorithms, if the lower bound for some tree node A is greater than the best permutation found so far, then A may be discarded from the search. This key idea of the PB&B algorithms significantly reduces the number of explored nodes. However, the execution time of a PB&B notably increases with the size of the tackled problem instance, and often only small or moderately sized instances can be practically solved.

For this reason, over the last decades, parallel computing has been revealed as an attractive way to deal with larger instances of combinatorial optimization problems. Recently, multi-core processors (CPUs), Graphics processing units (GPUs) and computing clusters have been used for the parallelization of several algorithms. To the best of our knowledge, all parallel PB&B algorithms developed in the literature are based on using one centralized or distributed pool which stores nodes [GC94]. In these conventional approaches, PB&B threads cooperate by adding nodes to or removing them from this or these pool(s), usually implemented as a linked list (LL) [Cra06]. A parallel PB&B algorithm stops when this or these pool(s) is/are empty. In the two theses, our work has led to the development of four approaches, renamed in this manuscript PB&B@CORE, PB&B@CPU, PB&B@GPU and PB&B@CLUSTER.

2.1 PB&B on single-core processors

A single-core processor is a microprocessor able to run one single thread at any time. Processors remained single-core until it was difficult to improve their computing power by increasing clock speed or the number of transistors. Therefore even processors of smartphones are nowadays no longer single-core processors.

However before developing a parallel approach of an algorithm, it is often important to optimize the serial approach of this algorithm. A serial algorithm can be optimized in order to run it more rapidly or use less memory. If code optimization can be done using new procedures or functions, memory optimization often requires the development of a new data structure.

In our work, we propose an original and pioneering single-core PB&B@CORE algorithm, based on a new data structure, called Integer-Vector-Matrix (IVM), and the factorial number system. This special numbering system [Lai88a], also called factoradic number system, is a mixed radix numeral system adapted to numbering permutations. The objective of our new IVM-based PB&B approach is to accelerate the management of the PB&B pools and to reduce the size of the memory used to store these pools.

Our new PB&B@CORE is compared with a typical LL-based approach [MMT13] in terms of memory usage and CPU time used to manage the PB&B pool. This comparison shows that our PB&B@CORE approach outperforms this typical LL-based PB&B approach.

2.2 PB&B on multi-core processors

As indicated before, for decades it was possible to improve performance of a general-purpose CPU by increasing its operating frequency. However in about 2012, manufacturers encounter two main technical barriers to improve the CPU performance, namely the power barrier (i.e. the power consumption exponentially increases with each factorial increase of operating frequency), and the memory barrier (i.e. the gap between memory and processor speeds is increasing). In order to continue delivering regular performance improvements for general-purpose processors, manufacturers such as Intel and AMD have turned to multi-core designs. Multi-core processors embed two or more independent execution CPU cores into a single processor.

By providing multiple execution cores, each sequence of instructions, or thread, has a hardware execution environment entirely to itself. This enables each thread run in a truly parallel manner. Nowadays, most processors can be considered as parallel machines. When designing an algorithm for a multi-core processor, it is therefore important to take into account a certain number of issues, such as the definition of work unit and work sharing/stealing strategies. A major advantage of multi-core processors is the possibility to parallelize using threads instead of processes. Unlike processes, which have their own virtual memory, the threads of a process share the same virtual memory. Therefore, communications between these threads are faster than between processes. Many programming methods (such as [EPS09a], [PŽ09], and [SECG11]) based on the use of threads have been developed.

Our new method PB&B@CPU is based on a new approach to manage the pools of nodes of a PB&B parallel algorithm using five different work stealing strategies. This approach aims at reducing the CPU time used to manage the thread private PB&B pools, the memory behavior of the algorithm, the number of performed work stealing operations and finally the total execution time. It is based on the use of the IVM data structure to manage the thread private PB&B pools. In our stealing strategies, work units stolen by threads are intervals of factoradics instead of sets of nodes. The IVM-based PB&B@CPU is compared with the LL-based PB&B approach by solving some hard flowshop scheduling problem instances using the five work stealing strategies.

2.3 PB&B on GPU accelerators

Commodity Graphics Processing Units (GPUs) have rapidly evolved to become high performance accelerators for data-parallel computing. Recent GPUs, like Tesla V100, contain more than 5000 of processing units, capable of achieving up to 14 TFLOPS for single-precision arithmetic, and 7 GFLOPS for double-precision calculations. In addition, modern high-performance computing optimized GPUs contain up to 32GB of on-board memory, and are capable of sustaining memory bandwidths up to 900GB/sec. GPU-based accelerators are also becoming popular because of their extraordinary energy efficiency, as illustrated by The Green500 List[1]. The maximum power consumption of Tesla V100 is 300W. However, the parallelization of an application on a GPU can not be done directly. It is important to take into account a certain number of issues, such as the optimization of data transfer between CPU and GPU, the reduction of thread divergence, an efficient mapping of data on the GPU cores, and the best location of data on the different memories of the GPU. Because of all these issues, the usage of GPU is often restricted to regular and data parallel applications.

The irregular nature, in terms of workload, control flow and memory access patterns, of applications such as PB&B may seriously degrade the performance of the GPU. The acceleration of PB&B algorithms using GPUs is therefore a challenging task which is

[1]http://www.green500.org

addressed by only a few works in the literature, such as [CM13], using flowshop as a test case, [CMNLdC11], applied to the traveling salesman problem and [LEB12], applied to the knapsack problem where the search tree is binary. All these approaches use linked lists (deques, stacks, etc.) to store and manage the pool of nodes, likewise most parallel PB&B algorithms in the literature. Such data structures are very difficult to handle on the GPU and often induce prohibitive performance penalties. For this reason all GPU accelerated PB&B algorithms at our knowledge perform the management of the pool of nodes at least partially on the CPU, requiring costly data transfers between host and device. In [MCB14] it is shown that the bounding operation for flowshop consumes $97 - 99\%$ of the execution time of a sequential PB&B and that the GPU based parallelization of this operation can provide a substantial acceleration of the algorithm. However, as the management of a list of pending nodes is performed on the CPU, the transfer of data between CPU and GPU constitutes a bottleneck for GPU accelerated B&B algorithms.

This manuscript describes our new PB&B@GPU approach, which is based on the parallelization of PB&B on GPU. This parallel PB&B@GPU algorithm is, to the best of our knowledge, the first one that implements all PB&B operations on the GPU, requiring virtually no interaction with the CPU during the exploration process. It is based on the IVM data structure, which allows the efficient storage and management of the pool of nodes in permutation based combinatorial optimization problems. The IVM structure provides some regularization as it allows to store and manage the pool of nodes with data structures of constant size. However, the IVM-based parallel PB&B is still highly irregular in terms of workload, control flow and memory access patterns. None of these three issues can be ignored when implementing the PB&B algorithm on the GPU and all three are addressed in PB&B@GPU. The focus is put on the reduction of thread divergence which arises in CUDA's SIMD execution model as a consequence of control flow irregularities.

2.4 PB&B on GPU-powered clusters

Large-scale GPU clusters are gaining popularity in high performance computing community. In November 2019, nearly 40% of the total compute power on the TOP500 clusters[2] (i.e. 626 petaflops) comes from GPU-accelerated systems. Just over a decade ago, no supercomputers and clusters on the list were accelerated.

By gathering GPU processing power distributed across multiple computing nodes, it is possible to run advanced, large-scale applications efficiently, reliably, and quickly. This acceleration delivers a dramatic boost in throughput and cost savings, paving the way

[2]https://www.top500.org

to exascale science applications. The massively parallel hardware architecture and high performance of floating point arithmetic and memory operations on GPUs make them particularly well-suited to many of the same scientific and engineering workloads that occupy HPC clusters, leading to their incorporation as HPC accelerators.

To distribute an algorithm on the computing nodes of a GPU-powered cluster, it is necessary to rethink the conventional parallel mechanisms in order to adapt them to the characteristics of this environment. Taking into account of such characteristics can be done by the resolution of the issues related to communication delays (i.e. optimizing the communication cost) and processor's heterogeneity (i.e. optimizing the load-balancing strategy).

In our work, we have developed a new method, called PB&B@GPU, based on the coordinator-worker B&B@Grid approach [MMT07c]. In order to enable the use of a hybrid CPU-GPU cluster in PB&B@GPU, a redefinition of work units is proposed using interval lists. A detailed description of the coordinator and worker new operations is provided, focusing on the revisited communication scheme. Finally, PB&B@GPU is experimented on three different GPU-enhanced clusters with up to 36 GPUs. The experimental evaluation includes scalability and stability analysis on solving very large flowshop instances.

2.5 Document contents

In addition to Chapter 3, which gives a state of the art of B&B algorithms, this manuscript is organized in four main chapters. Chapter 4 proposes an original and pioneer serial PB&B@CORE algorithm, based on a new data structure. Chapter 5 explains our PB&B@CPU, designed to work on multi-core CPUs. Chapter 6 describes the new PB&B@GPU approach, which is based on the parallelization of PB&B on GPUs. And finally, Chapter 7 presents our new PB&B@GPU approach for GPU-powered clusters.

CHAPTER 3

Background and related works

Contents

3.1 Introduction

The exact resolution methods used in combinatorial optimization are often B&X algorithms. These methods are mainly available in three variants: Branch-and-Bound (B&B), Branch-and-Cut (B&C) and Branch-and-Price (B&P). There are other less known B&B variants like Branch-and-Peg [GGS04], Branch-and-Win [PC04] and Branch-and-Cut-and-Solve [CZ06]. This list is certainly not exhaustive. It is also possible to consider a simple tree-based algorithm like Divide-and-Conquer as a base for the B&B algorithm. It is enough to remove the pruning operation of the B&B, explained later, to obtain the Divide-and-Conquer algorithm. Some authors consider the B&C, B&P algorithms and other variants as separate B&B algorithms. In the remainder of the manuscript, the B&B algorithm designates the simple B&B itself or any other variant of this algorithm. In addition, permutation

B&B (PB&B) designates any B&B algorithm or its variants when it solves a permutation problem.

Many problems, solved by B&B algorithms, are permutation problems, where the goal is to find the optimal scheduling in a set of elements. To the best of our knowledge, these permutation problems are solved using B&B without taking into account the permutation aspect of these problems. With taking into account this aspect, it is possible to significantly improve B&B algorithms. Therefore, the remainder of this manuscript focuses primarily on PB&B. In other words, our manuscript focuses on solving permutation problems using B&B algorithms.

In addition to this introduction and a conclusion, this chapter is divided into three sections. In Section 3.2 and Section 3.3, a background is given on respectively permutation problems and the conventional PB&B algorithm based on linked-list. Finally, Section 3.4 gives a state of the art of B&B algorithms.

3.2 Permutation problems

This section presents a brief background on permutations focusing on the flowshop problem considered as a test case to validate our approaches.

3.2.1 Permutations

Let's assume a permutation problem where the objective is to find the best permutation in a set of N elements. These elements can be jobs, cities, locations, and so on. It is always possible to assign a number to each of these N elements. The first element can be designated by 1, the second element by 2, ..., and the last by N. Therefore, any permutation problem of N elements can be represented as a permutation problem of the first N positive natural numbers.

A permutation is obtained after the assignment of numbers to positions. Before making this assignment, all numbers are free and the positions are empty. At the end of the assignment, all numbers are assigned and positions are occupied. During the assignment, some numbers are free, others assigned, some positions are occupied and others empty. In the rest of the manuscript, **a number can be said free or assigned, and a position can be said occupied or free**.

After the assignment of all numbers to all positions, the permutation is called complete. For example, $(2, 3, 1, 4)$ is a complete permutation where the numbers 2, 3, 1 and 4 are respectively assigned to positions 1,2, 3 and 4. Before starting the assignment, the permutation is said to be empty. For example, $\{1, 2, 3, 4\}$ is an empty permutation where all numbers are free and all positions are empty. During an assignment, the permutation is said to be partial. For example, $(2, 3, \{1, 4\})$ is a partial permutation, where numbers 2 and 3 are assigned, numbers 1 and 4 are free, positions 1 and 2 are occupied, and positions 3 and 4 are free. A partial permutation can also have the form $(2, \{1, 4\}, 3)$. In this example, positions 1 and 4 are occupied and the other positions are empty.

Among the empty positions, it is possible to specify **the first, the second, ... and the last empty position**. For example, in $(2, \{1, 4\}, 3)$, the first empty position is 2 and the last empty position is 3. Among the assigned numbers, it is possible to specify also **the first, the second, ... and the last number assigned**. For example, in $(2, \{3, 1, 4\})$, the last number assigned is 2. On the other hand, looking at the $(2, \{1, 4\}, 3)$, it is not possible to find the first and the last number assigned. This manuscript defines **the size of a permutation** as the number of free numbers or empty positions. For example, the permutations $(2, \{1, 4\}, 3)$, $\{1, 2, 3, 4\}$ and $(2, 3, 1, 4)$ respectively have as sizes 2, 4 and 0.

Any permutation, whether empty, partial or complete, contains a certain set of complete permutations. For example, the partial permutation $(2, \{1, 4\}, 3)$ contains the two permutations $(2, 1, 4, 3)$ and $(2, 4, 1, 3)$. The empty permutation $\{1, 2, 3, 4\}$ contains all $4!$ possible permutations. In the same way, the complete permutation $(2, 3, 1, 4)$ contains only one complete permutation. **The space of a permutation** P, denoted $Space(P)$, is defined as the set of the complete permutations contained in P.

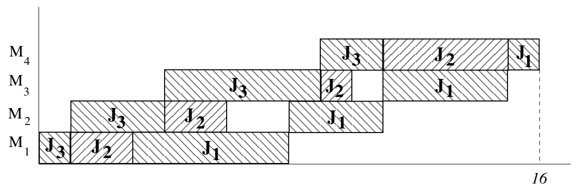
3.2.2 Flowshop problem

The four approaches presented in this document are validated using the flowshop problem. In manufacturing environments, it is common to find permutation flowshop scheduling problems [BG76, KS80, AGA99] where n jobs have to be processed on m machines where the goal is to optimize an objective function. The objective of the flowshop is to schedule a set of n jobs on a set of m machines where each job J_1, J_2, ..., J_n is processed on the machines M_1, M_2, ..., M_m organized in the line. Each job J_i with $i = 1, 2, \ldots, n$ is made of a sequence of m operations O_{i1}, O_{i2}, ...,O_{im} where operation O_{ik} is the processing of job J_i on machine M_k for a processing time p_{ik} that can not be interrupted. The objective of the flowshop is to find a processing order on each machine M_k which minimizes the time necessary to complete all jobs, also known as the *makespan*. In this manuscript, each reference to the flowshop is actually a reference to the permutation flowshop [AGA99, HS05]. Using Johnson's algorithm [Joh54], it is possible to find an optimal schedule for the flowshop in $O(n \log n)$ steps when $m = 2$. The problem is NP-hard when $m \geq 3$ [GJS76]. This is why it is often tackled using metaheuristics [Bas05] to deal with large problem instances. Figure 3.1 shows an example of a flowshop instance where $n = 3$ and

$m = 4$, it also shows the optimal complete permutation.

	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 3.1: Illustration of a permutation flowshop where $n = 3$ and $m = 4$. The table shows the processing times of the jobs on each machine. The Gantt diagram shows the optimal complete permutation for this particular instance.

These are the constraints that a valid flowshop complete permutation should satisfy:

- A machine can not start processing a job before the preceding machines have finished the processing of that job. In other words, machine M_j can not process operation O_{ij} before it is completed on machine M_{j-1}.
- An operation can not be interrupted, and since a machine processes one job at a time, the machines are critical resources.
- The sequence of jobs must be the same on all machines, e.g. if job J_3 is processed in second position on the first machine, job J_3 must also be processed in second position on all the other machines.

The lower bound proposed by Lageweg *et al.* [LLK78] is used in our bounding operation. This bound is known for its good results and has complexity of $O(m^2 nlog(n))$, where n is the number of jobs and m the number of machines. This lower bound is mainly based on Johnson's theorem [Joh54] which provides a procedure for finding an optimal complete permutation for a flowshop scheduling problem with 2 machines.

3.3 Linked-list based PB&B

This section describes the conventional PB&B algorithm based on a linked-list. As indicated by its name, the PB&B is based on two main operations, namely *branching* and *bounding*.

In addition to these two operations, the PB&B algorithm is defined by other operations, as shown in Figure 3.2.

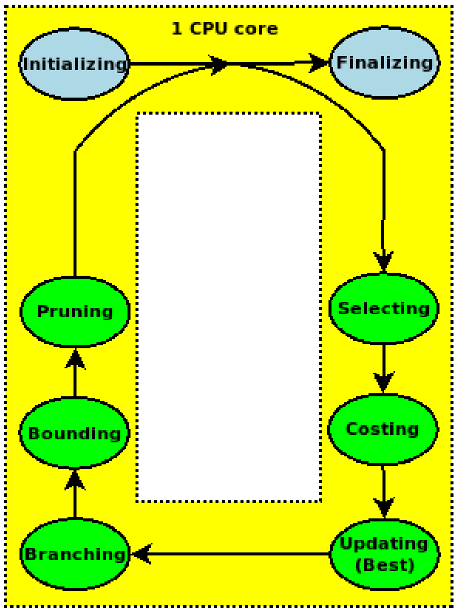


Figure 3.2: The conventional PB&B algorithm and its operations.

Two of the PB&B operations are called *initializing* and *finalizing* operations. The main role of the initializing operation is to allocate memory for the data structures of the algorithm, to initialize them, and to add the empty permutation to the PB&B pool. The finalizing operation is invoked when this PB&B pool is empty. Its role is to provide the decision maker by the best found permutation, and to release the allocated data structures.

Between these two operations, the PB&B algorithm runs a large number of iterations. At each iteration, the operations of selection, costing, updating, branching, bounding, and pruning always intervene in this order. The selection operation takes a permutation from the pool. Two cases may arise: in the first case, the selected permutation is complete, and in the second case, the permutation is empty or partial.

If the first case occurs, then the costing operation calculates the cost of the permutation. If the calculated cost improves the cost of the best known complete permutation, then the updating operation saves this complete permutation as the new best known complete permutation. In the second case, costing and updating operations are not used. On the other hand, the operations of branching, bounding and pruning intervene. The role of the branching operation is to divide the selected permutation into several permutations. The bounding operation calculates the bound of each of the obtained permutations, and the pruning operation removes a certain number of permutations and puts into the pool the un-deleted permutations. Each of the following subsections presents in more details one of the main operations of the conventional PB&B.

3.3.1 Branching

The branching operation works according to the political and sociological strategy *Divide and rule* (from Latin dīvide et imperā). The idea here is to divide a partial permutation P into a set of E sub-permutations. Of course, there are different ways to divide the same permutations P and, for each division, a different set of E of sub-permutations can be obtained. For example, the permutation $(1,\{2,3,4\})$ can be divided into two different ways:

- $E_1 = \Big\{(1,2,\{3,4\}),\ (1,3,\{2,4\}),\ (1,4,\{2,3\})\Big\}$.
- $E_2 = \Big\{(1,\{3,4\},2),\ (1,\{2,4\},3),\ (1,\{2,3\},4)\Big\}$.

The two divisions are made by filling respectively the first and last empty positions, using one of the three free numbers. In our work, we use only the first free and last positions. However, from a theoretical point of view, it is also possible to use the other empty positions, such as the second empty position, the third position, etc. The branching operation can use any division of a permutation P in a permutation set $P_1, P_2, ..., P_n$ that meets both of the following conditions:

- $\forall(P_i, P_j),\ Space(P_i) \cap Space(P_j) = \emptyset$. This first condition indicates that the sub-permutations of P must be disjoint.
- $Space(P) = \cup_{i=1}^{i=n} Space(P_i)$. This second condition is that the permutation space P equals the union of all sub-permutation spaces of P.

The same permutation P can not be split simultaneously with different techniques. For example, P can not be split using both first and last free positions simultaneously. However, it is possible to use a certain technique of division of the permutation P, and another division technique to divide one of the sub-permutations of P. Therefore, it is possible to use a single division technique, but it is also possible to combine several techniques in the same PB&B algorithm. The example, Figure 3.3 shows the result of a complete division from an empty permutation of size 4. In this example, the division technique used is the one that is based on the first empty position.

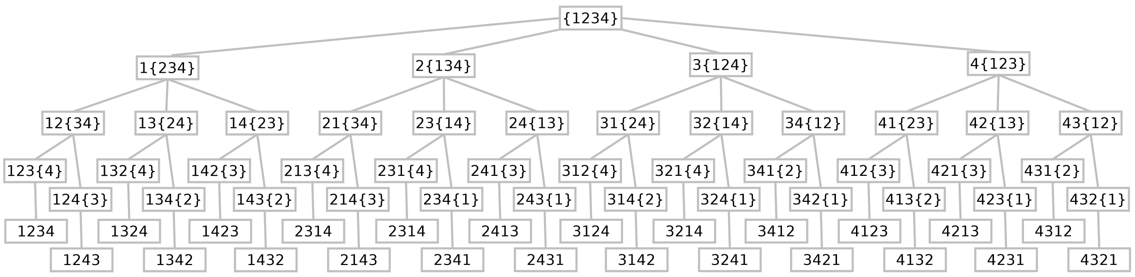


Figure 3.3: Example of a complete branching tree using the first empty position division technique.

A PB&B always starts with the division of the empty permutation of size N. The division of this empty permutation produces N partial sub-permutations of size $(N-1)$. The division of each of these partial sub-permutations produces $(N-2)$ sub-sub-permutations of size $(N-2)$. This process continues until the obtained permutations are complete and have size 0. These successive divisions make it possible to define a tree, called branching tree, where the empty permutation is the root of the tree, the complete permutations are the leaves of the tree, and partial permutations constitute the internal nodes of the tree.

The branching tree obtained is constituted of 1 empty permutation, N partial permutations of size N, $N(N-1)$ partial permutations of size $(N-1)$, $N(N-1)(N-2)$ partial permutations of size $(N-2)$, ..., and finally, $N!$ complete permutations. As the size of this tree is exponential, it is impossible to explore, in a reasonable time, all permutations when N exceeds a certain threshold value.

3.3.2 Bounding

To avoid exploring the entire branching tree, the PB&B algorithm uses another operation called the bounding. This operation receives a permutation, as input, and returns a cost, as output. If it is a minimization problem (i.e searching a permutation of minimal cost), then the bounding operation returns a lower bound. On the other hand, if it is a maximization problem, then this operation returns an upper bound. In the remainder of the manuscript, a problem of minimization is considered, but the proposed approaches are also valid for maximization problems.

As explained previously, the space of a partial permutation P, denoted $Space(P)$, is equal to the set of complete permutations contained in P. For example, the permutation space of $(1,\{2,3\},4)$ includes the permutations $(1,2,3,4)$ and $(1,3,2,4)$. If a bounding operation receives as input the partial permutation $(1,\{2,3\},4)$, then this operation should return a value $Bounding(P)$ such that the costs of $(1,2,3,4)$ and $(1,3,2,4)$ must be greater than $Bounding(P)$. In the general case, the following condition must always be satisfied.

- $\forall P' \in Space(P),\ Bounding(P) \leq Costing(P')$

For a permutation P, the higher the value of $Bounding(P)$ is, the better the bounding operation which is used. Let P' be the complete permutation with the minimal possible cost in $Space(P)$. The best bounding operation is the one that returns a value of $Bounding(P)$ equals $Costing(P')$. In addition to the values of $Bounding(P)$, bounding operations can also be compared according to their computation times. The faster a bounding operation is, the better is this operation. If a permutation P is complete, this manuscript assumes that $Bounding(P)$ is always equal to $Costing(P)$.

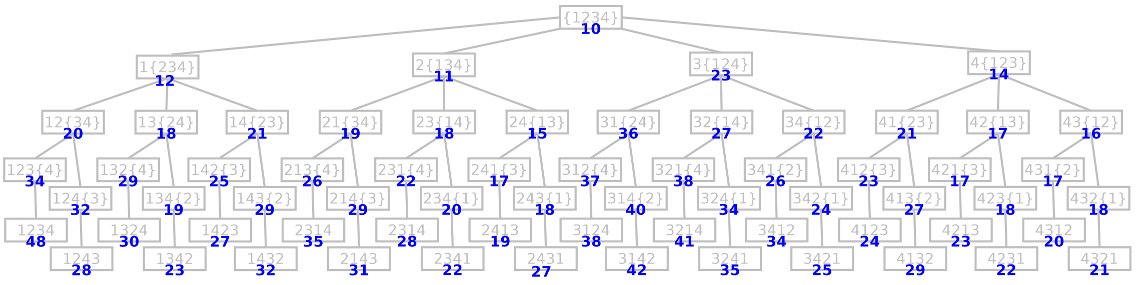


Figure 3.4: Example of a bounding tree with bound values of each node.

3.3.3 Pruning

Subsection 3.3.1 shows that the branching tree is very large. To find the permutation with minimum cost, it is impossible to explore the entire tree for large problem instances. By knowing the lower bound of each partial permutation, it is possible to avoid exploring a large part of this tree. Let's assume that the algorithm knows a complete permutation P, which has a cost equal to $Costing(P)$, and that this algorithm encounters a partial permutation P', which has a lower bound equal to $Bounding(P')$. In other words, all complete permutations in $Space(P')$ have a cost greater than $Bounding(P')$. If $Costing(P)$ is less than $Bounding(P')$, then there is no complete permutation of $Space(P')$ that can have a better cost than P. Therefore, it is unnecessary to further divide the partial permutation P'. This permutation P' can be ignored and pruned from the branching tree. The pruning operation receives a cost $Costing(P)$ of complete permutation P and the lower bound $Bounding(P')$ of a partial permutation P' and returns a boolean value indicating whether P' should be branched or pruned. Figure 3.5 gives an example of the bounded part of a branching tree. The bounded part of a tree is composed by all the nodes of the tree where the bounds are computed.

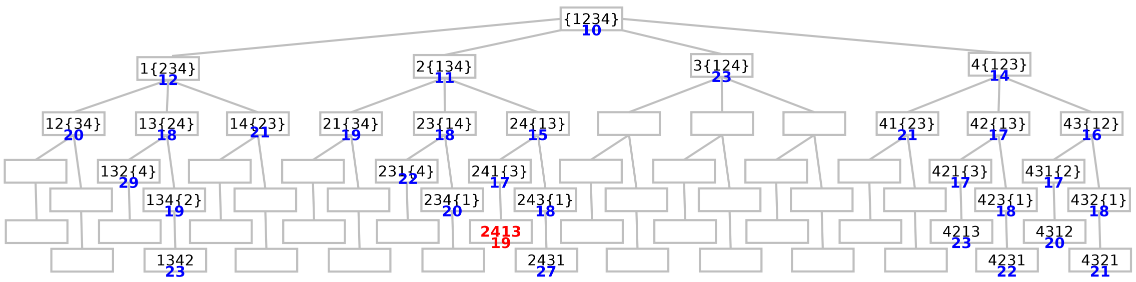


Figure 3.5: An example of the bounded part of a branching tree

3.3.4 Selection

The role of the selection operation is to determine the order in which the permutations of the tree are explored. In other words, its role is always to choose the next permutation, of the pool, coded using a linked-list, to branch. In a PB&B, there are three main strategies for selecting the next permutation of the pool to be branched.

- **Best first:** The permutation of the pool having the smallest bound is chosen first. The advantage of this strategy is to increase the probability of finding more quickly a complete permutation of minimum cost. But its disadvantage is that the number of permutations of the pool increases rapidly.
- **Largest first:** This strategy selects the permutation of the pool with the largest size. In other words, it first selects the permutation close to the root of the branching tree. This strategy makes it possible to considerably increase the number of permutations of the pool. In parallel computing, the advantage of this strategy is to generate a pool large enough to occupy lot of computing units.
- **Depth first:** This strategy selects the permutation with the smallest size. In other words, it first selects the furthest permutation from the root of the branching tree.

In our four PB&B approaches, the strategy used is a combination of depth and best first strategies. As already explained, a permutation is defined by its size, its bound and its last assigned number. These concepts of the last assigned number and the size of a permutation are explained in Subsection 3.2.1. Using these three values, the selection operation in our PB&B approaches combines depth and best first strategies. More exactly, the Selection operation (1) receives a permutation pool as input, (2) makes sure this pool is not empty, (3) determines the largest size S of all the permutations of the pool, (4) selects the set E of the permutations of the pool having this size S, (5) determines the best bound B of all the permutations of E, (6) selects the subset E' of all the permutations of E having the bound B, (7) selects the permutation P having the lowest last number assigned, (8) removes this permutation P from the pool, and (9) returns the permutation P as the output of this Selection operation.

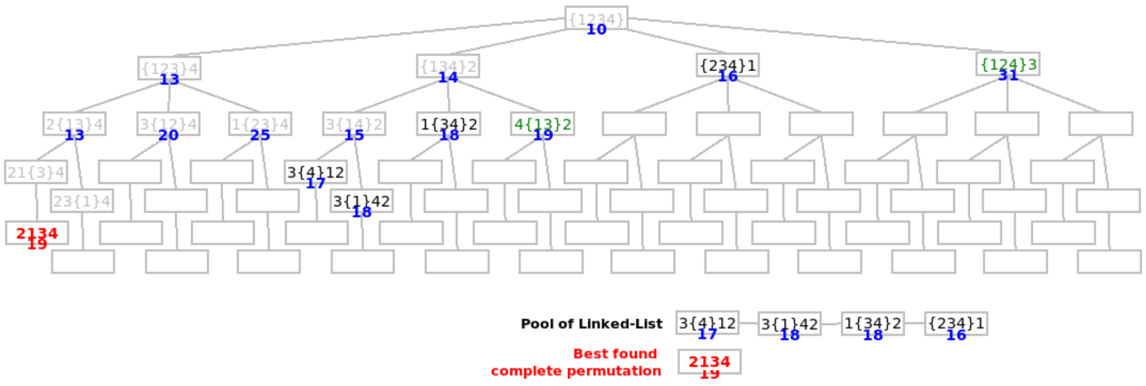


Figure 3.6: Example of the best found complete permutation and the content of the pool, coded using a linked-list, when using the depth best first strategy.

For example, Figure 3.6 shows the PB&B tree during its construction and exploration. All the empty nodes of the tree are not explored, the gray permutations are already branched, the green permutations are pruned because of their bounds, the red complete permutation is the one having the minimal cost so far, and black permutations are neither pruned nor branched. In this tree, it is useless to keep in the memory the gray and green permutations. The only permutations that must be kept are those in black and red. Non-branched and non-pruned permutations are generally kept in a data structure called a linked-list, and the best complete permutation found so far can be kept in an array.

3.4 Related works

The related works of this section is already published in the thesis of Jan Gmys [Gmy17]. The design of parallel B&B algorithms is strongly influenced by the target architecture and the characteristics of the problem being solved [BHP05]. Therefore, and in spite of the simple and generic formulation of B&B, a large number of parallel algorithms have been proposed for different problems and architectures. [GC94] provides a complete, but over twenty year old survey of parallel B&B.

3.4.1 Parallel CPU B&B

Because of the simple basic formulation of B&B it is interesting to have a framework that allows users to easily customize B&B to solve their problems. Many software frameworks have been proposed, including Bobpp [Men17, Bob], PEBBL [EHP15] and PICO [EPH01], parts of the ACRO project [ACR], ALPS/BiCePS [RLS04], BCP and SYMPHONY, which are parts of the COIN-OR project [CO].

This list includes only those frameworks which appear to be maintained at the time of writing. B&B frameworks establish an interface between the user and the parallel machine by defining abstract types for search tree nodes and solutions. As a user, one provides concrete implementations of these types as well as branching and bounding procedures, while the framework handles more generic parts of parallel B&B. The mentioned frameworks differ by the variant(s) of B&B they provide, the type of parallel model they propose and the parallel programming environment. They are implemented with these frameworks are usually designed as multi-layered class libraries, integrating additional features by building on top of existing layers. For example, BiCePS is built on top of ALPS to provide data handling capabilities required for implementing relaxation-based B&B, and PEBBL began its existence as the "core" layer of the parallel mixed integer programming (MIP) solver PICO.

The older versions of these frameworks are often based on master-worker approaches. In order to avoid that the master processor becomes a bottleneck, hierarchical organizations revealed more efficient than pure master-worker implementations [EHP15, Men17, BMT12b]. In these approaches groups of workers form clusters, cooperating locally and interacting with the master through a form of middle management. The idea is to improve locality and relieve the master process by introducing hubs, each handling several workers (master-hub-worker approach). In general, the role of hubs consists in providing work units to a set of workers and coordinating the search process locally, while limiting interaction with the master and worker processes. For the PEBBL framework near-linear speedups on over $6\,000$ CPU cores are obtained for large B&B trees and node evaluation costs of about 10 seconds [EHP15].

Recently, [HSH$^+$17] compared three implementations of a global optimization (GO) B&B algorithm using different levels of abstraction: the Bobpp framework, Intel Thread Building Blocks and a custom P-thread implementation. While they find the Bobpp implementation easiest to code, the authors show that the two other solutions offer better scalability for the used test case. For the optimized test functions, the authors report node processing rates of about 1 million nodes per second ($\mathrm{Mn/s}$) for the sequential version of their custom implementation on a 2 GHz Sandy Bridge CPU.

[EPS09b] presents a software platform called BNB-Solver, allowing the use of serial, shared memory and distributed memory B&B. The proposed approach uses a global work pool and local work pools for each thread. Each thread stores generated nodes in its local pool during N B&B iterations. After N iterations a part of the local nodes are transferred to the global pool. When the local pool of a thread is empty, the thread attempts to take nodes from the global pool and blocks if the global pool is empty. The algorithm terminates when the global pool is empty and all threads are blocked. The authors compare the performance of BNB-Solver with the ALPS and PEBBL frameworks and obtain results similar to [HSH+17], in the sense that, for a knapsack problem (with a reported sequential node processing rate in the order of $1\,\mathrm{Mn/s}$) BNB-Solver outperforms both frameworks.

[CMGH08] proposes two schemes for parallelizing B&B algorithms for global optimization on shared memory multi-core systems, Global and Local PAMIGO (Parallel advanced multidimensional interval analysis global optimization). Both algorithms are parallelized using POSIX threads. In Global PAMIGO, threads share a global work pool and therefore a synchronization mechanism is used for mutually exclusive accesses to the pool. For Local PAMIGO, where thread has its own pool of nodes, a dynamic load balancing mechanism is implemented. A thread stops when its local pool of nodes is empty. When the number of running threads is less than the number of available cores, and a thread has more than one node in its local pool it creates a new thread and transfers a portion of its pool to the new thread. Local PAMIGO ends when there exists no more running threads, and Global PAMIGO ends when the global pool is empty. The authors report profiling results for PAMIGO which show that memory management represents a large percentage of the computational burden. As a very large number of nodes are created in a relatively short amount of time, the kernel needs to satisfy memory allocation and deallocation requests from all threads, creating memory contention. The vast majority of parallel B&B algorithms in the literature store nodes in one or several pool(s) implemented as linked-lists (e. g. priority queues, stacks, deques).

3.4.2 Parallel GPU B&B

The study of [JAO+11] provides a good overview of the challenges faced when implementing parallel backtracking on GPUs. Most of their conclusions from the investigation of GPU-based backtracking paradigm remain valid for B&B algorithm using a depth first search strategy. A fine-grained parallelization of the search space exploration and/or the node evaluation is necessary in order to make use of the GPU's massive parallel processing capabilities. This strongly depends on the nature of the problem being solved and on the choice of the parallelization model. Other critical factors include latency hiding through coalescence, saturation, and shared memory utilization [JAO+11]. Generally speaking, the algorithmic properties of B&B, irregularity of the search space, irregular control flow and memory access patterns are at odds with the GPU programming model. Also, memory requirements for backtracking and B&B algorithms are often difficult to estimate and may exceed the amount of memory available on GPUs. Several approaches for GPU-accelerated B&B algorithms have been proposed. These approaches correspond to different parallelization models and their design is often motivated by the nature of the problem being solved. According to the characteristics of the bounding function one may distinguish among approaches for fine-, medium- and coarse-grained problems.

The GPU B&B and backtracking algorithms for fine-grained problems proposed in [CMNLdC11, CNNdC12, FRvLP10, LLW+15, RS10, ZSW11] perform massively parallel searches on the GPU, based on the parallel tree exploration model. The evaluation of a node for the n-Queens problem in [FRvLP10, LLW+15, ZSW11] requires only a few registers of memory and only a couple of bit operations. The lower bound for the Asymmetric Traveling Salesman Problem (ATSP) used in [CMNLdC11, CNNdC12] is incrementally obtained by adding the cost of the last visited edge to the current cost and therefore has a complexity of $\mathcal{O}(1)$. It requires an access to the distance matrix which can be stored in constant or texture memory. The size of the problems being solved is < 20 for both the ATSP and the n-Queens problems. These algorithms for fine-grained problems share a common approach: the search is split in two parts, an initial CPU search and a final GPU search. The upper tree of depth d_{cutoff} is processed in sequential or weakly parallel manner on CPU, generating a set of active nodes at depth d_{cutoff}. This active set is sent to the GPU, where the lower part of the tree is processed in parallel. Each node of the active set is used as root node for an independent search, which is mapped either to a thread or a warp. This approach requires very careful tuning of the cutoff depth, which strongly influences granularity and load balancing.

Because of varying thread granularities, one of the major issues faced by such approaches is load imbalance. In all of these works the GPU search is performed without dynamic load balancing. However, as noted by [RS10], if "a job is divided into sufficiently many parts, an idle processor will be instantly fed with waiting jobs" and the "GPU's Thread Execution Manager performs that task automatically". This approach assumes two things: first, the initial CPU search is able to generate a large amount of nodes in a reasonable amount of time, and second, the work distribution among independent B&B searches is not *too* irregular.

For many combinatorial optimization problems the cost of the bounding operation is very high, compared to the rest of the algorithm. For instance, the most used lower bounding function for the flowshop consumes $97-99\%$ of the sequential execution time [MCB14]. However, the cost of evaluating one node is sufficiently small to be efficiently performed by a single GPU thread. We therefore refer to this type of problem as medium-grained. For these problems, existing GPU-accelerated B&B algorithms in the literature use the GPU to evaluate large pools of nodes in parallel [CMMB13, VDM13, LEB12]. They use conventional stacks or queues to store and manage the B&B tree on the host, offloading the parallel evaluation of bounds to the device. Indeed, for these problems substantial speedups can be achieved despite sequentially performing pool management on the host. Substantial efforts have been made to port larger portions of the algorithm to the GPU and to reduce overheads incurred by data transfers between CPU and GPU. For instance, branching nodes on the device allows to copy only parent nodes to the GPU. Similarly, pruning evaluated nodes on the device reduces the sequential portion and requires only the transfer of non-pruned children nodes back to the host. Further performance improvements can be obtained by overlapping data transfers with GPU computations, as for example in [VDM13]. For fine-grained problems this approach is likely to perform poorly.

For coarse-grained problems the best way to use the GPU may be as an accelerator for the bounding function itself. In [ABEB+16] a GPU-accelerated B&B algorithm for the jobshop scheduling problem is proposed. The approach also offloads nodes to the GPU but uses a block-based parallelization for each node evaluation. The number of nodes that need to be offloaded in order to saturate the GPU is therefore smaller than for medium-grained problems. A GPU-accelerated algorithm for problems with linear programming bounds is proposed in [MCA13]. Using a GPU-based LP-solver to accelerate this type of problems is very challenging. However, the authors report that for large problems above a certain density threshold their hybrid GPU-accelerated solver outperforms the sequential CLP solver of the open-source COIN-OR library.

3.4.3 Cluster B&B

There are very few works on the parallelization of B&B using multiple GPUs and CPUs in distributed heterogeneous systems. In [VDM13] a linked-list based fully distributed hybrid B&B algorithm combining multiple GPUs and CPUs is proposed. As a test case 20 jobs-on-20 machines flowshop instances are considered in their experiments using a platform composed of 20 GPUs and 128 CPUs. For load balancing a random work stealing mechanism is used. The authors propose an adaptive granularity policy to adapt the amount of stolen nodes at runtime to the normalized computing power of thief and victim. The algorithm is based on a 2-level parallel model, using GPUs for parallel evaluation of lower bounds. In order to reduce CPU-GPU communication overhead, an asynchronous implementation with overlapping host and device computations is proposed. Experimentally, near-linear mixed scalability is shown up to 20 GPUs and 128 CPUs. In [CMMT13] the combined usage of multi-core and GPU processing is investigated. An experimental comparison of concurrent and cooperative approaches shows that the cooperative approach improves the performance with respect to a GPU-only approach while the concurrent approach is not beneficial. Among other issues, the authors identify the reduction of CPU-GPU communication overhead as a major challenge and propose overlapping communication schemes and auto tuning of the offloaded pool sizes to answer this challenge.

Some of the largest known exact resolutions of combinatorial optimization problems have been performed using the master-worker paradigm in combination with grid computing technologies (e. g. *nug30* [ABGL02]). The B&B@Grid platform [MMT07c] uses an interval encoding for work units which significantly reduces the size of messages communicated in distributed B&B. Designed for volatile computing environments, B&B@Grid is fault tolerant thanks to its checkpointing mechanism.

In [BMT12a] an adaptive multi layer hierarchical master-worker approach is applied to the B&B algorithm, using flowshop as a test case. The proposed approach evolves as new resources join the computation, and integrates three types of processes, a super master, masters and workers. Results obtained at the scale of up to 2 000 CPUs show that the multi-layered hierarchical approach clearly outperforms single-layered and classical master-worker approach in terms of efficiency for instances smaller than *Ta056*, as it minimizes bottlenecks at the level of the master and reduces idle time of the workers. In [BMT14] the authors extend their approach, proposing a fault tolerance mechanism.

3.5 Conclusions

This chapter presents the permutation problems, the PB&B algorithms and a state of the art of the B&B algorithm. To the best of our knowledge, permutation problems are solved with the B&B without permutation awareness. The objective of this manuscript is to present new approaches to solve permutation problems, using a B&B that takes into account the permutation aspect of these problems. The next chapters of the manuscript present four new approaches, noted PB&B@CORE, PB&B@CPU, PB&B@GPU and PB&B@CLUSTER. These approaches are developed for four hardware architectures, namely a multi-core computing node, a CPU-accelerator of cores, a GPU and a cluster of computing nodes.

- PB&B@CORE is a serial method, unlike the other three approaches which are parallel methods.
- PB&B@CPU is a method dedicated to a multi-core CPU deployment. Since several years, all commercialized CPUs contain several computing cores, and each CPU can be seen as a parallel machine.
- PB&B@GPU uses a data parallelism, unlike the other three methods based only on the parallelism of instructions. In a GPU, the same instruction can run on several data

simultaneously, according the Single Instruction Multiple Data (SIMD) computing model.

- PB&B@CLUSTER is an approach dedicated to an architecture with distributed memory, while the other three approaches are all dedicated to an architecture with shared memory.

These PB&B@CORE, PB&B@CPU, PB&B@GPU and PB&B@CLUSTER approaches are presented respectively in the chapters 4, 5, 6 and 7.

CHAPTER 4

Single-core IVM-based permutation B&B

Contents

4.1 Introduction

Conventional B&B algorithm is designed to work on all optimization problems. Conversely, the new approach PB&B@CORE is especially dedicated to the resolution of permutation problems. In other words, the B&B algorithm is generic to all optimization problems, while our new PB&B@CORE approach is specific to permutation problems. Compared to the generic B&B algorithm, the PB&B@CORE approach manages a partial permutation pool more efficiently. This PB&B@CORE efficiency, in terms of memory and CPU usage, can be explained by our new data structure.

While B&B algorithms use a linked-list to encode a pool, our approach PB&B@CORE uses a new data structure, called Integer-Vector-Matrix (IVM). The operations of the conventional B&B, explained in Chapter 3, are designed to work on a pool encoded by a linked-list. It is therefore necessary to rethink these operations to work effectively on an IVM structure. This chapter, which contains three main sections, explains IVM and the PB&B@CORE operations.

Section 4.2 describes our new IVM structure, used for the PB&B pool coding. Section 4.3 explains the PB&B operations designed to work on this IVM structure. Finally, Section 4.4 presents the experiments carried out to validate the PB&B@CORE approach.

4.2 IVM data structure

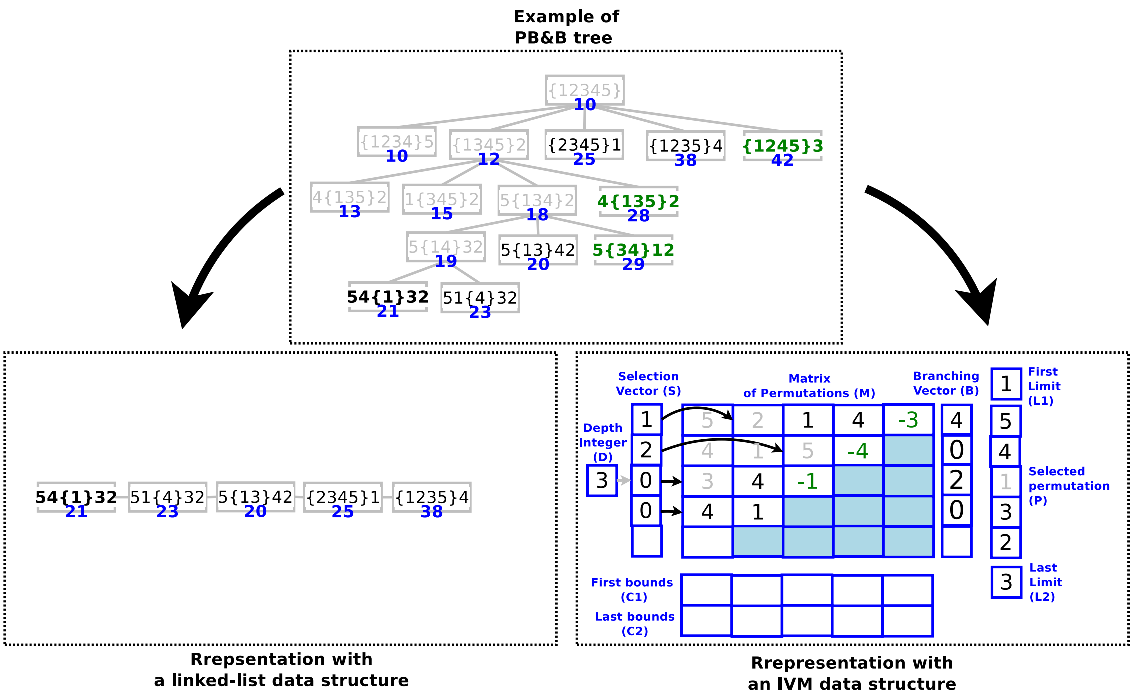


Figure 4.1: Example of representing a pool with a linked-list and an IVM structure.

Figure 4.1 shows an example of the representation of a pool with, on the one hand, a linked-list and, on the other hand, our IVM structure. This figure shows the PB&B tree during its construction and exploration. In the tree, the gray permutations are already branched, green permutations are pruned because of their bounds, and black permutations are neither branched nor pruned. In the linked-list, there is no need to keep gray and green permutations. The only permutations that must be kept are those which are black. Using the depth best first strategy described in Chapter 3, the size of the linked-list can reach $N\sum_{i=1}^{i=N-1} i$ integers, N being the size of the permutation problem. The objective of the IVM structure is twofold, namely the reduction of the pool size and accelerating the processing of this pool by its operations. The IVM structure is composed of six main sub-structures:

- **Permutation Matrix (M):** This structure is a matrix, denoted M in Figure 4.1, of size N^2, where N is the size of the permutation. In this matrix, only half of the cells are occupied. In order to encode a PB&B tree using this matrix, five rules are used:
  - **Rule 1**: Each cell $M_{i,j}$ of the matrix corresponds to a permutation $Perm(M_{i,j})$ of the PB&B tree. For example, $Perm(M_{0,1})$ is equal to $(\{1,3,4,5\},2)$.
  - **Rule 2**: The permutations of depth i of the tree are encoded at the row i of the matrix. For example, the depth 3 of the tree contains the permutations $(5,4,\{1\},3,2)$ and $(5,1,\{4\},3,2)$. These permutations are therefore encoded at the row 3 of the matrix, more exactly at cells $M_{3,0}$ and $M_{3,1}$.
  - **Rule 3**: In the same row of the matrix, the permutations are ordered according to their lower bound values. For example, in the row 3, $Bounding(5,4,\{1\},3,2)$ is smaller than $Bounding(5,1,\{4\},3,2)$ (i.e. 21 < 23). Therefore, $Perm(M_{3,0})$ is equal to $(5,4,\{1\},3,2)$, and $Perm(M_{3,1})$ is equal to $(5,1,\{4\},3,2)$.
  - **Rule 4**: Each cell $M_{i,j}$ contains only the last number assigned to its permutation $Perm(M_{i,j})$. For example, the cell $M_{0,1}$ has as permutation $(\{1,3,4,5\},2)$ and 2 is the last number assigned in this permutation. Therefore, the value of $M_{0,1}$ is equal to 2.
  - **Rule 5**: If k is the last element assigned to $Perm(M_{i,j})$ and $Perm(M_{i,j})$ must be pruned, then $M_{i,j}$ is equal to $-k$. For example, since $Perm(M_{0,4})$ is equal to $(\{1,2,4,5\},3)$, the last integer assigned to this permutation is 3 and the permutation $(\{1,2,4,5\},3)$ must be pruned, then $M_{0,4}$ is equal to -3.

- **Selection positions (S):** This structure is a vector, denoted S in Figure 4.1, of size N, where N is the size of the permutation. Each position S_i of this vector corresponds to the depth i of the PB&B tree. The value of S_i is equal to the rank of the last permutation selected at the depth i of the tree. For example, $(5,\{1,3,4\},2)$ is the last permutation selected in the depth 2 of the tree. This permutation occupies the rank 1 in this depth. Therefore, S_2 is equal to 1. This vector is called the selection vector because it keeps the ranks of the selected permutations at each level of the tree. As shown in Figure 4.1, each cell S_i of the vector can be seen as a pointer to a cell of the row i of the matrix M.

- **Branching positions (B):** This structure is a vector, denoted B in Figure 4.1, of size N, where N is the size of the permutation. Each position B_i of this vector corresponds to the depth i of the PB&B tree. The value B_i is equal to the position filled by the branching operation at the depth i of the tree. For example, at the depth 0 of the tree, integers are assigned to the position 4 of the permutation. Therefore B_0 is equal to 4. This vector is called branching vector because it keeps the filling positions at each level. Each cell B_i of the vector is therefore associated with a row i of the matrix M.

- **Depth position (D):** This structure is an integer, denoted D in Figure 4.1, which saves the depth of the next permutation to be selected. For example, in Figure 4.1, the next permutation to select is $(5, 4, \{1\}, 3, 2)$. In the PB&B tree, this permutation belongs to the depth 3. As a result, D is equal to 3. The value of D can also be viewed as a pointer to the last filled row of M.
- **Permutation structure (P) and its limits (L_1 and L_2):** This structure is composed of a vector, denoted P in Figure 4.1, and two integers, denoted L_1 and L_2. It allows to store any empty, full or partial permutation using the limits L_1 and L_2.
- **Bounding structure (C):** This structure, denoted C in the Figure 4.1, allows to store the lower bounds calculated by the bounding operation. The role of this structure is explained when the bounding operation is presented in Subsection 4.3.3.

4.3 PB&B Operations

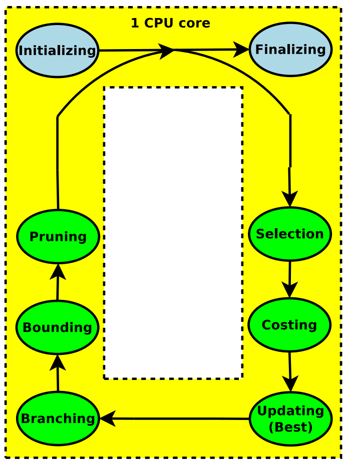


Figure 4.2: Overview of the PB&B@CORE approach and its operations.

The linked-list data structure is well suited for the conventional B&B algorithm. On the other hand, the IVM structure is exclusively designed for the PB&B algorithm, which is a special case of the B&B algorithm. As another data structure is proposed, it is necessary to adapt the operations of B&B to this structure.

4.3.1 Selection

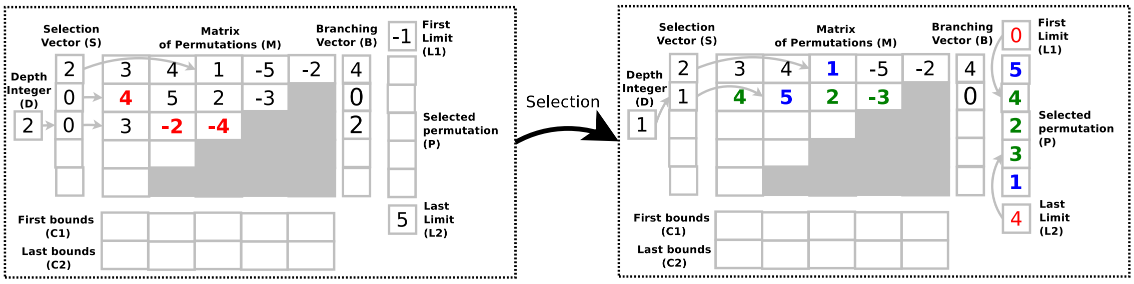


Figure 4.3: Illustration of the IVM-based selection of PB&B.

Like its linked-list counterpart, the selection operation returns a permutation, and writes it in the structures P, L_1 and L_2. This permutation is obtained using four steps:

- **Update of the depth integer D and the selection vector S:** This step works using five rules. These rules must be repeated until Rule 3 or Rule 5 indicates to stop the step. The five rules are given below:
  - **Rule 1:** In the beginning, S_D is incremented.
  - **Rule 2:** If M_{D,S_D} is negative, then S_D is incremented. The negative value of M_{D,S_D} means that $Perm(M_{D,S_D})$ is pruned.
  - **Rule 3:** If M_{D,S_D} is positive, then the step is stopped. In this case, a permutation is found and the pool is still not empty.
  - **Rule 4:** If S_D points after the last cell of the row, then the depth D is decremented and S_D is incremented.
  - **Rule 5:** If D points before the first row of the matrix, then the step is stopped. In this case, there is no permutation and the pool is empty.

  In the left side of Figure 4.3, S_D (i.e. S_2) points to $M_{2,0}$. After applying the five previous rules, the right side of Figure 4.3 is obtained. The application of these rules is done as follows:

  - The step starts by incrementing S_2, for the first time, and points it to $M_{2,1}$.
  - Because of the negative value of $M_{2,1}$, the S_2 is incremented, for the second time, to point to $M_{2,2}$.

  - Since $M_{2,2}$ is also negative, S_2 is incremented, for the third time, to point to $M_{2,3}$.
  - Because $M_{2,3}$ is located after the last cell of the row, D is decremented and S_D (i.e. S_1) is incremented.
  - Because S_1 points to $M_{1,1}$, which is a positive value, the step is stopped and the right side of Figure 4.3 is obtained.

- **Copying the cells pointed by the selection vector S to the permutation vector P:** In addition to S, the copy of this step also takes into account the values of the branching vector B. In the general case, all the elements M_{0,S_0}, M_{1,S_1}, ..., M_{D,S_D} in this order, and depending on whether each of the B_0, B_1, ..., B_D are equal to zero or not, the elements are respectively placed at the first or the last free position of the vector P.

  In Figure 4.3, the elements copied during this step are written in blue. In this example, there are only two copies to be made since the depth D points to S_1:

  - Since S_0 points to $M_{0,2}$, which is equal to 1, the integer 1 is copied to the vector P. Because B_0 is not equal to 0, the integer 1 is copied to the last free position of P, which is P_4.
  - Since S_1 points to $M_{1,1}$, which equals 5, the element 5 is copied to the vector P. Because B_1 is equal to 0, the element 5 is copied to the first free position of P, which is P_0.

- **Copying the elements from the last filled row of M to the permutation vector P:** The integer D always indicates the last filled row. On the right side of Figure 4.3, the elements of the last filled row are 4, 5, 2 and -3. Except the element 5, which is pointed by S_D, all these elements are still not copied into P. Therefore, the elements 4, 2 and 3 are copied into the three free positions of the vector P. The elements, copied during this step, are written in green in the Figure 4.3.

  In the general case, this step copies all the absolute values of the elements $M_{D,0}$, $M_{D,0}$, ..., $M_{D,N-D}$, except the element M_{D,S_D}, to the free positions of the vector P.

- **Update of the limits L_1 and L_2:** These two pointers indicate the free positions of the permutation P. In other words, all L_1, L_1+1, ..., L_2 positions of P are empty, and the elements P_{L_1}, P_{L_1+1}, ..., P_{L_2} are free. The value of the pointer L_1 is equal to the number of zeros of the vector B, and the value of the pointer L_2 is equal to N minus the number of non-zero values of B, where N is the size of the permutation. In this example, L_1 and L_2 are equal to 1 and 3, respectively.

4.3.2 Branching

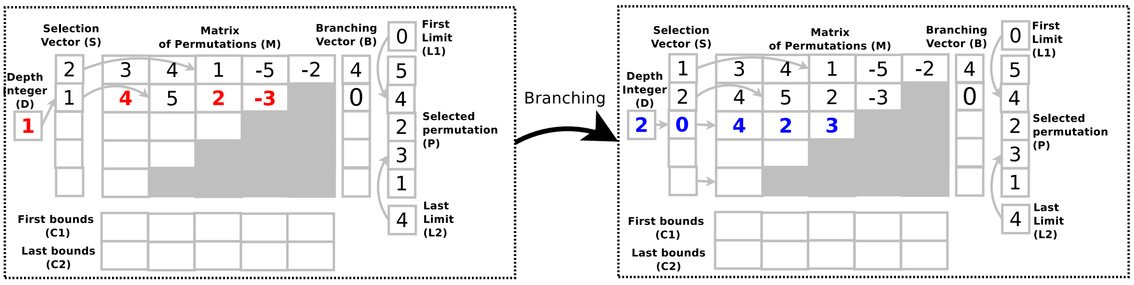


Figure 4.4: Example showing how branching operation works on IVM.

The branching operation is much simpler than the selection operation. This operation is done in three steps.

- **Step 1:** This step copies elements from the last filled row of M to the first empty row of M. During this operation, all the absolute values of the elements, except the one pointed to by S_D are copied. For example, in the left part of Figure 4.4, the elements of M, written in red, namely 4, 2 and -3, are copied. In the right part of Figure 4.4, it is possible to see these copied elements written in blue.
- **Step 2:** The role of this step is to increment the pointer D. In Figure 4.4, the value of D, which was 1 in the left side of the figure, becomes 2 in the right side.
- **Step 3:** This step points S_D to $M_{D,0}$ by writing 0 in S_D. As shown in Figure 4.4, S_2 points to $M_{2,0}$.

4.3.3 Bounding

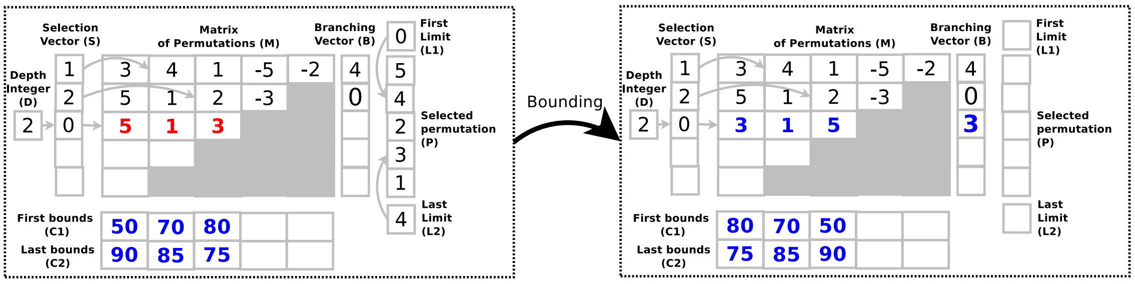


Figure 4.5: Illustration of the IVM-based bounding of PB&B.

The objective of the bounding operation is to update the elements of the last filled row of M, and the last filled position of B. Figure 4.5 shows an example of the state of the IVM structure before and after a bounding operation. This update is done using three steps:

- **Compute of all possible lower bounds:** In Figure 4.5, the permutations of the last filled row of the left IVM are written in red. The cells in this row correspond to those obtained after the branching of $(2, \{5, 1, 3\}, 4)$. Since the position B_2 is still empty, it is not known yet whether the elements $\{5, 1, 3\}$ are placed at the first empty position of the permutation P or at the last empty position of this permutation. Depending on the value of B_2, there are two cases:

  - **Case** $B_2 = 0$: This case corresponds to the case where one of the free elements is written in the first empty position of the permutation. As a result, the partial permutations of the last filled row correspond to $(2, 1, \{3, 5\}, 4)$, $(2, 3, \{1, 5\}, 4)$ and $(2, 5, \{1, 3\}, 4)$.
  - **Case** $B_2 = 3$: This case corresponds to the case where one of the free elements is written in the last empty position of the permutation. In this case, the partial permutations of the last filled row are $(2, \{3, 5\}, 1, 4)$, $(2, \{1, 5\}, 3, 4)$ and $(2, \{1, 3\}, 5, 4)$.

  For each of these six permutations, the obtained bound values are placed in cells $C_{1,0}$, $C_{1,1}$, $C_{1,2}$, $C_{2,0}$, $C_{2,1}$ and $C_{2,2}$. In this example, the value of these six bounds are assumed to be 50, 70, 80, 90, 85 and 75, respectively.

- **Choice of the bounding group:** The objective of this step is to choose either to place the free elements in the first free position or to place them at the last free position. In other words, the objective of this step is to determine if the value of B_2 is 0 or 3. In our algorithm, this choice is made by comparing the sum of the bounds of each group. Since the sum $(50 + 70 + 80 = 200)$ is smaller than $(90 + 85 + 75 = 250)$, the bounding operation chooses to place the free elements at the last empty position. This is why, in the right IVM of the figure, B_2 is equal to 3.

- **Sorting of the permutations according to the bound values:** In the second step, the selected partial permutations have the bounds 90, 85 and 75. These bounds correspond respectively to the filling of the fee elements 5, 1 and 3 in the last empty position of the permutation $(2, \{5, 1, 3\}, 4)$. The objective of this third step is to sort these free elements 5, 1 and 3 according to the increasing values of their corresponding bounds. As shown in the right IVM of Figure 4.5, the elements are sorted according to the ascending order of their bounds.

4.3.4 Pruning

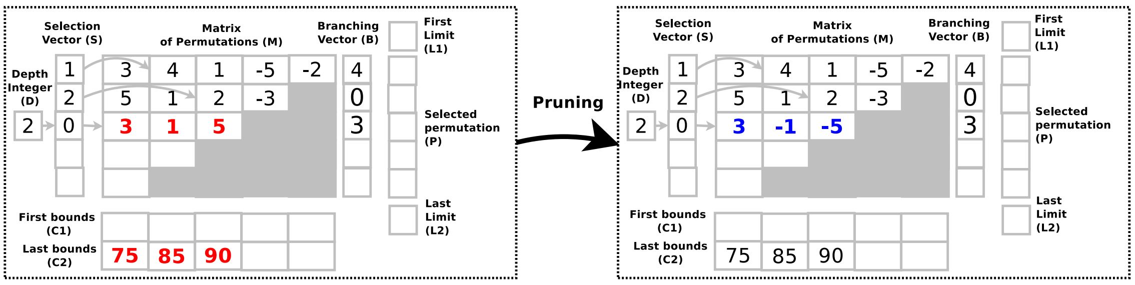


Figure 4.6: Example showing how pruning operation works on IVM.

At the end of the bounding operation, the bounds of the permutations corresponding to the cells of the last filled row of the IVM are calculated. The objective of the pruning operation is to eliminate some permutation of this last filled row. As explained before, any permutation, which has a higher bound than the cost of the best complete permutation found so far, must be pruned.

In the left IVM of Figure 4.6, the last filled row contains three cells. These cells correspond to the placement of the free elements 3, 1 and 5 in the last empty positions of the permutation $(2, \{3, 1, 5\}, 4)$. The obtained three sub-permutations are therefore respectively $(2, \{1, 5\}, 3, 4)$, $(2, \{3, 5\}, 1, 4)$ and $(2, \{3, 1\}, 5, 4)$. These permutations respectively have the bounds 75, 85 and 90. As the best complete permutation found so far has a cost of 80, the two permutations $(2, \{3, 5\}, 1, 4)$ and $(2, \{3, 1\}, 5, 4)$ are pruned. In our approach, pruning is done by multiplying cells by -1. Therefore, the elements in the last filled row are equal to 3, -1 and -5 respectively, as shown by the right IVM of Figure 4.6.

4.3.5 Initialization and finalization

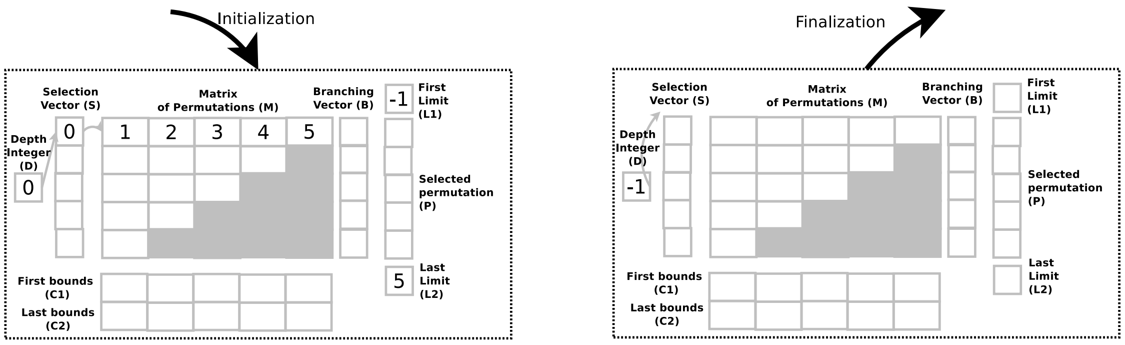


Figure 4.7: Example showing how initialization and finalization operations work on IVM.

In addition to the four previous basic operations, any PB&B must also use an operation to initialize the IVM, and another operation to detect the end of PB&B. The left IVM of Figure 4.7 shows the status of the IVM at its initialization, and the right IVM of this figure shows the state of IVM at its end. As shown in the figure, the structure is initialized as follows:

- The value of depth D is equal to 0;
- The first position P_0 of the position vector is equal to 0;
- The first row of the matrix M is initialized to 1, 2, ..., N, where N is the size of the permutation;
- The limits L_1 and L_2 are initialized to the values -1 and N;

The right IVM of the figure shows the state of the structure at the end of the PB&B algorithm. When the IVM is in this state, the PB&B algorithm must be stopped. The value of depth D is equal to -1.

4.4 Experiments

4.4.1 Experimental protocol

- **Hardware testbed:** All the experiments were run on the computer Poincaré which belongs to Maison de la Simulation (Saclay, Paris). Each of Poincaré's 92 CPU nodes is composed of 2 8-core Intel Xeon Sandy Bridge E5-2670 processors running at 2.60 GHz and has 32 Gb of memory. Each of the 16 physical cores has 32 KB of L1 instruction cache, 32 KB of L1 data cache and 256 KB of L2 cache. Each of the 2 processors has 20 MB of L3 cache. The 32 GB of memory are spread across 2 NUMA nodes, one for each processor.
- **Software tools:** The serial LL-based PB&B and PB&B@CORE have been implemented in C++ and compiled using GCC 4.6 with $-O3$ option. For each instance tested, the computational time spent in managing the pool of permutations is measured using the *clock_gettime* function with a nanosecond precision.
- **Problem instances:** In our experiments, we used the flowshop instances defined by Taillard [Tai93]. These standard instances are often used in the literature to evaluate the performance of methods that minimize the makespan. In the experiments of this chapter, we used the 10 instances defined with 20 jobs and 20 machines (these instances are named Ta021, Ta022, ..., and Ta030), and the 10 instances defined with 50 jobs and 10 machines (these instances are named Ta041, Ta042, ..., and

Ta050). The other instances are not used in our validation because they are either easy or difficult to solve with a serial algorithm.

- **Theoretical memory study:** It is possible to make a theoretical study of the maximum memory needed by the LL and IVM approaches to store their pool. For both approaches, the maximum size depends only on the number of jobs N of an instance. The size of the memory used by an IVM structure is always constant. In bytes, this size can be calculated using Equation (4.1).

$$\begin{aligned} Maximum - size(IVM) &= [\frac{N\ (N+1)}{2} + 3\ N + 1] \text{ bytes} \\ &= [\frac{1}{2}N^2 + \frac{7}{2}N + 1] \text{ bytes} \end{aligned} \tag{4.1}$$

Unlike IVM, the advantage of LL is to not require additional CPU calculation time for generating a permutation. In the LL approach, a conventional coding of a permutation is to write the list of scheduled jobs and the list of unscheduled jobs. By assuming that a job is encoded with 1 byte, the size of a permutation of an instance defined by N jobs is always equal to N bytes, and therefore, the size of a pool LL, which contains X permutations, is equal to $N \times X$ bytes. In LL, a pool reaches its maximum size when visiting the first complete permutation. At this moment, the pool contains $N-1$ permutations with 1 job scheduled, $N-2$ permutations with 2 jobs scheduled, ..., until $N-(N-1)$ (i.e. 1) permutation with $N-1$ jobs scheduled. The maximum size of a pool LL can be calculated using Equation (4.2).

$$\begin{aligned} Maximum - size(LL) &= [\sum_{i=1}^{i=N-1} (N-i)] \text{ permutations} \\ &= [\sum_{k=1}^{k=N-1} k] \text{ permutations} \\ &= [\frac{N\ (N-1)}{2}] \text{ permutations} \\ &= [\frac{N^2\ (N-1)}{2}] \text{ bytes} \\ &= [\frac{1}{2}N^3 - \frac{1}{2}N^2] \text{ bytes} \end{aligned} \tag{4.2}$$

In terms of space, this means that the IVM structure has a maximum of $O(N^2)$ complexity, while the LL data structure has a maximum of $O(N^3)$ complexity. Therefore, IVM approach can be up to N better than the LL approach in terms of memory size.

Besides, Table 4.1 concretely shows the ratio between the two approaches. These ratios are computed using Equation (4.1) and Equation (4.2). The values of this table are given for different sizes of jobs (i.e. 20, 50, 100, 200 and 500) of Taillard instances.

Instance size	**LL max. memory size (Bytes)**	**IVM max. memory size (Bytes)**	**IVM/LL max. memory ratio**
20	3800	271	14.07
50	61250	1426	42.98
100	495000	5351	92.52
200	3980000	20701	192.27
500	62375000	126751	492.11

Table 4.1: Comparison of PB&B@CORE and LL PB&B algorithms in terms of maximum memory.

However, the comparison of Table 4.1, in terms of maximum memory of both approaches, is not the best indicator to get an idea about the ratio of memory sizes really used by LL and IVM approaches. It is therefore important to compare the IVM and LL structures in terms of their memory size really used during a resolution. The average size of IVM is constant and is the same than the value given in Equation (4.2), while the average size of LL can not be deduced from a theoretical study. It is therefore necessary to solve an instance to know its average memory size when the resolution uses an LL structure.

4.4.2 Obtained results

- **Memory evaluation:** Tables 4.2 and 4.3 give the average obtained sizes for the instances defined with 20 jobs and 50 jobs, respectively. In these two tables, the first column gives the name of the instance, the second column the average number of permutations when using LL, the third column the average size in bytes for LL, the fourth column the average size in bytes for IVM which is constant, and the last column the ratio between the sizes of LL and IVM. The last row of the table gives the average of the values of each column.

  According to Table 4.1, the maximum expected ratio between the sizes of LL and IVM is equal to 20 for the instances defined by 20 jobs and 50 for the instances defined by 50 jobs. The experiments show that the average obtained ratios are respectively 9 and about 35. These results show that on average an IVM structure clearly occupies much less memory space than LL data structure.

Instance	LL average size (Permutations)	LL average size (Bytes)	IVM size (Bytes)	LL/IVM size ratio
Ta021	118	2360	271	8.74
Ta022	127	2540		9.41
Ta023	111	2220		8.22
Ta024	123	2460		9.11
Ta025	117	2340		8.67
Ta026	122	2440		9.04
Ta027	115	2300		8.52
Ta028	127	2540		9.41
Ta029	124	2480		9.19
Ta030	131	2620		9.70
Average	121.50	2430.00	271	9.00

Table 4.2: Comparison of the size of IVM and the average size of LL when solving the ten instances defined with 20 jobs.

Instance	LL average size (Permutations)	LL average size (Bytes)	IVM size (Bytes)	LL/IVM size ratio
Ta041	961	48050	1426	33.72
Ta042	980	49000		34.39
Ta043	1059	52950		37.16
Ta044	1085	54250		38.07
Ta045	824	41200		28.91
Ta046	1056	52800		37.05
Ta047	1038	51900		36.42
Ta048	1049	52450		36.81
Ta049	1094	54700		38.39
Ta050	993	49650		34.84
Average	1013.90	50695.00	1426	35.58

Table 4.3: Comparison of the size of IVM and the average size of LL when solving the ten instances defined with 50 jobs.

- **CPU Time evaluation:** As written in the previous subsection, it is clear that the IVM approach uses much less memory than the LL approach. However, unlike LL, IVM requires coding and decoding mechanisms of the permutations of the pool. A question then arises about the cost of IVM encoding and decoding mechanisms. Indeed, the gain in IVM memory should not be to the detriment of an additional computing cost to manage the pool. Therefore, the objective of this subsection is to compare the LL and IVM approaches in terms of pool management CPU time.

  In our comparative study, the pool management time does not only include the CPU time spent by reading and writing operations in the LL and IVM structures. This time also includes the CPU time spent by the selection, pruning and branching operations. In other words, the pool management time includes all the operations made in the PB&B algorithm except the the part of the bounding operation which computes the bounds.

Table 4.4 gives the average time obtained for the instances defined with 20 jobs, and Table 4.5 for the instances defined with 50 jobs. In both tables, the first column gives the name of the instance, the second column the pool management CPU time of the LL approach, the third column the pool management CPU time of the IVM approach, and the last column the ratio between the pool management CPU times of LL and IVM. The last row of the table gives the average of the values of each column.

The previous paragraphs show that the IVM structure uses much less memory space than the LL structure. As the information is encoded in IVM, unlike LL, it is intuitively logical to expect that the management of a pool coded with IVM takes more time than the management of a pool coded with LL. However, Table 4.4 shows that the management of the IVM pool takes on average about 2.75 less CPU time than the LL pool when solving instances defined by 20 jobs. In addition, Table 4.5 shows that this pool management takes on average about 3 times less CPU time than the LL pool when solving instances defined by 50 jobs.

These average ratios can be certainly explained by the adaptation made for the PB&B operations. The bounding, selection, branching and pruning operations are more optimized in the IVM approach compared to the LL approach. For example, the LL branching operation involves dynamically creating a certain number of permutations, and each permutation must contain a permutation almost similar to the permutation which is branched. Unlike this LL operation, the IVM branching operation merely copies the content of a matrix row to another row.

4.5 Conclusions

In this chapter, we proposed a new data structure, called IVM (Integer Vector Matrix), to implement the pool of permutations generated by a PB&B algorithm for solving permutation optimization problems. The bounding, selection, branching and pruning operations have been revisited to operate on this new data structure.

This new approach is validated using standard instances of the flowshop which is a permutation problem presented in the previous chapter. This evaluation is performed in terms of memory and CPU time usages. Experiments show that the use of IVM does not only greatly reduce the used memory size, but also reduces the CPU time spent for the pool management. Indeed, compared to LL, these experiments show that on average the IVM structure (1) occupies 9 times less memory space for the instances defined with 20 jobs, (2) uses about 35 times less memory for the instances with 50 jobs, (3) manages the pool about 2.5 times faster for instances with 20 jobs, and (4) manages the pool about 3 times faster for instances with 50 jobs.

Instance	LL time	IVM time	LL/IVM ratio
Ta021	411.09	149.66	2.75
Ta022	538.89	195.10	2.76
Ta024	393.28	146.01	2.69
Ta026	1425.14	537.28	2.65
Ta027	767.72	281.55	2.73
Ta028	133.54	47.71	2.80
Ta029	424.42	153.27	2.77
Ta030	76.45	28.10	2.72
Average	521.32	192.34	$\approx$ **2.75**

Table 4.4: Comparison of PB&B@CORE and LL-based PB&B algorithms in terms of the CPU time used for the management of the pool when solving the instances defined with 20 jobs.

Instance	LL time	IVM time	LL/IVM ratio
Ta041	7.45	2.44	3.05
Ta042	3235.67	1030.24	3.14
Ta043	690.58	228.17	3.03
Ta044	3.59	1.30	2.76
Ta045	19.60	6.08	3.22
Ta046	34.43	11.54	2.98
Ta047	153.76	51.66	2.98
Ta048	66.59	22.14	3.01
Ta049	4.74	1.78	2.66
Ta050	1396.59	452.31	3.09
Average	561.30	180.77	$\approx$ **3.00**

Table 4.5: Comparison of PB&B@CORE and LL-based PB&B algorithms in terms of the CPU time used for the management of the pool when solving the instances defined with 50 jobs.

CHAPTER 5

Multi-core Interval-based permutation B&B

Contents

5.1 Introduction

When running PB&B@CORE, the values of the position vector S are continuously updated. As described in Chapter 4, when the end of row is reached the algorithm backtracks to the previous level. Therefore, at level $D = 0, 1, \ldots, N-1$ the value of S_D is bounded by $S_D < N - I$. The vector is equal to 00000 when the algorithm points to the first complete permutation of the PB&B tree, and equal to 43210 when it points to the last complete permutation. Between these values, the vector successively takes the values 00010, 00100, 00110, 00200, 00210, $\ldots$, 43210. For each of these values, the algorithm points to a different complete permutation. There are 120 possible values because there are 120 complete permutations (i.e. $5!$). These 120 position vector values correspond to the numbering of the 120 complete permutations using a special numbering system, called factorial number system [Knu97]. The factorial number system, also called factoradic, is a mixed radix numeral system adapted to numbering permutations. For this number system, the French term used in 1888 [Lai88b] is *numération factorielle*.

In a multi-threaded approach, it is possible to ask a thread to explore only the complete permutations with factoradic numbers in the interval $[a, b[$, where a and b are factoradic numbers. In order to explore only these numbers, the thread initializes the vector position S to factoradic number a, launches the PB&B@CORE algorithm, and stops this algorithm when the number of the vector position S is equal to b. In this approach, the work unit of a PB&B thread is therefore an interval of factoradics. When a thread finishes exploring its interval, this thread can steal from another thread a portion of its interval in order to explore this portion. In a PB&B@CPU approach, these PB&B threads can be on the same CPU, or on multiple CPUs but sharing the same memory.

Figure 5.1 gives an overview of the PB&B@CPU approach, based on units of work defined by factoradics intervals. In this approach, only one thread is started at the beginning of a resolution. The role of this thread is (1) to initialize some data structures of the algorithm, (2) to launch a number of PB&B threads to explore the entire interval $[0, N![$, and (3) to restitute the complete permutation found, when the PB&B threads have finished exploring $[0, N![$. Compared to the PB&B@CORE approach, this new PB&B@CPU approach has an additional operation to allow the work stealing between the PB&B threads.

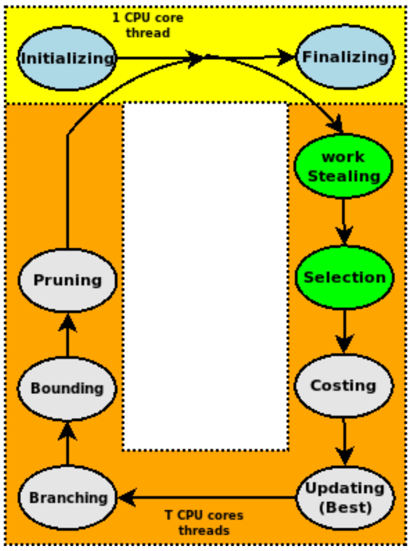


Figure 5.1: Overview of the PB&B@CPU approach and its operations.

This chapter is divided into three sections. Section 5.2 section gives an overview of the factorial number system, on which the work units of our approach are based. Section 5.3 describes the work stealing of the PB&B@CPU approach. Section 5.4 presents the experiments carried out to validate the PB&B@CPU approach using the flowshop problem.

5.2 Factoradic intervals

As the work unit of PB&B@CPU is defined by an interval of factoradic numbers, this section recalls some concepts related to number systems and describes the factorial number system.

5.2.1 Number systems

A number system, called also numeral system or system of numeration, is a writing system or a mathematical notation in order to express and represent a set of numbers using symbols in a consistent manner. A number system is defined by its digits, its bases, also called radixes, and its place values. This subsection reminds these concepts before their definitions for the factorial system are given in Subsection 5.2.2.

Digits:

Etymologically, the word *digit* comes from ancient Latin word *digit* which means fingers. Therefore, this word is related to the decimal system where ten digits are used like the ten fingers. However, the word digit is used nowadays for all other number systems including the binary system where the word bit is more appropriate. A number is a sequence of digits which can have an arbitrary length. Each digit, in a number system, represents an integer. In the decimal and the hexadecimal systems, for instance, the digits 1 and A represent the integers one and ten, respectively. In Roman numerals, where seven symbols are used, each symbol represents also a different integer as shown in Table 5.1.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

Table 5.1: Roman digits and their values.

There is a particular number system defined only with one digit. This simplest numeral system is called the unary numeral system, and can be used to represent all natural numbers. In order to represent any number N, an arbitrarily symbol, which represents the integer 1, is simply repeated N times. This system is often used in tallying. For example, using the tally mark |, the number 5 is represented as |||||. Unlike multiplication or division, the other operations, namely addition, subtraction and comparison, are particularly simple to be implemented in the unary system. Compared to other numeral systems, the unary system is not used in practice for large calculations but can be convenient for small operations, like representing a number with fingers. Unary system is used in some data compression algorithms such as Golomb coding [Gol66].

Place value:

Roman system, used in ancient Rome, employs combinations of letters from the Latin alphabet in order to express numbers. For example, the first ten numbers can be expressed as follows: I, II, III, IV, V, VI, VII, VIII, IX, X. In this system, numbers are written by combining symbols and adding or subtracting the values these symbols. For example, XIII means thirteen by adding a ten and three ones, and IX means nine by subtracting one from ten. Unlike the decimal system, there is no zero in Roman system and symbols do not represent tens or hundreds according to their positions. Therefore, unlike Roman system, which is not a positional system, decimal system is a positional numeral system.

Ancient number systems, like Roman system, were not positional, and all of the number systems most commonly used today, like binary system, are positional systems. Place value is a positional system of notation in which the position of a number determines its value. In other words, the value of a number in such system is determined not just by the digits but also by the positions of each of the digits. For example, all place values, also called order of magnitudes, in the unary system are equal to 1, the places values of decimal are powers of ten, like ones place, tens places, hundreds place, etc. One of the advantages of positional notation is the use of the same symbols for different order of magnitudes. This greatly simplifies arithmetic operations.

Radix:

Etymologically, the word radix is a Latin word for the word root, and root is a synonym for base in the arithmetical sense. In a positional numeral system, the radix is the number of unique digits, including zero, used to represent numbers. For example, the radix is ten for decimal system since this system uses ten digits from 0 through 9 in order to represent its numbers. In a positional numeral system, the number X and the radix Y are conventionally written as $(X)_Y$. However, the radix can be implicitly assumed and not written for some systems like decimal or unary systems.

Mixed radix numeral systems are non-standard positional numeral systems. Unlike most common systems, where the base is similar to all positions, the numerical base can vary from position to position. Such representation is used when a value or a number is written using a sequence of units that are each a multiple of the next smaller unit. For example, this type of number systems can be found when expressing time. A time of 10 hours, 30 minutes, and 50 seconds might be expressed as $10 : 30 : 50$ in mixed radix notation where the radix of the second and third positions is 60 and the radix of the first position is 24.

In positional fixed radix number system, where the base R is fixed, each digit a_i in any number $(a_{n-1}...a_0)_R$ is an integer in the range 0 to $(R-1)$ and the number is interpreted as shown in Equation (5.1).

$$(a_{n-1}...a_0)_R = a_{n-1}R^{n-1} + ... + a_1R^1 + a_0R^0 \tag{5.1}$$

Since Equation (5.1) is a polynomial in R, fixed radix system can be also called polynomial system. The decimal and binary systems are both fixed radix systems, with a radix of 10 and 2, respectively. Fractional values can also be represented with the same polynomial notation.

$$0.a_1a_2...a_n = a_{-1}R^{-1} + a_2R^{-2} + ... + a_nR^{-n} \tag{5.2}$$

In mixed-base or radix number system, the digit a_i in any number belongs to the interval 0 to R_i, where R_i is not the same for all the values of i. The number is then interpreted as shown in Equation (5.3).

$$a_{n-1}...a_0 = (...((a_{n-1}\ R_{n-1}) + a_{n-2})R_{n-2} + ... + a_1)R_0 + a_0 \tag{5.3}$$

5.2.2 Factorial number system

Factorial system, also called factoradic, is a mixed radix number system which is well adapted for numbering permutations. This system is not named like most numeral systems. For example, unary, binary and decimal are named like this because their radixes are one, two and ten, respectively. Unlike these systems, the factorial system is named according to its place value instead of its mixed radix. The term factorial number system is used the first time recently in 1998 [Knu98] while the French name *numération factorielle* is first used in 1888 [Lai88c]. The term factoradic appears to be much more recent [McC03]. General properties of mixed radix number systems also apply to the factorial system.

As explained in Table 5.2, the i^{th} digit from the right has base i and the place value $i!$. Therefore, the i^{th} digit must be less than i. And in order to compute the value of a number, the value of the i^{th} digit must be multiplied by $i!$. From this, it follows that the rightmost digit is always 0, the second can be 0 or 1, the third 0, 1 or 2, and so on.

Place	...	7th	6th	5th	4th	3rd	2nd	1st
Radix/base	...	7	6	5	4	3	2	1
Place value	...	6! = 720	5! = 120	4! = 24	3! = 6	2! = 2	1! = 1	0! = 1
Allowed digits	...	0 1 2 3 4 5 6	0 1 2 3 4 5	0 1 2 3 4	0 1 2 3	0 1 2	0 1	0

Table 5.2: Factorial system and its radixes, place values and digits for the seven first positions.

It is possible to define factorial numbers without writing the rightmost digit since it is always equal to zero. In our report, a factorial number representation will be flagged by a subscript "!", so for instance $(322110)_!$ stands for $(3_{5!}2_{4!}2_{3!}1_{2!}1_{1!}0_{0!})$. In principle, the factorial system may be extended to represent fractional numbers. However, the natural extension of place values (-1)!, (-2)!, etc. are undefined. In factorial system, the symmetric choice of radix values n = 0, 1, 2, 3, 4, etc. after the point may be used instead. The corresponding place values are therefore $1/(0!)$, $1/(1!)$, ..., $1/(n!)$, etc. In our work, factorial fractional numbers are not used.

Decimal to factoradic:

When converting a decimal number into its factorial representation, digits are produced from right to left. This conversion consists in repeatedly dividing the number by the radixes 1, 2, 3, etc. After each division, the remainder should be considered as the digit. The division operation continues with the integer quotient until this quotient becomes 0. Let assume a decimal number $D = 349$ to convert into a factorial number. This conversion is done using successive Euclidean divisions as shown in Equation (5.4).

$$\begin{aligned} 349 &= 1 \times 349 + 0 \\ 349 &= 2 \times 174 + 1 \\ 174 &= 3 \times 58 + 0 \\ 58 &= 4 \times 14 + 2 \\ 14 &= 5 \times 2 + 4 \\ 2 &= 6 \times 0 + 2 \end{aligned} \tag{5.4}$$

Euclidean division is the operation of division of two integers, which produces a quotient and a remainder. Each line of Equation (5.4) represents an Euclidean division $Q_i = i \times Q_{i+1} + R_i$ such as:

- Q_{i+1} is the quotient of the division $\frac{Q_i}{i}$
- R_i is the reminder of this division
- $Q_1 = A$ is the decimal number to convert
- Q_{n+1} is always equal to zero and is the last quotient
- R_1 is always equal to zero and is the first remainder

The factorial representation F of A is equal to concatenation of all reminders, as shown in Equation (5.5), is $F = (R_n...R_2R_1)_! = 242010$.

$$\begin{aligned} 349 &= 1 \times 349 + 0 \\ &= 1 \times (2 \times 174 + 1) + 0 \\ &= 1 \times (2 \times (3 \times 58 + 0) + 1) + 0 \\ &= 1 \times (2 \times (3 \times (4 \times 14 + 2) + 0) + 1) + 0 \\ &= 1 \times (2 \times (3 \times (4 \times (5 \times 2 + 4) + 2) + 0) + 1) + 0 \\ &= 1 \times (2 \times (3 \times (4 \times (5 \times (6 \times 0 + 2) + 4) + 2) + 0) + 1) + 0 \\ &= 2\ 5! + 4\ 4! + 2\ 3! + 0\ 2! + 1\ 1! + 0\ 0! \\ &= (242010)_! \end{aligned} \tag{5.5}$$

Algorithm 1 explains the used method to perform this conversion.

Algorithm 1 DECIMAL-TO-FACTORIAL(D)

```
1: Place $\leftarrow$ 1
2: while D $\neq$ 0 do
3:     $F_{i-1}$ $\leftarrow$ D mod i
4:     D $\leftarrow$ D div i
5:     i $\leftarrow$ i+1
6: end while
7: return F
```

Factoradic to decimal:

Converting a factorial number to a decimal number is simpler. Let $(R_{n-1}...R_1R_0)_!$ a factorial number. In order to have its decimal equivalent, it suffices to calculate the value of the polynomial $\sum_{i=0}^{i=n-1} R_i\ i!$. The conversion of a factorial number to its decimal equivalent is therefore a sum of multiplications, while the conversion of a decimal number to its factorial equivalent is a concatenation of divisions. Algorithm 2 explains this sum of multiplications.

Algorithm 2 FACTORIAL-TO-DECIMAL(F)

```
1: Place-value $\leftarrow$ 1
2: D $\leftarrow$ 0
3: for i $\leftarrow$ 1 to $n$ do
4:     Place-value $\leftarrow$ Place-value $\times$ i
5:     D $\leftarrow$ D + $F_i$ $\times$ Place-value
6: end for
7: return D
```

Addition:

Let assume two factorial numbers $A = (A_{n-1}...A_1A_0)_!$ and $B = (B_{n-1}...B_1B_0)_!$ to be added and both having the size n. As these numbers are factorial, both conditions $\forall i, A_i \leq i$ and $\forall i, B_i \leq i$ are always true. The addition of A and B would be simple if $\forall i, A_i + B_i \leq i$. In this case, the result of the addition would be $C = (C_{n-1}...C_1C_0)_!$ such as $\forall i, C_i = A_i{+}B_i$. However, the condition $\forall i, A_i + B_i \leq i$ is not always satisfied. Let assume i such as

$A_i + B_i > i$. The value of C_i can be calculated as explained in Equation (5.6).

$$\begin{aligned} C_i &= A_i\ i! + B_i\ i! \\ &= [A_i + B_i]i! \\ &= [(i+1) - (i+1) + (A_i + B_i)]i! \\ &= (i+1)i! + [-(i+1) + (A_i + B_i)]i! \\ &= (i+1)! + [(A_i + B_i) - (i+1)]i! \end{aligned} \tag{5.6}$$

Therefore, there is a carry $+1$ for the calculation of C_{i+1}, and $C_i = (A_i + B_i) - (i+1)$ since the rule of Equation (5.7) is always true.

$$[(A_i \leq i)\ and\ (B_i \leq i)\ and\ (A_i + B_i > i)] \Rightarrow 0 \leq [(A_i + B_i) - (i+1)] \leq i \tag{5.7}$$

Equation (5.8) gives the value of C_i for any values of A_i and B_i.

$$C_i = \begin{cases} (A_i + B_i) & If\ (A_i + B_i) \leq i \\ (A_i + B_i) - (i+1) & Otherwise \end{cases} \tag{5.8}$$

Algorithm 3 explains the method used for the addition of A and B. The algorithm proceeds from the least significant position to the most significant one, in other words, from right to left. So, it is possible to check whether there is a carry $+1$ when computing C_i. If this is the case, this carry is taken into account when computing C_{i+1}.

Algorithm 3 FACTORIAL-ADDITION(A, B)

```
1: for i ← 0 to (n-1) do
2:     $C_i \leftarrow C_i + A_i + B_i$
3:     if $C_i > i$ then
4:         $C_i \leftarrow A_i - (i+1)$
5:         $Ci + 1 \leftarrow 1$
6:     end if
7: end for
8: return C
```

Subtraction:

Let assume two factorial numbers $A = (A_{n-1}...A_1A_0)_!$ and $B = (B_{n-1}...B_1B_0)_!$. As these numbers are factorial, both conditions $\forall i, A_i \leq i$ and $\forall i, B_i \leq i$ are always satisfied. The objective of this subsection is to explain how to perform a subtraction $A - B$ when assuming $A \geq B$. To facilitate the explanation, both numbers are assumed to have the same size n. If the size of B is smaller than A, then it is possible to complete B with zeros at the left.

The subtraction $A - B$ would be simple if $\forall i, A_i \geq B_i$. In this case, the result of the subtraction would be $C = (C_{n-1}...C_1C_0)_!$ such as $\forall i, C_i = A_i - B_i$. However, the condition $\forall i, A_i \geq B_i$ is not always satisfied. Let assume i such as $A_i < B_i$. The value of C_i can be computed as explained in Equation (5.9).

$$\begin{aligned} C_i &= A_i\ i! - B_i\ i! \\ &= [A_i - B_i]i! \\ &= [-(i+1) + (i+1) + (A_i - B_i)]i! \\ &= [-(i+1)]i! + [(i+1) + (A_i - B_i)]i! \\ &= [-1](i+1)! + [(i+1) + (A_i - B_i)]i! \end{aligned} \tag{5.9}$$

Therefore, there is a carry -1 to be taken into account when computing C_{i+1}, and $C_i = (A_i + B_i) - (i+1)$ since the rule of Equation (5.10) is always true.

$$[(A_i \leq i)\ and\ (B_i \leq i)\ and\ (A_i < B_i)] \Rightarrow 0 \leq [(A_i - B_i) + (i+1)] \leq i \tag{5.10}$$

Equation (5.11) gives the value C_i for any values of A_i and B_i.

$$C_i = \begin{cases} (A_i - B_i) & If\ (A_i - B_i \geq 0) \\ (A_i - B_i) + (i+1) & Otherwise \end{cases} \tag{5.11}$$

Algorithm 4 explains the operation of the subtraction $A - B$. Like the addition, the subtraction algorithm proceeds from the least significant position to the most significant position. It is possible to check if there is a carry -1 when computing C_i. If this is the case, this carry is taken into account when computing C_{i+1}.

Algorithm 4 FACTORIAL-SUBTRACTION(A, B)

```
1: for i ← 0 to $(n-1)$ do
2:     $C_i \leftarrow C_i + A_i - B_i$
3:     if $C_i < 0$ then
4:         $C_i \leftarrow C_i$ + (i+1)
5:         $C_{i+1} \leftarrow$ -1
6:     end if
7: end for
8: return C
```

5.3 Interval-based work stealing

A work stealing strategy can be defined by three major components: a work unit definition, a victim selection policy and a granularity policy. The victim selection policy determines how a thief thread R chooses its victim S. The granularity policy determines the amount and which part of work thread R steals from thread S. An ideal victim selection strategy is one which (1) chooses the victim S with the largest amount of work, (2) and makes this choice as rapidly as possible. A good granularity policy reduces the number of work stealing operations, meaning that it allows both victim and thief to work as long as possible without initiating another work stealing event. This section presents respectively the work unit of our approach, its victim selection policy and its granularity policy.

5.3.1 Work unit: interval of factoradics

Dynamic load balancing is well suited for multi-core parallel PB&B algorithms. All threads must share their work. To the best of our knowledge, except in rare works such as [MMT07a], work units exchanged between threads (or processes) are sets of nodes. Since the nodes explored in our approach are numbered, it is possible to define an interval of node numbers as the work unit. Let's assume the interval that must be explored by a PB&B algorithm is $[00000, 43210[$. It is therefore possible to have two threads *T1* and *T2* such as *T1* explores $[00000, X[$ and *T2* explores $[X, 43210[$. If *T2* ends exploring its interval before *T1*, then *T2* sends a request to *T1* to recover a portion of its interval. Therefore, *T1* and *T2* can exchange their interval portions until the exploration of the whole $[00000, 43210[$ interval.

In order to emphasize on the difference between a conventional set of nodes and an interval of factoradics, it should be noted that the notion of *work* is slightly different. Indeed, a work unit $[A, B[$ does not necessarily contain any nodes to branch. In that sense it rather represents a potential amount of work. The processing of an interval of factoradics amounts to scanning the corresponding part of the search space. In contrast, the set of pending permutations contains only nodes that have actually been generated and not been removed from the tree.

In [MMT07a], we have presented an original load balancing strategy where the work unit is an interval of integers. Compared to [MMT07a], the new strategy presented in this chapter brings two new contributions. The first contribution is that the PB&B is based on a matrix of integers instead of a linked-list of permutations of integers, and the second is that it is not necessary to use the fold and unfold operations defined in [MMT07a] to transform an interval into a linked-list of nodes and *vice versa*. Besides, the integers of

[MMT07a] are coded using the decimal system, while in this chapter, the integers are coded using the factoradic system.

5.3.2 Work unit communication

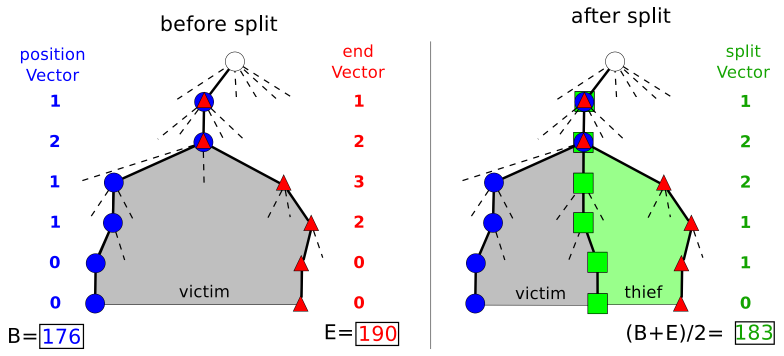
In this subsection two alternative protocols for the communication of work units between PB&B@CPU threads are described. An illustration of both procedures is shown in Figure 5.2.

- **Work unit splitting at an arbitrary position:** An integer interval $[B, E[$ can be split at any integer C that is convex linear combination of B and E, i. e. $C = \lfloor(1-\alpha)A + \alpha B\rfloor$, $0 < \alpha < 1$. The computation of a splitting point C can be performed either by using decimal arithmetic operations or by implementing elementary arithmetic operations for factoradic numbers. The granularity of this procedure is controlled by the value of α. In the example shown in Figure 5.2a, the interval of the victim is split in two parts of equal size, setting $\alpha = 0.5$.

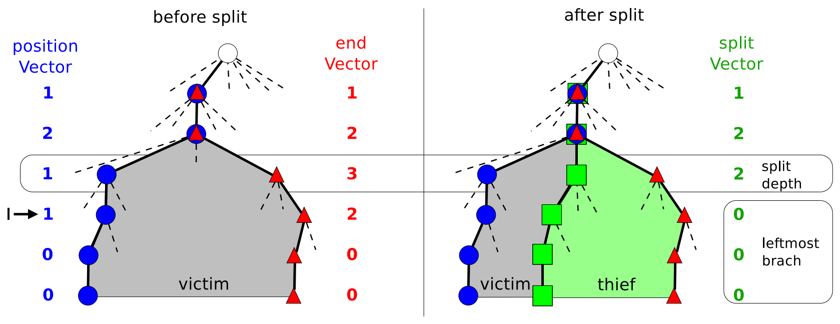
  In both cases it is necessary to have a procedure that initializes the IVM structure at an arbitrary valid position vector $V = C$. For that, it is not enough to build the matrix by iterative application of the branching operation, selecting the jobs pointed by V, because the information about pruned nodes is lost.

  Initialization of the IVM structure at any position vector $V = C$ can be achieved as follows. Starting from $I = 0$ all nodes pointed by V are expanded, bounded and pruned until the last line is reached, i. e. until $I = n$. In other words, n iterations of a modified PB&B algorithm, selecting the permutations indicated by V, are performed. As this initialization procedure involves the bounding operation, initialization overhead can become significant.

  A first observation can be made: this initialization process can actually be stopped when V points to a pruned permutation. This reduces the number of initialization steps considerably. A second observation allows to further decrease the amount of time spent in initialization: Suppose that an IVM structure was used to explore an interval $[B, E[$ and that it has reached the end of this task, that is $B = E$. Let l be the level at which the exploration stopped. Now we want to initialize this IVM at a new position $V = \tilde{B}$. If $B[i] = \tilde{B}[i]$ for $i = 0, 1, ..., k$ with $k < l$, then these first k lines of the matrix M are already correctly initialized. The initialization process described before can thus begin at $I = k$.



(a) Arithmetic interval splitting



(b) Subtree-based interval splitting

Figure 5.2: Illustration of work unit splitting.

- **Subtree-based work unit splitting:** In [Ler15b] a method for exchanging work units between PB&B@CPU threads without initialization is proposed. This procedure is based on splitting the interval of the victim IVM directly in its factoradic form, without converting it to decimals. In addition to the new position and end vectors, an initialized matrix is transferred from the victim IVM to the thief IVM. Figure 5.2b illustrates this procedure. The transfer of a work unit from IVM S (Sender) to IVM R (Receiver) can be performed as follows.

  1. Let $[B_S, E_S[$ be the interval to split. Let l be the smallest index such that $B_S[l] \neq E_S[l]$.
  2. For $i = 0, 1, \ldots, l-1$, copy row i of IVM from S to R (position, end, direction vectors and matrix).
  3. Split the tree at level l by choosing C such that $B_S[l] < C \leq E_S[l]$. For the receiving IVM R, set $B_R[l] = C$ and $E_R[l] = E_S[l]$. For the sending IVM S, $E_S[l] = C - 1$.

4. For $i = l + 1, \ldots, n - 1$, set $E_S[i] = n - i - 1$ and $B_R[i] = 0$.
5. The receiving IVM R is now initialized and exploration can start at level l. Therefore I_R is set to $I_R = l$.

This initialization method proposed by [Ler15b] requires no additional computation of bounds. Therefore it reduces the overhead induced by work stealing operations, compared to the previously introduced initialization procedure. Another important advantage of this method is that it avoids redundant computations. Indeed, if intervals are split at arbitrary points, some permutations may be branched redundantly. In this example, thread T1 explores interval $[0000, 1100[$ and thread T2 explores $[1110, 2110[$, which may cause redundant computation of bounds along the frontier $11XX$.

However, there are also disadvantages. Especially in a distributed memory setting the size of data transfers should be kept low. Indeed, this is the primary motivation for using interval encoded work units [MMT07c]. Also, in the subtree-based interval splitting, the granularity is controlled in a coarser way because *full* subtrees are communicated. This method is similar to stack splitting strategies where nodes (and implicitly the subtrees rooted in these nodes) are transferred from non-empty to empty pools. Dividing intervals by the first method allows a finer control of granularity.

We use the second (subtree-based) work splitting method for communicating work units *via* shared memory and the first method in distributed memory contexts.

5.3.3 Victim selection policies

In this Subsection, four victim selection policies are described. Two of them, the *random* and *ring* policies have a low computational complexity and require no access to shared data structures or knowledge about the global workload repartition. The two other selection policies, namely the *largest* and the *honest* policies, use simple heuristics which aim at selecting a stealing victim holding a large or difficult piece of work. These victim selection policies use the available information about the workers' activity and require some additional computation and protected accesses to shared data structures.

- ***Ring victim selection policy***: In this deterministic policy, threads are connected to each other with an unidirectional ring. A thread R always steals its precedent thread R'. If the thread R is different from the thread 1, then the thread R' is equal to the thread $R - 1$. Otherwise, the thread R' is equal to the thread T, where T is the number of threads. In this policy, the work stealing operation of a thread R is a blocking event when the thread R' has no work. In this case, the work stealing operation will be satisfied when the thread R' will receive work. This policy is also used, for instance in [KRR88]. If the thread numbering is matched with the underlying architecture the deterministic nature of this policy can be used to reduce communication costs.

The average distance to another thread is $T/2$. For a large number of threads this distance should be decreased and each thread should have more than one neighbor. Moreover the possibility of two threads selecting each other mutually must be excluded. For instance, the lifeline scheme [SKK+11], based on cyclic hypercubes, satisfies these properties and has been shown to be scalable to clusters of several thousand cores. The ring strategy corresponds indeed to the simplest form of the lifeline scheme, to which it could be extended when a large number of threads is used. As shown in function *choose-ring*, the cost of the victim selection function is very low. The main issue of this strategy is that work units may not propagate fast enough through the ring. No locking or access to shared data structure is required by this strategy: the only information an idle worker needs to select a victim is its own thread/process ID.

- ***Random victim selection policy***: The *random* selection policy is provably efficient [BL99] and the most frequently used in the literature. In this policy, a thread victim R' is randomly selected when a thread R initiates a work stealing operation. Unlike the ring policy, this work stealing operation is not a blocking event. In other words, the thread R continues to choose other threads randomly as long as it does not find a thread with a non-empty interval or linked-list. Besides its own thread/process ID a thief only needs to access a variable that indicates the state of the randomly selected victim.

- ***Largest victim selection policy***: In a PB&B algorithm, it is often impossible to determine the hardness of a work item. This policy is based on a simple heuristic to choose the thread with the most difficult work to finish. Indeed, the largest policy assumes that probably the larger the size of a work is, the more difficult this work will be. Therefore, this policy computes the amount of work of each thread, chooses the thread with the biggest size, and returns the rank R' of this thread. In the linked-list based approach, the size of a linked-list is equal to the number of its nodes, and in the interval-based approach, the size of an interval $[A, B[$ is equal to $B - A$. As shown in function *choose-largest*, this policy has a higher computational complexity than the three other victim selection policies. In particular a thief requires locked accesses the *length* or *size* variable of each busy worker. Although this polling may compromise the scalability of this strategy, good results for this policy are reported in [ASW+14].

Algorithm 5 Pseudocode of the victim selection policies.

```
function CHOOSE-THREAD(R, strategy)
    switch strategy do
        case RING:
            return CHOOSE-RING(R)
        case RANDOM:
            return CHOOSE-RANDOM(R)
        case LARGEST:
            return CHOOSE-LARGEST(R)
        case HONEST:
            return CHOOSE-HONEST(R)
end function
function CHOOSE-RING(R)
    return $(R-1)\%T$
end function
function CHOOSE-RANDOM(R)
    while true do
        R’←random(1,T)
        if (has-work(R’) AND (R’≠R)) then
            return R’
        end if
    end while
end function
function CHOOSE-LARGEST(R)
    max-size←0
    for all R" ∈ $\{1, 2, ..., T\}$ AND (R"≠R) do
        if (size(R")> max-size) then
            R’←R"
            max-size←size(R")
        end if
    end for
    return R’
end function
function CHOOSE-HONEST(R)
    remove(rank-threads,R)
    while not-empty(rank-threads) do
        R’←pop-front(rank-threads)
        if (has-work(R’) then
            push-back(rank-threads,R)
            return R’
        end if
    end while
end function
```

- ***Honest victim selection policy***: This strategy is based on another heuristic to determine the thread with the most difficult work to finish. The heuristic assumes that if a thread R_1 has stolen work less recently than a thread R_2, then the thread R_1 has probably a work which is more difficult than the work of the thread R_2. Therefore, the thread R steals the work from the thread victim R' which is the least recent thief. As shown in function *choose-honest*, this policy has a higher computational complexity than the ring and random policies but a smaller computational complexity than the largest policy. In the largest victim selection policy, it is important to compute the amount of work of each pool and computing the size of any pool is a blocking operation for the thread which owns this pool. In the honest victim selection policy, one operation of removing is performed on the rank threads list, and this operation is non-blocking. The operations on the global rank threads list must be protected by locks.

5.3.4 Granularity policies

When a thread R' is contacted by a thread R, the thread R must determine the amount of work of its thread victim R' to steal. In this chapter, two granularity policies are used.

- ***Steal half policy***: This policy indicates that the thread R steals the second half of the work of the thread R' and leaves the other half for the thread R'. In the linked-list based approach, the work of a thread R' is constituted by a set of N nodes. The thread R steals the last $N/2$ nodes and leaves the other nodes for the thread R'. Nodes are always stolen from the tail, i.e. from the end which is opposite to the working end of the private deque. While in the interval-based approach, the work of a thread R' is constituted by an interval $[A, B[$. The thread R steals the interval $[(A+B)/2, B[$ and leaves the interval $[A,(A+B)/2[$ for the thread R'. Leaving the first half of the interval $[A, B[$ avoids the thread R' to initialize its matrix and vectors.

- ***Steal T^{th} policy***: Theoretically, steal half policy may not be appropriate for certain victim selection policies. Suppose, for instance, four threads where thread 1 has a certain amount of work W, and threads 2, 3 and 4 have completed their work. The amount of work W may be the number of nodes or the size of the interval. In a ring selection, the threads 2, 3 and 4 steal work from the threads 1, 2 and 3, respectively. Using the steal half policy and the ring selection, the amounts of work $W/2$, $W/4$, $W/8$ and $W/8$ are allocated to the threads 1, 2, 3 and 4, respectively. Steal T^{th} policy indicates that the thread R leaves W/T of the work to its thread victim R', where T is the number of threads, and steals $(T-1)W/T$ of the work. In the previous example, using steal T^{th} policy and the ring selection allocate the amount of works $W/4$, $3W/16$, $9W/64$ and $27W/64$ to the threads 1, 2, 3 and 4, respectively. For this example, steal Tth policy gives a better granularity policy than the steal half policy. In our experiments, steal Tth policy is tested only for the ring selection. Indeed, steal half policy seems to be theoretically appropriate for the other victim selection policies.

5.4 Experiments

In this section, we present an experimental study to evaluate the performance of the different work stealing strategies for our PB&B@CPU approach on a multi-core system. These strategies are compared with the multi-threaded linked-list based PB&B (CPU LL PB&B) approach when solving some hard flowshop scheduling problem instances.

5.4.1 Experimental protocol

- **Hardware testbed:** All the experiments are run on a computer composed of 2 8-core Sandy Bridge E5-2650 processors and 32 GB of memory.

- **Software testbed:** The operating system installed is a CentOS 6.5 Linux distribution. The UNIX `time` command is used to measure the elapsed execution time for each flowshop instance and time measures for specific parts of the algorithms are obtained using the `clock_gettime` function with nanosecond precision.

- **Problem instances:** In our experiments, we used only the 10 instances where the number of machines and the number of jobs are equal to 20. Instances where the number of machines is equal to 5 or 10 are easy to solve. For these instances, the used bounding operation gives such good lower bounds that it is possible to solve them in few seconds using a multi-core PB&B. Instances where the number of jobs is equal to 50, 100, 200, or 500, and the number of machines is equal to 20 are very hard to solve.

- **PB&B initialization:** When an instance is solved twice using a PB&B performing a parallel tree exploration, the number of explored permutations is often different between the two resolutions, because the nodes are expanded in a different order. Moreover, the parallel algorithm is likely to explore a (much) larger or smaller number of nodes than the serial version of the algorithm. This may lead to speed-up anomalies, i.e. speed-ups greater that P on P cores.

To compare the performance of two PB&B algorithms, the explored search space should be exactly the same between the different tests. In order to study the performance of the parallel algorithm in the absence of speed-up anomalies, we choose to always initialize our PB&B by the optimal complete permutation of the instance to be solved. With this initialization it is ensured that both, the sequential and parallel algorithms explore exactly the critical subtree. The critical subtree is composed of all nodes for which the bounding operation gives a lower bound smaller than the cost of the optimal complete permutation. It corresponds to the part of the search space which a PB&B algorithm must explore to prove the optimality of the optimal complete permutation.

It is not unusual to use this technique in the study of parallel PB&B algorithms (see, for instance [KRR88]). Although this situation is not realistic in practice, an initialization to ∞ (or $-\infty$ in the case of a maximization problem) is neither realistic in practice. Indeed, the number of branched nodes that do not belong to the critical subtree can be reduced by improving the selection strategy or by initializing the algorithm with a good, suboptimal complete permutation produced by a metaheuristic. However, even with a suboptimal initialization of the algorithm the size of the explored search space still varies between two parallel resolutions. The initialization of the algorithm to the optimal cost ensures that the size and share of the search tree do not depend on the decrease of the best complete permutation found so far and that the number of explored permutations is the same between the two sequential or parallel resolutions. This allows a fair comparison between the two versions.

- **Sequential PB&B:** Table 6.1 shows the number of nodes branched during the resolution of instances *Ta021–Ta030* initialized with the optimal complete permutation. This number represents the total amount of work to be done and ranges from 1.6 for the smallest to 140.8 million nodes for the largest instance. It also shows the elapsed execution times for the resolution of these instances with sequential linked-list PB&B and PB&B@CORE algorithms. It should be noted that the resolution time is not directly proportional to the number of branched nodes. The node processing speed (in nodes/sec), also shown in Table 6.1, varies from one instance to another, as a result of the trees irregularity (the cost of the bounding operation depends on the depth of a permutation). As the flowshop bounding operation consumes between 97% and 99% of the elapsed time, the choice of the data structure, LL or IVM, has no significant impact on the elapsed execution time. In order to speed up the exploration process which lasts on average for more than 6 hours, parallel processing is necessary.

Inst.	Branched nodes	Elapsed time (sec)		Nodes/sec		Rate
	$\times 10^6$	CPU LL PB&B	PB&B@CPU	CPU LL PB&B	PB&B@CPU	
Ta021	41.4	24489	23889	1691	1734	1.03
Ta022	22.1	11758	11450	1877	1927	1.03
Ta023	140.8	79322	77298	1776	1822	1.03
Ta024	40.1	19753	19367	2028	2069	1.02
Ta025	41.4	25332	24661	1636	1680	1.03
Ta026	71.4	34562	33722	2065	2117	1.02
Ta027	57.1	28295	27535	2018	2074	1.03
Ta028	8.1	4569	4440	1770	1822	1.03
Ta029	6.8	3674	3583	1845	1892	1.03
Ta030	1.6	898	873	1835	1888	1.03
Avg	**43.1**	**23265**	**22682**	**1854**	**1902**	**1.03**

Table 5.3: Number of branched permutations during the resolution of Taillard's instances *Ta021–Ta030* initialized with the optimal cost (in millions of permutations) and sequential execution times using LL-based and IVM-based pool management.

5.4.2 Pool management evaluation

Inst.	Random 1/2			Honest 1/2			Largest 1/2			Ring 1/2			Ring 1/T		
	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	LL	IVM	Rate
Ta021	1275	121	10.6	1304	121	10.8	1276	121	10.6	1284	1727	0.7	3189	121	26.3
Ta022	704	67	10.5	706	67	10.5	682	67	10.1	657	1675	0.4	1393	68	20.5
Ta023	4700	399	11.8	4442	400	11.1	4620	400	11.5	4645	10117	0.5	9009	401	22.5
Ta024	1248	111	11.2	1256	111	11.3	1290	111	11.6	1247	1714	0.4	2529	112	22.6
Ta025	1197	115	10.5	1176	115	10.3	1192	115	10.4	1216	6419	0.2	1965	116	16.9
Ta026	2177	203	10.7	2175	203	10.7	2169	203	10.7	2160	2431	0.9	4498	204	22.1
Ta027	1894	156	12.1	1934	156	12.4	1953	156	12.5	1917	5926	0.3	3556	156	22.7
Ta028	262	24	11.1	261	24	11.0	263	24	11.1	261	1556	0.2	600	25	24.5
Ta029	216	20	11.0	219	20	11.2	220	19	11.3	225	1415	0.2	396	21	19.0
Ta030	54	5	11.1	55	5	10.8	53	5	10.9	54	374	0.1	118	6	20.0
Avg	**1373**	**122**	**11.3**	**1353**	**122**	**11.1**	**1372**	**122**	**11.2**	**1367**	**3487**	**0.4**	**2725**	**123**	**22.2**

Table 5.4: Comparison of CPU LL PB&B and PB&B@CPU in terms of time spent managing the pool of permutations (in seconds).

Inst.	Random 1/2			Honest 1/2			Largest 1/2			Ring 1/2			Ring 1/T		
	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	LL	IVM	Rate
Ta021	14.0	14.3	1.02	14.0	14.3	1.02	14.0	14.3	1.02	14.0	13.0	0.93	11.0	14.3	1.30
Ta022	13.7	14.0	1.02	13.7	14.1	1.03	13.8	14.1	1.03	13.8	11.7	0.85	11.4	14.1	1.21
Ta023	13.8	14.1	1.02	13.8	14.0	1.02	13.8	14.0	1.02	13.8	12.1	0.88	11.6	14.1	1.22
Ta024	13.6	13.9	1.03	13.6	13.9	1.03	13.6	13.9	1.02	13.6	11.7	0.86	11.0	13.9	1.27
Ta025	14.2	14.4	1.01	14.2	14.4	1.02	14.2	14.4	1.02	14.1	10.9	0.77	12.9	14.4	1.12
Ta026	13.7	14.0	1.02	13.7	14.0	1.02	13.7	14.0	1.02	13.7	12.9	0.94	11.1	14.0	1.27
Ta027	13.6	13.7	1.01	13.5	13.8	1.02	13.5	13.7	1.02	13.5	11.0	0.82	11.2	13.7	1.22
Ta028	13.9	14.2	1.02	13.9	14.1	1.02	13.9	14.2	1.02	13.8	10.0	0.72	11.1	14.0	1.26
Ta029	13.8	14.1	1.02	13.8	14.1	1.02	13.8	14.0	1.02	13.7	9.7	0.70	12.0	14.0	1.17
Ta030	13.6	13.8	1.02	13.6	13.6	1.00	13.7	13.5	0.98	13.7	9.7	0.71	11.2	13.4	1.19
Avg	**13.8**	**14.0**	**1.02**	**13.8**	**14.0**	**1.02**	**13.8**	**14.0**	**1.02**	**13.8**	**11.3**	**0.82**	**11.4**	**14.0**	**1.2**

Table 5.5: Comparison of multi-core IVM-based and linked-list based parallel PB&B algorithms in terms of speed-up over their sequential counterparts.

In our experiments five work stealing strategies are evaluated, *Ring-1/T*, *Ring-1/2*, *Random-1/2*, *Largest-1/2* and *Honest-1/2*. As explained in Subsection 5.3.4 the steal-Tth policy is theoretically not appropriate for the random, largest and honest victim selection policies. For each of the ten instances *Ta021–Ta030* Table 5.4 shows the time spent managing the pool of permutations. The time spent managing the pool is the cumulated time spent outside of bound computation. For each work stealing strategy, the table shows the obtained results for the LL-based approach, the IVM-based approach, and the ratio between these two approaches. For example when solving instance *Ta021* using the *Random-1/2* strategy the LL-based approach spends 1275 seconds managing the linked-lists of permutations, while the IVM-based approach spends 121 seconds managing the IVM structure, which makes the IVM-based pool management 10.6 times faster than the LL-based pool management.

Except for the *Ring-1/2* strategy the IVM-based approach performs on average at least 11 times better than its LL-based counterpart. While the *Ring-1/2* is clearly inappropriate for the PB&B@CPU, this strategy shows good performances, comparable with *Random*, for the LL-based approach. Conversely, the steal-Tth granularity policy leads to good results for the IVM-based approach while it is not well-suited to the LL-based approach. Table 5.5 shows the speed-up over the respective serial algorithm for the different work stealing strategies and both approaches. Our PB&B@CPU algorithm reaches a reasonable average speed-up of up to about $14\times$ using 16 threads. For work stealing strategies *Largest*, *Honest*, *Random* the PB&B@CPU solves the ten flowshop instances on average 14.0 times faster than the serial PB&B@CORE, while the LL-based multi-core algorithm yields a slightly inferior average speed-up of 13.8 over its serial counterpart, so the PB&B@CPU shows a slightly better scalability. Although the use of the IVM structure allows a significant reduction of the pool management time (for most strategies) this has only a modest impact on the overall execution time. For all strategies the total execution time for the IVM-based approach is only $4-5\%$ faster than its LL-based counterpart (comparing *Ring-1/T* for IVM with *Ring-1/2* for LL). This is due to the predominant cost of the bounding operation.

5.4.3 Interval-based work stealing evaluation

In this subsection we report more detailed experimental results comparing the five proposed work stealing strategies. For each strategy and both approaches, CPU LL PB&B and PB&B@CPU, Table 5.6 shows the time spent by threads waiting for new work units. The waiting time is the cumulated time, for all threads, that elapses between the moment an idle thread starts searching for a victim and the reception of a non-empty interval or set of nodes, i.e. a successful steal attempt. These measured time intervals completely cover the time a victim thread is answering the request. The double of this waiting time is therefore an upper bound for the time spent outside of useful work, i.e. waiting and communication time.

For example, using the *Random* strategy and solving instance *Ta021*, LL-based threads wait 4.67 seconds for new nodes, which is 15.9 times longer than IVM-based threads, which wait 0.29 seconds for new intervals. For all instances and both approaches, the *Largest* strategy results in the lowest waiting time, between 0.10 and 0.17 seconds. The *Random* and *Honest* strategies result in longer waiting times which, however, remain below < 0.5 seconds for all tested instances. For all these three strategies the average waiting time for the LL approach is at least 10 times higher. When using the ring topology, if one wants to keep the waiting time low, then the granularity policy must be chosen in accordance with the data structure used for the pool management. For instance, using the *Ring-1/2* strategy in combination with the IVM approach threads wait on average for 3360 seconds, which corresponds to $\approx 15\%$ of the of the average serial execution time. Similarly, using the *Ring-1/T* strategy in a LL-based PB&B algorithm results in an average waiting time of 1176 seconds and over 2 million work stealing operations on average. These important waiting times explain the high pool management times reported for these strategies in Table 5.4. Choosing the unsuitable granularity policy for the ring topology, i.e. Steal-1/2 for IVM and Steal-T^{th} for LL, limits the average speed-up over the serial algorithm to 11.3, respectively 11.4.

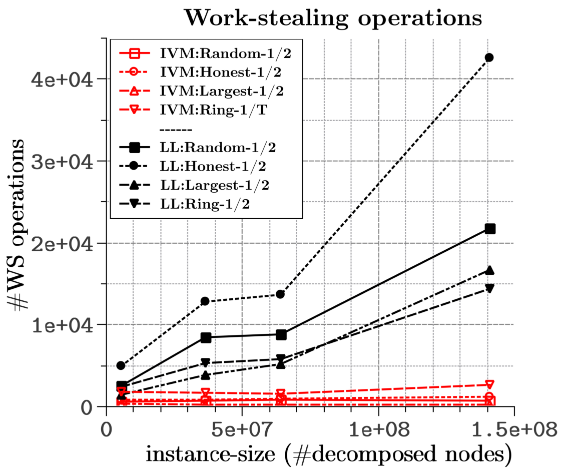
It can be seen from Table 5.6 that the waiting time for the IVM-based approach is much less sensitive to the instance's size and shape than for the LL-based approach. To illustrate this, Figure 5.3 shows the number of performed work stealing operations and the time spent waiting for new work units as a function of the instance size. In order to analyze the behavior of the proposed work stealing strategies according to varying tree sizes, the figure shows the averaged values for groups of instances in which the number of branched nodes is similar: *Ta028–Ta030* are considered as a group of small instances, *Ta021*, *Ta022*, *Ta024* and *Ta025* as medium, *Ta026* and *Ta027* as large, and finally *Ta023* as a very large instance. Figure 5.3 does not show the unsuitable strategies, i.e. *Ring-1/2* using IVM and *Ring-1/T* using LL. In this figure the experimental results for the IVM-based approach using strategies *Random*, *Honest* and *Largest* are barely distinguishable because these values are very small compared to the LL-based approach for large instances.

Figure 5.3 illustrates an interesting feature of the IVM-based work stealing strategies. The resolution of the largest instance *Ta023* requires less work stealing operations and induces less waiting time than the smallest instance *Ta030*, even though its exploration lasts 80 times as long. While the LL-based PB&B@CPU algorithm shows a strong variation in the number of work stealing operations according to the size of the instance being solved, this correlation is apparently very weak or even absent in the IVM-based algorithm.

The largest policy is also the most costly selection policy in terms of time spent per victim selection (including the time spent in synchronization on shared data). Selecting a victim with the largest policy takes more than twice as much time as with the honest, and almost 10 times as much as with the random selection policy. However, the total time measured for victim selection with the largest policy is 13 ms against 3 ms for the random policy, which corresponds to about 10% of the waiting time for *Largest*, respectively 1% of the waiting time for *random*. As a percentage of the total elapsed time this cost for the victim selection is certainly negligible – however, if the thread-count is increased the cost of the *Largest* and *Honest* strategies may become important, due to contention on the shared data structures accessed by the victim selection function.

Inst.	Random 1/2			Honest 1/2			Largest 1/2			Ring 1/2			Ring 1/T		
	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate	CPU LL PB&B	PB&B@CPU	Rate
Ta021	4.67	0.29	15.9	8.95	0.35	25.5	2.84	0.12	24.4	3.98	1610	<0.01	1677	1.11	>100
Ta022	1.89	0.29	6.8	3.69	0.32	11.4	1.07	0.17	6.4	1.49	1609	<0.01	616	1.20	>100
Ta023	7.87	0.30	26.0	15.78	0.44	35.5	6.09	0.13	46.1	6.33	9699	<0.01	3850	1.63	>100
Ta024	2.68	0.25	10.7	3.83	0.28	13.7	0.97	0.16	6.2	3.04	3114	<0.01	1076	0.70	>100
Ta025	3.39	0.33	10.4	3.83	0.34	11.3	1.18	0.13	9.1	3.01	6288	<0.01	670	1.69	>100
Ta026	3.08	0.26	11.9	5.77	0.30	19.0	1.81	0.10	17.3	3.20	2234	<0.01	1983	1.34	>100
Ta027	2.68	0.31	8.7	4.25	0.37	11.4	1.76	0.11	15.8	2.33	5758	<0.01	1365	0.83	>100
Ta028	1.34	0.22	6.2	2.87	0.40	7.1	0.88	0.12	7.2	1.49	1529	<0.01	303	1.02	>100
Ta029	0.85	0.25	3.4	1.60	0.27	5.9	0.48	0.13	3.8	1.51	1392	<0.01	157	1.56	>100
Ta030	0.56	0.25	2.3	1.42	0.39	3.6	0.42	0.13	3.2	1.51	368	<0.01	57	1.25	45.8
Aver.	**2.90**	**0.27**	**10.6**	**5.20**	**0.35**	**14.9**	**1.75**	**0.13**	**13.9**	**2.79**	**3360**	**<0.01**	**1176**	**1.23**	**>100**

Table 5.6: Comparison of multi-core IVM and LL-based parallel PB&B algorithms in terms of time spent waiting for work units; cumulated waiting time for 16 threads (in seconds)



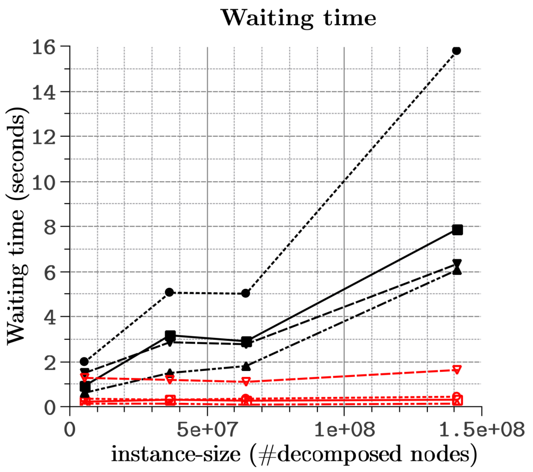


Figure 5.3: Number of work stealing operations and waiting time for the CPU LL PB&B and PB&B@CPU using different work stealing strategies.

5.5 Conclusions

In this chapter, we propose a new approach to manage the pools of permutations of a parallel PB&B algorithm using work stealing parallel strategies. Our new PB&B approach aims at reducing the number of work stealing operations and the total execution time of the algorithm.

To the best of our knowledge, all other parallel PB&B algorithms published in the literature are based on conventional structures, like linked-lists for the depth first strategy, to store the permutations of each thread-private pool. Our new approach is based on the factorial number system, which is a mixed radix numeral system adapted to numbering permutations, and the use of an IVM structure to manage the permutations of the private pools.

A work stealing strategy can be defined by its victim selection policy and its granularity policy. The victim selection policy indicates the thread victim that a thread can steal, and the granularity policy determines the amount of work stolen by the thread. In this chapter, five work stealing strategies are defined for the CPU LL PB&B and the PB&B@CPU algorithm. These strategies are *Ring 1/T*, *Ring 1/2*, *Random 1/2*, *Largest 1/2*, and *Honest 1/2*.

The obtained results, on 10 flowshop instances, demonstrate that our new IVM-based approach outperforms the conventional linked-list based approach in terms of total elapsed CPU time and performed work stealing operations. Also, compared to the linked-list based approach, the IVM-based approach allows to reduce the idle time of workers. We have shown that, especially for large instances, the IVM-based work stealing performs much less work stealing operations than the LL-based approach..

Moreover, the IVM-based approach, contrary to the linked-list based approach, has constant and controllable memory requirements. The memory usage pattern is a huge advantage for many-core architectures, such as GPUs, where the performance depends a lot on memory usage and memory access patterns. For 20-job instances, more than 100 IVM structures fit easily into 48KB of shared memory available per multiprocessor on most devices. Although the implementation of a deque, a stack, etc. is theoretically feasible in CUDA, this type of data structure would necessarily reside in global memory, which is much slower than shared memory. The introduction of Unified Memory since CUDA 6 is likely to make the use of linked-lists on the GPU more practical. However, fitting into the fastest part of the hierarchical GPU memory, the IVM is clearly the better adapted data structure for the GPU architecture. The next chapter introduces the PB&B@GPU approach based on the use of IVM and interval concepts on GPU accelerators.

CHAPTER 6

GPU-centric permutation Branch-and-Bound

Contents

6.1 Introduction

In this chapter, the PB&B@CPU algorithm is revisited for GPUs. In contrast to the other existing GPU-accelerated B&B algorithms, this chapter deals with a GPU-centric implementation of PB&B, meaning that it completely bypasses the CPU. To the best of our knowledge, our PB&B@GPU algorithm is the first to perform all PB&B operations on the GPU device. The key idea to achieving this is to use the IVM structure for pool management, which is much better suited for GPUs than dynamic LL data structure. In addition, the use of intervals, as work units, makes work stealing strategies on GPUs more efficient. As shown in Figure 6.1, the PB&B@GPU approach is very similar to the PB&B@CPU approach. However, PB&B@GPU differs from PB&B@CPU in several aspects:

- The number of PB&B threads of PB&B@GPU is much higher than the number of PB&B threads of PB&B@CPU.

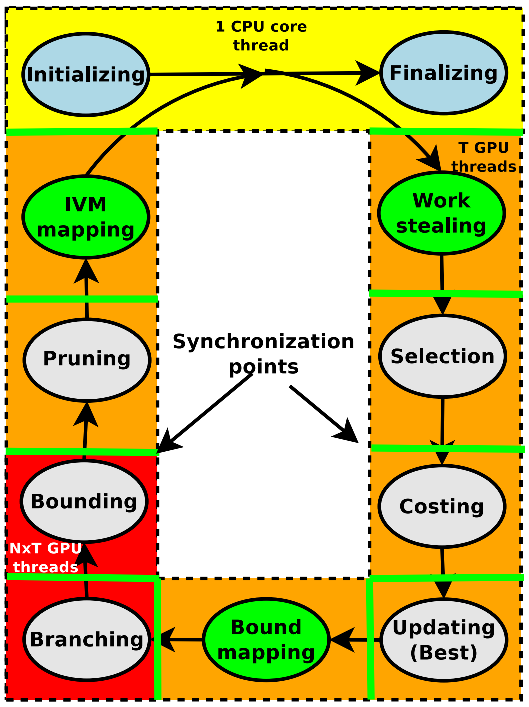


Figure 6.1: Overview of the PB&B@GPU approach and its operations.

- The work stealing operation of PB&B@GPU is different from the work stealing operation of PB&B@CPU; Subsection 6.3.3 describes the work stealing operation of the PB&B@GPU.

- In addition to the other operations, the PB&B@GPU approach also uses a supplementary operation called a bound mapping and IVM mapping operations; Subsection 6.3.1 and Subsection 6.3.2 present respectively these bound mapping and IVM mapping operations.

- Unlike the bounding operation of PB&B@CPU, where this operation calculates the set of bounds of all the sub-permutations of a partial permutation, the bounding operation of PB&B@GPU calculates the bound of a single sub-permutation. Therefore, for each sub-permutation, it is necessary to launch up to N GPU threads, where N is the size of the permutation problem. Each of these threads calculates the bound of a single sub-permutation. In a PB&B@GPU approach with T PB&B threads, it is possible to run up to $N \times T$ threads for a single bounding operation.

- In PB&B@CPU, the operations of the different PB&B threads are run asynchronously. In other words, in this approach, a PB&B thread may run the branching operation at the same time when another PB&B thread runs the bounding operation. On the other hand, in the PB&B@GPU approach, the operations of different PB&B threads are executed synchronously. It means that, if a PB&B thread of PB&B@GPU executes a certain operation, then all the other threads perform the same operation. For example, if a PB&B thread runs the bounding operation, then all the other threads run this operation. The synchronization between all the operations is represented in Figure 6.1 by the synchronization points, drawn in green.

This chapter is divided into three main sections. In addition to Section 6.2 and Section 6.3, which respectively describe the data structures and the operations of PB&B@GPU, Section 6.4 presents the experiments performed to validate PB&B@GPU approach.

6.2 IVM-based Data structures

This section focuses on the data structure of our PB&B@GPU based on IVM. The memory requirements of the IVM structure are very advantageous for a GPU implementation of the PB&B algorithm. The required amount of memory and possible data placements in the hierarchical device memory are discussed in Subsection 6.2.1. The amount of used memory depends on the number of IVM structures used by the algorithm. This number also has a direct impact on the degree of parallelism which is analyzed in Subsection 6.2.2.

6.2.1 Memory requirements

Compared to a conventional LL-based approach, the IVM structure allows to reduce the CPU time and memory required for the storage and management of the pool of permutations [MLMT14b]. Contrary to LL, the IVM structure is well adapted to the GPU memory model. Instead of using a variable length queue that requires dynamic memory allocations and tends to be scattered in memory, the IVM structures are constant in size and need only one allocation of contiguous memory. For a problem instance with N elements, the storage of the matrix M requires N^2 bytes of memory (for $N < 127$, using 1-byte integers). Moreover $3N$ bytes are needed to store the position, end and branching vectors, 1 byte to store the depth, and N bytes to store permutations before calling the bounding operation. In total, the IVM structure requires a constant amount of $1 + 4N + N^2$ bytes of memory, i.e.

481 bytes per IVM for a 20-element instance. It is possible to store only the upper triangular part of M, requiring $1 + 4N + \frac{N(N+1)}{2}$ bytes per IVM, i.e. 291 bytes when $N = 20$. For $N = 20$ it is therefore possible to fit ≥ 100 IVM structures into 48KB of shared memory. In our PB&B@GPU, only the upper triangular part of M is stored.

From a programming perspective the IVM structures are easy to handle. The components of all IVMs are merged into single one dimensional arrays. For instance, solving a N-element instance using T IVM structures, the matrices are stored in a one dimensional array `matrices` of size $T \times \frac{N(N+1)}{2}$ allocated in global device memory. The element $M(i, j)$ of the kth IVM is accessed by `matrices[indexM(i,j,k)]`, where `indexM` is a wrapper function defined as in Equation (6.1) if M is stored as a square and as in Equation (6.2) if the upper triangular part of M is stored.

$$indexM(i, j, k) = k \times N \times N + i \times N + j \tag{6.1}$$

$$indexM(i, j, k) = k \times \frac{N(N+1)}{2} + i \times N - \frac{i(i-1)}{2} + j \tag{6.2}$$

For the flowshop problem, the data needed for the computation of the lower bounds is mostly read-only and require 34.5KB of memory. These data are stored in the constant memory space, residing in global device memory but accessed through a cache on each Streaming Multiprocessor (SMX). Some of the data structures used for the bounding may be loaded to shared memory during the computation of the lower bounds. Concerning the use of shared memory for those data structures, this chapter follows the recommendations made in [Cha13], where this difficult choice is examined.

6.2.2 Memory constraint: Number of IVMs

Often, the bounding operation is by far the most time-consuming part of a PB&B algorithm. In the case of flowshop it amounts for about $97 - 99\%$ [MCB14] of the total execution time for a sequential PB&B. It is therefore crucial for the performance of our PB&B@GPU that the parallel bounding operation makes the best use of the GPU resources. The choice of the number of PB&B processes (=IVMs) to use is therefore guided by its impact on the performance of the bounding kernel. On the one hand, if too few IVMs participate in the exploration process, the bounding kernel under-utilizes the GPU. On the other hand, if too many IVMs are used, then the number of generated permutations per iteration exceeds the maximum occupancy of the device and the computation of bounds is partially serialized. The number of permutations generated per IVM per iteration is variable and unpredictable in shape and size. However, the workload for the bounding kernel can be roughly estimated. For flowshop instances of 20 jobs, the bulk of permutations is situated at depth 10, leading to approximately 20 bound evaluations per IVM and iteration. Supposing that the number of empty IVMs is low thanks to dynamic load balancing, and given the approximative number of $20,000$ concurrent threads at full occupancy, the number T of used IVMs should be around $T = 1000$.

6.3 Operations

This section introduces the two mapping operations and the work stealing operation.

6.3.1 Bound mapping operation

The shape of the tree explored by a PB&B algorithm is highly irregular and unpredictable, resulting in an irregular workload, irregular control flow and irregular memory access pattern. If not addressed properly, these irregularities may cause a low occupancy of the device, serialized execution of instructions and poor bandwidth usage due to uncoalesced memory accesses. Both, the application's memory access pattern and the divergent behavior of threads depend strongly on the chosen mapping of threads onto the data. When a GPU application runs, each streaming multiprocessor is assigned one or more thread block(s) to execute. Those threads are partitioned into groups of 32 threads[1], called *warps*, which are scheduled for execution. CUDA's single instruction multiple-thread (SIMT) execution model assumes that a warp executes one common instruction at a time. Consequently, full efficiency is realized when all 32 threads of a warp agree on their execution path. However, if threads of a warp diverge due to a data dependent conditional branch, the warp serially executes each branch path taken. Threads that are not on that path are disabled, and when all paths complete, the threads converge back to the same execution path. This phenomenon is called *thread divergence* and often causes serious performance degradations. In a very similar way, if the threads in a warp agree on the location of a requested piece of data, it may be fetched in single cycle, otherwise serialization of the data accesses occurs. In this subsection the focus is put on reducing thread divergence and increasing warp execution efficiency by making judicious mapping choices.

Static mapping method

The most straightforward approach probably consists in mapping each thread onto a child permutation directly from its `threadId`. This naive and static approach is shown in Algorithm 6. For instance, launching $2 \times N \times \#IVM$ threads (line 2), the first $N \times \#IVM$

[1]We assume using the GK110 model

threads place unscheduled elements in the beginning, the second $N \times \#IVM$ threads in the end. Regardless of the IVM's state or current depth in the tree, $2 \times N$ threads are reserved for each IVM. Each thread is assigned an IVM to work on and an element to schedule, like shown in lines 4-6 of Algorithm 6. The approach of Algorithm 6 has several disadvantages. The *if*-conditionals in line mask many of the launched threads, precisely $2 \times k$ threads per father permutation of depth k, plus $2N$ threads per empty IVM. Moreover, different lanes in the same warp work on different IVMs, thus thread divergence occurs due to different values of L_1 and L_2. If $T \times N$ is a multiple of *warp-size*, then the *if-else* conditional (lines 10 and 14) does not cause any thread divergence.

Algorithm 6 Kernel `STATIC-BOUND-MAPPING`
IN: Fathers (father,L_1,L_2)
OUT: Lower bounds begin, lower bounds end, sums of lower bounds

```
1:  kernel NAIVE-BOUND
2:      <<< 2 × $\#elements \times \#IVM$ threads>>>
3:      thId←blockIdx.x*blockDim.x + threadIdx.x
4:      if (state[ivm] == not-empty) then
5:          if ($L_1$[ivm] < element < $L_2$[ivm]) then
6:              if (dir == 0) then
7:                  swap( schedule, $L_1$[ivm]+1, element )
8:                  LB-begin[ivm][element]←computeLB( schedule )
9:                  sum-begin[ivm] += LB-begin[ivm][element]
10:             else if (dir == 1) then
11:                 swap( schedule, $L_2$[ivm]-1, element )
12:                 LB-end[ivm][element]←computeLB( schedule )
13:                 sum-end[ivm] += LB-end[ivm][element]
14:             end if
15:         end if
16:     end if
17: end kernel
```

Remapping method (First step)

Algorithm 7 describes how to build the maps *ivm-map* and *element-map* sequentially. However, sequential execution of this procedure on the device has prohibitive cost, exceeding 25% of the total execution time. The remapping should therefore be built in parallel. The parallelization of the outer *for*-loop (Algorithm 7, line 3) is not straightforward, because it is unknown at which location the data for each IVM is to be written to. Computing the *prefix-sum* of a vector containing the number of elements to be scheduled per IVM allows its parallelization.

Algorithm 7 Build mapping (serial)

```
1: kernel SERIAL PREPARE BOUND
2:     running-index ← 0
3:     for (ivm = 0 → T)) do
4:         if (state[ivm] = not-empty) then
5:             for (element = $L_1$[ivm] + 1 → $L_2$[ivm]) do
6:                 ivm-map[running-index] ← ivm
7:                 element-map[running-index] ← element
8:                 running-index++
9:             end for
10:        end if
11:    end for
12:    todo ← running-index
13: end kernel
```

Algorithm 8 STEP1-REMAPPED-BOUND-MAPPING

```
1: for all (non-empty ivm) do
2:     todo-per[ivm] ← ($L_2$[ivm]-$L_1$[ivm]-1)
3: end for
4: Aux ← parallel-prefix-sum(todo-per)
5: prepare-bound<<< $\#IVM \times \#ELEMENTS$ >>>
6: kernel [KERNEL] PREPARE-BOUND
7:     thId ← blockIdx.x*blockDim.x + threadIdx.x
8:     ivm ← thId / N
9:     thPos ← thId % N
10:    if (thPos < todo-per[ivm]) then
11:        ivm-map[Aux[ivm]+thPos] ← ivm
12:        element-map[Aux[ivm]+thPos] ← $L_1$[ivm]+1+thPos
13:    end if
14:    todo←Aux[#IVM]+todo-per[#IVM]
15: end kernel
```

The operation *prefix-sum* is defined as

$$prefix-sum: \quad [a_0 \quad a_1 \quad a_2 \quad \dots \quad a_n] \longmapsto [0 \quad a_0 \quad (a_0+a_1) \quad (a_0+a_1+a_2) \quad \dots \quad \sum_{i=0}^{n-1} a_i].$$

Efficient parallel CUDA implementations for this operation have been proposed in the literature [HSO07]. It is also available in the CUDA Thrust library. However, for relatively small vectors it may be preferable to reimplement the operation, in order to avoid casting the input data to a `thrust::device_ptr`.
A first building step consists in filling an array `todo-per-IVM` with $L_2 - L_1 - 1$ for each IVM. The element R of `prefix-sum(todo-per-IVM)` indicates at which position of `ivm-map` and `element-map` the data of an IVM R starts to be written. The complete parallelized building of the mapping is shown in Algorithm 8. The building of the mapping ranges over several kernels.

Remapping method (Second step)

Algorithm 9 Kernel `STEP2-REMAPPED-BOUND-MAPPING`
in:fathers (father,L_1,L_2), ivm-map, element-map
out:lower bounds begin, lower bounds end, sums of lower bounds

```
1:  kernel REMAPPED-BOUND
2:      <<< 2 × todo threads>>>
3:      thId←blockIdx.x*blockDim.x + threadIdx.x
4:      dir←thId mod 2
5:      ivm←ivm-map[thId/2]
6:      element←element-map[thId/2]
7:      schedule←fathers[ivm]
8:      toSwap←(1-dir)*($L_1$[ivm]+1) + dir*($L_2$[ivm]-1)
9:      swap( schedule, toSwap, element )
10:     LB[dir][ivm][element]←computeLB( schedule )
11:     sum[dir][ivm] += LB[dir][ivm][element]
12: end kernel
```

The goal of the remapping procedure which prepares the bounding is to build two maps *ivm-map* and *element-map* which contain, for *todo* threads, the information which IVM to work on and which element to swap. Using an even/odd pattern these maps provide sufficient information for both groups of threads. After building these maps, the bounding kernel (as shown in Algorithm 9) is called with $2 \times todo$ threads, where:

- threads 0 and 1 work on IVM *ivm-map[0]*, swapping element *element-map[0]* respectively to begin/end,
- threads 2 and 3 work on IVM *ivm-map[1]*, swapping element *element-map[1]* respectively to begin/end,
- ...
- threads $2\times$todo-2 and $2\times$todo-1 work on IVM *ivm-map[todo-1]*,...

The remapped bounding kernel is launched at each iteration with a kernel configuration of $(2 * todo/blockDim) + 1$ blocks (simplified in Algorithm 9) which is adapted to the workload. The proposed approach is known as *stream compaction* in the literature. It reduces the number of idle lanes per warp as well as the number of threads launched per kernel invocation. However, any thread divergence resulting from the begin-end distinction should also be avoided, as this involves a serialization of the costly `computeLB` procedure. To achieve this, the bodies of the *if-else* conditional (Algorithm 6) can be merged into a single one (Algorithm 9, lines $8-11$). Two different arguments of the same type, occurring on the right-hand side of a statement can often be re-factored into a single one, like in Algorithm 9, line 8. The different arrays on the left hand side are merged into larger ones. This allows to merge the statements of lines 8,9 and 12,13 of Algorithm 6 into single statements (Algorithm 9, lines 10,11). The separation of data within these merged arrays is assured by indexing with the variable `dir`, which evaluates differently for even/odd threads.

6.3.2 IVM mapping operation

The IVM management kernels (i.e work stealing, branching, selection and pruning) require a single thread per IVM. The naive approach consists in launching T threads and mapping thread k on IVM k, for $k = 0, 1, ..., T-1$ (see Algorithm 10). Given the high number of conditional instructions in the IVM management kernels it is very unlikely that all 32 threads in a warp follow the same execution path if this mapping is used. Indeed, in these kernels control flow divergence results from different IVM states, different numbers of scheduled elements at both ends of the active permutation and from the search for the next node which requires an unknown number of iterations. An alternative mapping, shown in Algorithm 11, can solve this issue. An entire warp is assigned to each IVM, so all threads belonging to the same warp follow the same execution path. This strategy goes in the opposite direction of the stream compaction approach proposed for the bounding kernel. As only one thread per IVM is needed, all lanes in a warp except this first are masked. Thus, the kernels are launched with $32\times$ as many threads as necessary (i.e. $32\times$ T). Using this mapping, the overhead induced by thread divergence completely disappears (although technically, the disabled threads are diverging at line 5 of Algorithm 11). The drawback is obviously the launching of 31T idle threads. However, in Subsection 6.2.2 we argued that T should be chosen around $T = 1000$, which is small compared to #SM$\times$(max. threads per SM). This, and the fact that the control flow irregularity is very high, justifies the approach of using 1 warp per IVM. Moreover, using only 4-8 IVM structures per block allows to store them into shared memory without limiting the theoretical occupancy of the device. The loading of data from global to shared memory can be done very efficiently, using the additional threads which are not used for computation.

Algorithm 10 Kernel THREAD-IVM-MAPPING

```
1: kernel<<<#IVM threads>>>
2: ivm←blockIdx.x*blockDim.x + threadIdx.x
3: do-something-with[ivm]
```

Algorithm 11 Kernel WARP-IVM-MAPPING

```
1: kernel<<<warpsize×#IVM threads>>>
2: thId←blockIdx.x*blockDim.x + threadIdx.x
3: ivm←thId/32
4: thPos←thId%32
5: if (thPos == 0) then
6:     do-something-with[ivm]
7: end if
```

6.3.3 Work stealing operation on GPU

Work stealing is well adapted for irregular applications. Like threads of a PB&B@CPU, the IVM structures must share their work units. In a multi-core CPU environment, a thread that runs out of work becomes a thief that attempts to steal a portion of work from a victim thread which is selected according to a victim selection strategy. The same principle can be applied to the PB&B@GPU. The proposed load balancing strategy is conceptually different in the sense that an IVM-based PB&B process does not necessarily correspond to any particular thread but only to a segment of data. Secondly, compared to PB&B@CPU work stealing operation, the work stealing PB&B@GPU operations between IVMs are lock-free and performed synchronously. The kernel implements the 1D-Ring work stealing strategy. Algorithm 12 shows the pseudo-code of this procedure. Although designed for PB&B@CPU, the 1D-Ring strategy suits the synchronous execution mode of the GPU. The T IVM structures are numbered $R = 0, 1, ..., T - 1$ and are arranged as an oriented ring, i.e. such that IVM 0 is IVM $(T - 1)$'s successor. Each empty IVM R tries to steal work from its predecessor $(R - 1)\%T$. This operation can be performed in parallel, as the mapping of empty IVMs onto their respective victims is one-to-one. If the selected victim has a non-empty interval, then all but $1/T^{th}$ of its interval is stolen. The function `computeNewPos` (line 6) receives the victim's interval $[A, B[$ as input and returns a point $M = (1 - \frac{1}{T})A + \frac{1}{T}B$. The division of intervals can be performed directly on the factoradic numbers without explicitly converting them to decimals. The IVM which got stolen continues the exploration of the remaining interval $[A, M[$, while the stealing IVM needs to initialize its matrix at the new position vector M before starting the exploration of $[M, B[$. Its state variable is therefore set to *initializing* (line 10). Each IVM cycles through three distinct states, from *exploring* to *empty* to *initializing* and back to *exploring*. An IVM can be in one of these three states at any given stage of the algorithm. Depending on the state of an IVM, different actions are performed during an iteration. In this kernel one thread per IVM is required. More parallelism can hardly be exposed. However, more threads can eventually be used to assign vectors in one parallel operation (lines $7 - 9$).

Algorithm 12 Kernel `WORK-STEALING`

```
 1: kernel WORK-STEALING
 2:     thId ← blockIdx.x*blockDim.x + threadIdx.x
 3:     ivm ← map(thId)
 4:     victim ← (ivm-1)%T
 5:     if (state[ivm]=empty .and. state[victim]=exploring) then
 6:         new-pos← computeNewPos( pos[victim], end[victim] )
 7:         pos[ivm]← new-pos
 8:         end[ivm]← end[victim]
 9:         end[victim]← new-pos
10:         state[ivm]← initializing
11:     end if
12: end kernel
```

The topology used in the work stealing strategy, already described, is a unidirectional 1-dimensional ring (1D-Ring). The maximal distance between two IVMs in the 1D-Ring is T. Work units propagate through the ring as they are passed downstream from exploring to empty IVMs. As most of the explored PB&B nodes are actually contained in a relatively small interval, the workload tends to be concentrated in some part of the ring. Thus, workers situated far away from the source are only kept busy if the overall workload is large enough. With an increasing number T of IVM structures it becomes more likely that no work is dripping down to some of the workers. A topology that reduces the maximum distance between two workers should therefore improve the scaling with T.

The 1D-Ring can be easily generalized to a 2D-Ring, or torus, topology. Instead of using a single ring, IVMs are arranged in R rings of ring-size $C = T/R$. In a first step each empty IVM attempts to steal from its left neighbor within the same ring. A second step connects the rings between each other: each empty IVM selects the IVM with the corresponding ID in the preceding ring (with ring $R-1$ being connected to ring 0). The roles played by both directions are symmetric. Ideally, the number R is therefore such that $R = \sqrt{T}$, which is only possible if T is square. In that case the 2D-Ring reduces the maximum distance between two IVMs to $2\sqrt{T}$. If $C \neq R$, then the diameter of the 2D-Ring is $(C + R)$.

The 2D-Ring topology is implemented by two subsequent calls of kernel `work stealing` (Algorithm 12), where only line 4 of the algorithm needs to be modified. In particular, line 4 of Algorithm 12 is replaced by the following.

$$\text{In Step 1 IVM } i \text{ selects } victim(i) = \begin{cases} i-1, & \text{if } i \; mod \; C \neq 0 \\ i+(C-1), & \text{otherwise} \end{cases}$$

$$\text{In Step 2 IVM } i \text{ selects } victim(i) = \begin{cases} i-C, & \text{if } i > (C-1) \\ (R-1)C+i, & \text{otherwise} \end{cases}$$

Figure 6.2 illustrates the 2D-Ring topology in the form of a 2D grid. A torus, used in the 2D-Ring work stealing strategy is obtained by connecting the upper with the lower and the leftmost with the rightmost cells. Similarly, the topology can be extended to a hypercube, which is used for instance in [SKK+11] for unbalanced tree search.

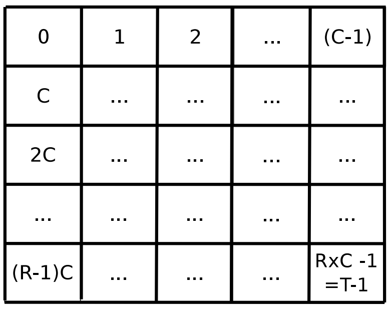


Figure 6.2: Illustration of 2D-Ring topology for T IVMs using R rings of ring size C.

6.4 Experiments

In this section the performance of PB&B@GPU is analyzed for different mapping choices, a varying number of IVMs and different work stealing strategies. In Subsection 6.4.2, the two strategies for the bound mapping operation are evaluated. Subsection 6.4.3 compares the two mapping strategies for the IVM mapping operation. The algorithm's scalability and load balancing issues are examined in Subsection 6.4.4. Finally, our PB&B@GPU algorithm is compared to the GPU LL PB&B algorithm presented in [CM13].

6.4.1 Experimental protocol

- **Hardware testbed:** All the experiments are run on a computer equipped with an NVIDIA Tesla K20m GPU based on the GK110 architecture. The device is composed of 2496 CUDA cores (clock speed 705MHz). Its maximum power consumption is 225W. The CPU is a 8-core Sandy Bridge E5-2650 processor.

- **Software tools:** Version 6.5.14 of the CUDA Toolkit is used. The operating system installed is a CentOS 6.5 Linux distribution. For the evaluation of the elapsed execution time the UNIX `time` command is used. The duration of each CUDA kernel and profiling of the kernels is done with the `nvprof` command line profiler. In order to reduce the profiling time, sample data was collected every 100 iterations of the algorithm.

- **Configuration choices:** The chosen size for the thread blocks is 128. The configurable size of the device's shared memory/L1 cache is set to $48/16$KB for kernels except `bound`, where the opposite configuration $16/48$KB is used. For the comparison of the mapping strategies the number of used IVM structures is set to $T = 768$, according to preliminary experiments. The best mapping found in Subsection 6.4.2 and Subsection 6.4.3 is used to determine an optimal value for T and the better work stealing strategy in Subsection 6.4.4.

- **Problem instances:** In our experiments, the validation is performed using the 10 instances where the number of machines and the number of jobs are equal to 20 which belong to the group $20x20$. When an instance is solved twice using a PB&B performing a parallel tree exploration, the number of explored permutations is often different between the two resolutions, because the order of exploration varies. To compare the performance of two PB&B algorithms, the number of explored permutations should be exactly the same between the different tests. Therefore, we choose to always initialize our PB&B by the optimal complete permutation of the instance to be solved. This initialization ensures that the tree-shape does not depend on the decrease of the best complete permutation found so far and that the number of explored permutations is the same between the two resolutions. Table 6.1 shows the number of branched nodes during the resolution of instances *Ta021-Ta030* initialized with the optimal complete permutation. This number represents the total amount of work to be done and ranges from 1.6 (for the smallest) to 140.8 million nodes (for the largest) instance.

Instance	Ta021	Ta022	Ta023	Ta024	Ta025	Ta026	Ta027	Ta028	Ta029	Ta030	Average
#Nodes (in millions)	41.4	22.1	140.8	40.1	41.4	71.4	57.1	8.1	6.8	1.6	**43.1**

Table 6.1: Number of branched permutations during the resolution of Taillard's instances *Ta021-Ta030* initialized with the optimal cost (in millions of nodes).

6.4.2 Evaluation of bound mapping schemes

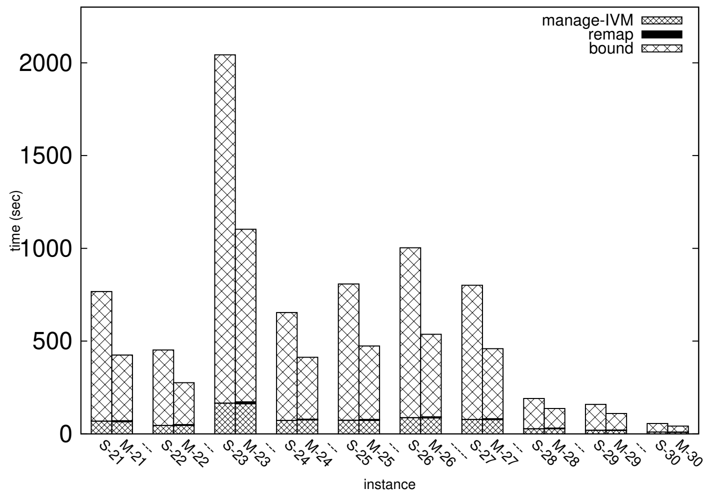


Figure 6.3: Execution time for instances *Ta021-Ta030* for thread data mappings *static* (S) and *remap* (M) for the kernel `bound`.

In this subsection the two mapping schemes for the bounding kernel, presented in Subsection 6.3.1, are compared to each other in terms of elapsed execution time of the algorithm. The first, using the remapping shown in Algorithm 9 is referred to as *remap*, the second, using the static mapping of Algorithm 6, as *static*. Figure 6.3 shows the total elapsed time for solving instances *Ta021-Ta030*. For both mappings and for each instance it shows the portion of time spent in the kernel bounding, in the IVM management kernels (i.e work stealing, branching, selection and pruning) as well as in the remapping operation (for *remapping*). However, as the building of the mapping consumes only 0.9% of computation time, the latter portion is barely visible in Figure 6.3. Table 6.2 shows total elapsed time as well as the time spent in the different operations of the algorithm as an average over the 10 instances *Ta021-Ta030*.

Bound mapping method	**Wall-clock time**	**bounding operation**		**IVM management operations**		**Bound mapping operation**	
	(sec)	**(sec)**	**(%)**	**(sec)**	**(%)**	**(sec)**	**(%)**
Static	696.4	632.9	89.4	63.5	10.6	0.0	0.0
Remapping	395.7	329.1	82.0	63.4	17.1	3.4	0.9

Table 6.2: Average elapsed time (in seconds) and average repartition of execution time among bounding, IVM management and remapping operations. Average taken over instances *Ta021-Ta030*.

The compacted mapping *remap* is clearly advantageous as it reduces the average time spent in the `bound` kernel by a factor 1.9. As the bounding operation amounts for more than 80% of the total execution time, the latter decreases by a factor 1.7. The overhead induced by compacting the mapping at each iteration is largely compensated by these performance gains. Indeed, thanks to the parallelization of this operation using the parallel prefix sum, the remapping operation amounts for less than 1% of the elapsed time. For comparison, using the CPU for the remapping, it amounts for about 7% of the algorithm's total execution time, mainly because of the transfer of the maps back to the device.

Using the more compact mapping *remap* instead of *static* improves the control flow efficiency[2] (CFE) of the kernel. For *static* the average CFE is 0.43, meaning that for an executed instruction on average more than half of the execution slots are wasted. For the mapping *remap* the average CFE is 0.83 - the launched warps are used almost twice as efficiently. The number of warps launched at each kernel call is 960 for mapping *static*, which exceeds theoretical maximum of $13 \times 64 = 832$ resident warps for the K20m. The average number of warps launched with mapping *remap* is 300 (average per kernel call and per instance), the average maximum (per instance) being 825 warps and the minimum 4. These results show that it is a high priority optimization to adapt the configuration of the algorithm's most time-depending part to the varying workload.

[2]defined as $CFE = \frac{\texttt{not\_predicated\_off\_thread\_inst\_executed}}{\texttt{32*inst\_executed}}$

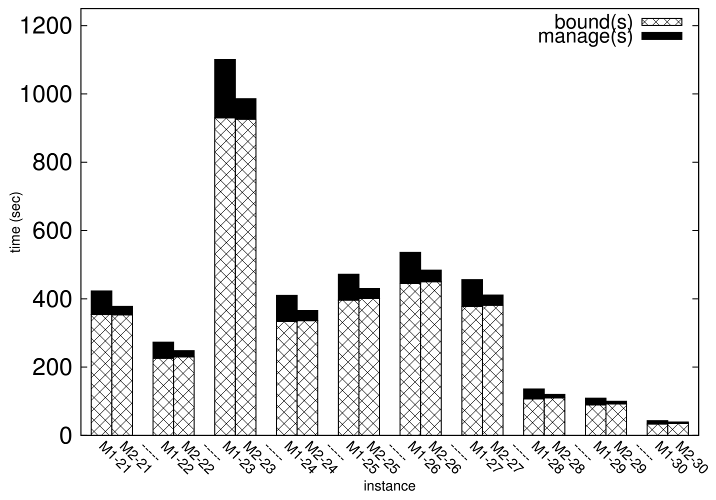


Figure 6.4: Execution time for instances *Ta021-Ta030* for different mapping choices in IVM management kernels.

6.4.3 Evaluation of IVM mapping schemes

In this subsection the two mapping schemes presented in Subsection 6.3.2 are evaluated and compared to each other. The kernels concerned by these mapping schemes are the IVM management kernels (i.e work stealing, branching, selection and pruning). Figure 6.4 shows the time spent for completing the exploration with both mapping schemes. Both, version *one-thread-per-IVM* ($M1$) and version *one-warp-per-IVM* ($M2$) use the same bounding kernel (with remapping). Although the time spent managing the IVM structures is moderate compared to the bounding operation, the mapping $M2$ allows a reduction of the total execution time by a factor 1.1 compared to the mapping $M1$. With respect to $M1$, mapping $M2$ decreases the share of IVM management operations from 18% to 7.5%. Table 6.3 shows the average duration per call of the kernels `bound` (in msec), `branching` and `selection` (in μsec) and their respective share of the elapsed time (in %). The kernels `pruning` and `work stealing` amount for at less than 2% of total execution time, so they are not evaluated.

The mapping $M2$ allows to use the supplementary lanes for efficient loading of the IVM structures into shared memory. In order to dissociate the impact of shared memory usage from the impact of remapping, the profiling of mapping $M2$ is performed with and without shared memory usage.

IVM mapping method	Branching operation			Selection operation			Bounding operation		Wall-clock time
	(μsec)	(%)	(IRO%)	(μsec)	(%)	(IRO%)	(μsec)	(%)	(sec)
1 thread/IVM	380	10.0	40.6	168	4.4	40.3	3.07	82.0	395.7
1 warp/IVM	130	4.0	14.0	94	2.8	14.7	3.07	91.1	364.2
1 warp/IVM (shared)	85	2.6	7.9	79	2.4	12.4	3.06	92.5	356.6

Table 6.3: Duration of different kernels per call (in μsec or msec), percentage of total elapsed time (%) and instruction replay overhead (IRO%), total execution time of PB&B@GPU. Average values for instances *Ta021-Ta030*.

Table 6.3 also shows the instruction replay overhead (IRO%)[3], which is a measure for instruction serialization (due to memory operations only). These results show that the fact of spacing the mapping to 1 warp=1 IVM also substantially improves the memory access pattern. It should be noted that the metric *control flow efficiency*, used in Subsection 6.4.2 drops from a poor average 0.22 for $M1$ to $0.03 \approx 1/32$ for $M2$ - as intended. Table 6.4 shows, for the different kernels, the number of branch instructions executed (per call average) and the number of branches that are evaluated differently across a warp. The results show that, as intended, undesired thread divergence completely disappears. Only instance *Ta022* is evaluated as one instance sufficiently illustrates the behavior.

IVM mapping method	Branching operation		Selection operation		Work stealing operation		Pruning operation	
	Branch	Diverge	Branch	Diverge	Branch	Diverge	Branch	Diverge
1 thread/IVM (M1)	11592	802	5875	860	851	15	404	121
1 warp/IVM (M2)	59921	1536 =2×#IVM	62020	768 =#IVM	3655	0	3131	768 =#IVM

Table 6.4: Per-call average of branch instructions executed and diverging branches (incremented by one per branch evaluated differently across a warp). Instance *Ta022*.

The `divergent_branch` counter indicates that the average number of diverging branches is a multiple of the number of IVMs. Indeed, the counter increments by one at the instruction `if(thId%32 == 0)` (Algorithm 11, line 5) which masks all but the leading thread in each warp. However, as the remaining 31 lanes of the warp are simply waiting for lane 0 to complete, no significant serialization of instructions occurs. Besides showing that the spaced mapping $M2$ is better adapted to the IVM management kernels, the results presented in this subsection illustrate that performance metrics for thread divergence or control flow must be interpreted very carefully.

[3]defined as $IRO\% = 100\% \times \frac{\texttt{instructions\_issued}-\texttt{instructions\_executed}}{\texttt{instructions\_issued}}$

6.4.4 Evaluation of work stealing schemes

In this subsection the behavior of the algorithm according to the instance sizes and its scalability with the number of used IVM structures (T) is examined. The algorithm's performance for problem instances of different sizes is compared in terms of node processing speed (#branched nodes/second), which is computed from wall-clock time. Obviously, using more explorers can only be beneficial if they can be supplied with enough work. Therefore, the relationship between instance size, the node processing speed and T needs to be studied for both proposed work stealing strategies, as they strongly impact this relationship. For the experimental study of scalability only the best version of the previous subsection is considered, i.e. the one using parallel remapping for the bounding kernel and the spaced mapping $M2$ for the management kernels. Another factor that has a significant impact on the algorithm's performance is the irregularity of the explored PB&B tree, which is very hard to quantify. In order to obtain a clearer dissociation between tree size and tree irregularity, the average node processing speed for instances of similar size is considered. The instances have been grouped as shown in Table 6.5.

Instance group	**#branched nodes**	**Flowshop Instances**	**Average #branched nodes**
small	$< 10\text{M}$	*Ta028,Ta029,Ta030*	5.5M
medium	$\in [10\text{M}, 50\text{M}]$	*Ta021,Ta022,Ta024,Ta025*	36.3M
large	$\in [50\text{M}, 100\text{M}]$	*Ta026,Ta027*	64.3M
huge	$> 100\text{M}$	*Ta023*	140.8M

Table 6.5: Groups of similar sized flowshop instances (20 jobs $\times$ 20 machines) and the corresponding average number of nodes branched when initialized with the optimal complete permutation.

Figure 6.5 shows the average node processing speed for these four groups of instances, according to different values of T and work stealing strategies 1D-Ring and 2D-Ring. T is chosen as a multiple of 64, as the GK110 architecture allows up to 64 warps per SM and the management kernels reserve one warp per IVM. The number of rings R in the 2D-Ring work stealing strategy is chosen such that R divides T while being as close to $\sqrt{T}$ as possible – the goal being to approach the ideal configuration where R equals the ring-size C.

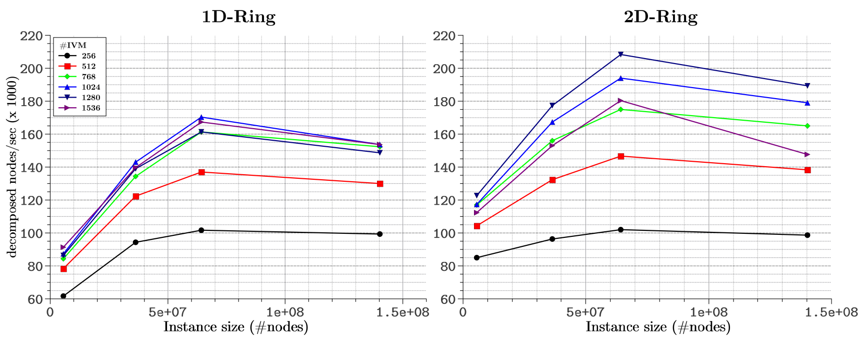


Figure 6.5: Average node processing speed for groups of instances *small*, *medium*, *large*, *huge* (Table 6.5), and $T = k \cdot 256$ $(k = 1, 2, ..., 6)$.
Left: 1D-Ring work stealing ; *Right*: 2D-Ring work stealing with dimensions $R \times C$ $= 16 \times 16$, 16×32, 24×32, 32×32, 32×40 and 32×48

From Figure 6.5 one can see that the node processing speed for *small* instances is lower than for intermediate sized instances, regardless of parameter T or the used work stealing strategy. This is partially due to warm-up and shut-down phases of the parallel exploration, which, for *small* instances last relatively long with respect to the total exploration time. In these phases a low overall workload limits the degree of parallelism and the work stealing mechanism must handle sharp variations of the workload. Using the 2D-Ring topology significantly improves the nodes-per-second rate for *small* instances, as the reduced diameter of the 2D-Ring accelerates the distribution of the workload. Another reason for the poorer performance for *small* instances is that the finer granularity of allocated tasks per IVM does not allow one to hide the initialization costs as efficiently as for larger instances. Similarly, the nodes-per-second performance for *medium* instances is below the performance for *large* and *huge* instances. As the groups *large* and *huge* do only count one, respectively two members, the results for these groups must be interpreted more carefully – however, they suggest that the node processing speed for large or very large instances is less influenced by the tree's size. The size of the instance also impacts the scalability of the algorithm with T, especially when the 2D-Ring work stealing strategy is used. Using the 2D-Ring topology, for *small* sized problems the best value for T improves performance to ≈ 120k nodes/sec from ≈ 85k nodes/sec for the worst T, while for *large* sized problems the node processing speed doubles from ≈ 105 to ≈ 210k nodes/sec, doing the same comparison.

Comparing both work stealing strategies, one can see from Figure 6.5 that the 2D-Ring topology improves the scalability of the algorithm. Indeed, for the 1D-Ring almost no performance is gained above $T = 768$ because additional workers are left idle or inefficiently initializing. In contrast, the 2D-Ring strategy allows one to use up to $T = 1280$ IVMs efficiently. For $T = 1536$, and only when using the 2D-Ring work stealing strategy, performance drops significantly for all instance sizes. The most likely explanation for this performance drop is that the computation of the bounds is partially serialized due to hardware limitations. On average, each node branching leads to the evaluation of approximately 20 bounds, for all considered instance sizes. So, if all $T = 1280$ IVMs are busy, 25600 bounds are evaluated per average iteration, which is close to the hardware limit of 26624 concurrent threads (13 SM $\times 64$ warps/SM $\times 32$ threads/warp) for the GK110 architecture. Therefore, supposing a well balanced workload, no performance improvement can be expected from increasing T beyond 1280 on this device and at constant tree size.

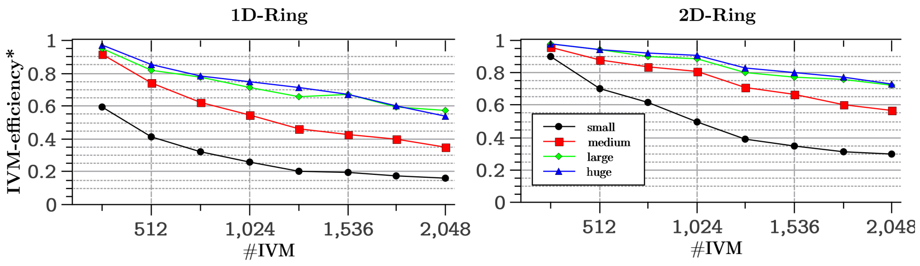


Figure 6.6: *IVM-efficiency=$\frac{\text{branched nodes}}{\text{iterations}\times\#\text{IVM}}$, measures the overall share of IVM structures in the *exploring* state. An IVM efficiency of 1 indicates that each IVM branches a node at each iteration.

However, Figure 6.6 shows that there is some margin left to increase the percentage of busy explorers. In order to measure the overall efficiency of the work stealing strategies, the number of branched nodes is divided by (#iterations $\times$ #IVM), which gives a measure for the portion of IVMs doing meaningful work. If this *IVM efficiency* equals 1, then each IVM branches a node at each iteration – if it equals 0, then all IVMs remain idle for an infinity of iterations. An *IVM efficiency* of 0.5 indicates that an average IVM spends half of its iterations idle or initializing. The results shown in Figure 6.6 confirm that the 2D-Ring work stealing allows to keep more explorers busy. Moreover, the fact that at $T = 1536$ no sharp degradation of the IVM efficiency occurs, confirms that the performance drop at $T = 1536$ is due to hardware limitations. One can also see in Figure 6.6 that for *small* and *medium* instances the IVM efficiency is lower than for larger instances. This explains the relatively poorer performances for these instances. The smaller the instance being solved, the likelier it becomes that a given IVM frequently steals intervals that are explored within a few or zero iterations. In that case, the cost of initialization is not covered by meaningful work. Without any *a priori* knowledge concerning the amount of work contained in a given interval, it seems difficult to resolve this problem.

Concerning the algorithm's sensitivity towards the irregularity of the tree structure, the performances obtained when solving same sized instances are compared in Figure 6.7.

Figure 6.7 zooms on the scalability with T of the three instances with ≈ 40M nodes. For all three instances the performance drop increasing the number of IVMs from $T = 1280$ to $T = 1536$ is clearly visible.

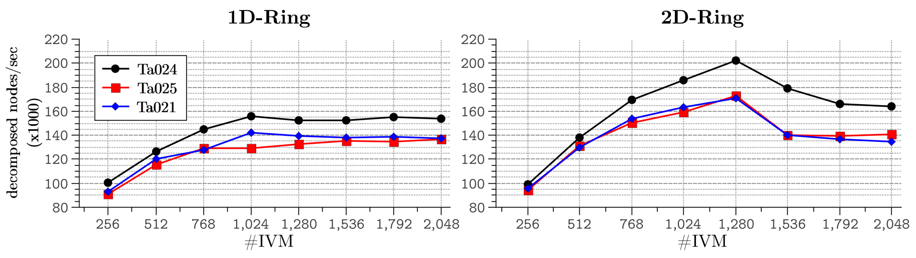


Figure 6.7: Scaling with T of the medium sized instances

Figure 6.7 also shows that the instance irregularity impacts performance significantly. Exploring a roughly equal number of nodes, the peak node processing speed attained for instance *Ta024* is 202k nodes/sec while it remains below 175k nodes/sec for instances *Ta021* and *Ta025*. However, in sequential execution *Ta024* has the best nodes-per-second rate of these three instances, which means that the performance gap is not due to a higher number of nodes in the upper part of the tree (which are more costly to evaluate). The performance variety under same sized instances rather seems to be due to sharper workload variations for instances *Ta021* and *Ta025*. Although for the three instances the proposed 2D-Ring strategy scales nicely up to 1280 IVM structures, it apparently has a moment of inertia which makes the algorithm sensitive to rapid workload variations. Maybe this gap can be closed with a further improved work stealing strategy, in the sense that the work stealing strategy should deliver a faster response to those variations. This is a difficult task, in particular because IVM needs to go through initialization after the reception of a new work unit.

6.4.5 Comparison of PB&B@GPU and *GPU LL PB&B*

In this subsection the best version (according to the previous experimentations) of the PB&B@GPU is compared to the GPU-accelerated PB&B presented in [CM13] which uses a conventional LL for the storage and management of the pool of permutations. Like the PB&B@GPU, GPU LL performs a depth best-first search, retaining after each branching the better of two generated pools. Moreover, both algorithms use the same device function to compute the lower bounds for a given permutation. Table 6.6 shows the elapsed time for solving each of the ten 20-job instances, as well as the number of branched nodes per second, in order to take into account the size of the instances. The average time spent by the PB&B@GPU for exploring the ten instances is 229.7 seconds, while the GPU

LL PB&B algorithm requires on average 865.6 seconds for the same tasks. In terms of node processing speed, the PB&B@GPU algorithm branches on average 3.3 times more nodes per second than its LL counterpart. The PB&B@GPU outperforms GPU LL PB&B by at least a factor 2 for all ten instances. The highest nodes-per-second performance is attained for instance *Ta027*, branching $213,000$ nodes/sec, which is almost 4 times more than the highest rate attained by the LL-based algorithm. As examined in Subsection 6.4.4, for the PB&B@GPU algorithm the node processing speed varies from one instance to another, depending on the instance's size and irregularity. In contrast, as the GPU LL PB&B algorithm regularizes the workload by dynamically adapting the size of the offloaded pools it provides an almost constant node processing speed for all instances. This performance variation is discussed in Subsection 6.4.4.

Flowshop	**GPU LL PB&B**		**PB&B@GPU**		**Ratio**
instance	**elapsed**	**(k nodes/sec)**	**elapsed**	**(k nodes/sec)**	
Ta021	833	49.7	242	171.1	3.4
Ta022	415	53.3	134	163.6	3.1
Ta023	3089	45.6	740	189.6	4.2
Ta024	738	54.3	200	202.2	3.7
Ta025	865	47.9	239	173.1	3.6
Ta026	1292	55.3	348	205.6	3.7
Ta027	1094	52.2	268	213.3	4.1
Ta028	171	47.4	53	152.0	3.2
Ta029	125	54.4	56	121.4	2.2
Ta030	34	47.1	17	94.6	2.0
Average	**865.6**	**50.7**	**229.7**	**168.6**	**3.3**

Table 6.6: Elapsed execution time and number of branched nodes per second (in 1000 nodes/sec) for the PB&B@GPU and the GPU LL PB&B [CM13]. The PB&B@GPU algorithm uses the 2D-Ring work stealing strategy and $T = 1280$ IVMs for all instances. Results for instances *Ta021-Ta030*.

6.5 Conclusions

This chapter proposes a GPU-centric PB&B algorithm which performs all PB&B operations on the GPU. During the exploration of the PB&B tree, the CPU core is only used for launching the CUDA kernels in a loop until a boolean variable, which the CPU receives at each iteration from the GPU, indicates the end of the algorithm. To the best of our knowledge, our PB&B@GPU algorithm is the first one that does not rely on the transfer of pools of permutations between host and device.

The proposed approach is based on the IVM structure, better adapted to the GPU than

LL-based data structures, which are conventionally used for the storage and management of the pool of permutations. The algorithm has two levels of parallelism. On a lower level it efficiently uses up to 1280 IVM structures to perform the branching, selection and pruning operations in parallel, exploring different parts of the PB&B tree simultaneously. For each exploring IVM the bounding operation is in turn parallelized, leading to an increased overall degree of parallelism in the bounding operation. At the junction of the two levels a remapping operation is introduced in order to adapt the configuration and the mapping of the bounding kernel to the varying workload. While the mapping for the bounding is compacted, the mapping in the management phase – characterized by a very high number of data dependent conditional instructions – is spaced, adding idle threads to the kernel. As a result, the bounding operation is accelerated on the one hand, as the control flow efficiency is improved – on the other hand, in the IVM management phase the opposed strategy is better adapted, as it reduces thread divergence and improves memory accesses.

We have proposed two work stealing strategies for work load balancing and analyzed the scalability of our algorithm with respect to both strategies. The reported experimental results show that the performance of the proposed algorithm depends crucially on the choice of the mapping, as well as on the used work stealing strategy. The proposed PB&B@GPU algorithm explores the PB&B trees of 10 Taillard's flowshop instances *Ta021-Ta030* on average with a 3.3 times higher node processing speed than a GPU LL PB&B algorithm. For all instances *Ta021-Ta030* the PB&B@GPU algorithm outperforms the GPU LL PB&B algorithm at least by a factor 2.0 and by up to a factor 4.0 on large instances, where our PB&B@GPU algorithm reaches its best performances.

CHAPTER 7

Permutation B&B for GPU-powered clusters

Contents

7.1 Introduction

This chapter presents a parallel distributed hybrid version of the PB&B algorithm (PB&B@CLUSTER) for large-scale heterogeneous clusters, integrating multi-core CPU and graphics processing units (GPUs). PB&B@GPU is based on the B&B@Grid framework [MMT07c]. B&B@Grid allows to efficiently partition the PB&B tree search among distributed computing nodes, which host one or several workers. Each worker explores a portion of the search space (an interval) using a sequential PB&B. Thus, while compute nodes in B&B@Grid may be composed of multi-core processors, the latter are seen as a collection of single-core processors. Therefore, B&B@Grid is revisited with the goal of adapting it to heterogeneous computing platforms.

This chapter is divided into three main sections. Section 7.2 defines a set of operators used to handle interval-lists. Section 7.3 presents the operations of the coordinator. Section 7.4 describes the experiments carried out to validate the PB&B@CLUSTER approach.

7.2 Interval-list operators

In our approach, there are two types of operators, namely the interval-list and interval operators. The interval-list operators are based on interval operators. Before presenting the interval-list operators, this section first describes the interval operators.

7.2.1 Interval operators

Like the PB&B@CPU and PB&B@GPU approaches, a work unit, in the PB&B@CLUSTER approach, is also an interval, and each interval corresponds to an IVM. An interval can be seen as a sorted set of positive factorial integers. In order to define our approach, we have introduced some operators on these intervals. As an interval is a set of integers, these operators are inspired by operators known in set theory. The interval operators are therefore the intersection, the subtraction, the right and left divisions, the cardinality and the norm. Let's assume:

- $\mathbb{I}$: interval space;
- $\mathbb{L}$: interval-list space;
- $\mathbb{N}$ and $\mathbb{N}^+$: respectively the space of integers and strictly positive integers;
- $[a, b[$: an interval;
- $[a_1, a_2[$ and $[b_1, b_2[$, two other intervals;
- n: a positive integer;

The interval operators and the empty interval are defined as follows:

- Intersection operator ($\cap$):

$$\mathbb{I} \times \mathbb{I} \to \mathbb{I}$$
$$[a_1, a_2[\cap [b_1, b_2[\mapsto [max(a_1, b_1), min(a_2, b_2)[$$

- Substraction operator ($\backslash$):

$$\mathbb{I} \times \mathbb{I} \to \mathbb{L}$$
$$[a_1, a_2[\backslash [b_1, b_2[\mapsto \{[a_1, min(a_2, b_1)[, [max(a_1, b_2), a_2[\}$$

- Right division operator ($\div$):

$$\mathbb{I} \times \mathbb{N}^+ \rightarrow \mathbb{I}$$

$$[a, b[\div n \mapsto \begin{cases} [a, b[, & \text{if } (b-a) \leq \epsilon \\ [b - \lfloor (b-a)\frac{1}{n} \rceil, b[, & \text{if } (b-a) > \epsilon \\ 0, & \text{otherwise} \end{cases}$$

- Left division operator ($\div$):

$$\mathbb{N}^+ \times \mathbb{I} \rightarrow \mathbb{I}$$

$$n \div [a, b[\mapsto \begin{cases} [a, b[, & \text{if } (b-a) \leq \epsilon \\ [a, a + \lfloor \frac{n-1}{n}(b-a) \rceil [, & \text{if } (b-a) > \epsilon \\ 0, & \text{otherwise} \end{cases}$$

- Cardinality operator ($|.|$):

$$\mathbb{I} \rightarrow \mathbb{N}$$

$$|[a_1, a_2[| \mapsto \begin{cases} 1, & \text{if } a_1 < a_2 \\ 0, & \text{otherwise} \end{cases}$$

- Norm operator ($\|.\|$):

$$\mathbb{I} \rightarrow \mathbb{N}$$

$$\|[a_1, a_2[\| \mapsto \begin{cases} a_2 - a_1, & \text{if } a_1 < a_2 \\ 0, & \text{otherwise} \end{cases}$$

- Emptiness value ($\emptyset$):

$$([a, b[= \emptyset) \iff (\|[a, b[\| = 0)$$

7.2.2 Interval-list operators

A series of intervals form an interval-list. In our approach, each worker handles a list of IVMs and, therefore, an interval-list. In the same way, the previous subsection defines some interval operators, this section extends the definition of these operators on interval-lists, and also defines the concept of empty interval-list. Let's assume:

- $A = (A_1, ..., A_I)$: an interval-list of I intervals;
- $B = (B_1, ..., B_J)$: another interval-list of J intervals;
- n: an integer;

Interval-list operators and empty interval-list are defined as follows:

- Intersection operator ($\cap$):

$$\begin{gathered}\mathbb{L} \times \mathbb{L} \to \mathbb{L} \\ A \cap B \mapsto \cup_{i=1,I}^{j=1,J} (A_i \cap B_j)\end{gathered}$$

- Substraction operator ($\setminus$):

$$\begin{gathered}\mathbb{L} \times \mathbb{L} \to \mathbb{L} \\ A \setminus B \mapsto \cup_{i=1,I}^{j=1,J} (A_i \setminus B_j)\end{gathered}$$

- Right division operator ($\div$):

$$\begin{gathered}\mathbb{L} \times \mathbb{N}^+ \to \mathbb{L} \\ (A_1, ..., A_I) \div n \mapsto (A_1 \div n, ..., A_I \div n)\end{gathered}$$

- Left division operator ($\div$):

$$\begin{gathered}\mathbb{N}^+ \times \mathbb{L} \to \mathbb{L} \\ n \div (A_1, ..., A_I) \mapsto (n \div A_1, ..., n \div A_I)\end{gathered}$$

- Cardinality operator ($|.|$):

$$\begin{gathered}\mathbb{L} \to \mathbb{N} \\ |A| \mapsto \sum_{i=1,I} |A_i|\end{gathered}$$

- Norm operator ($\|.\|$):

$$\begin{gathered}\mathbb{L} \to \mathbb{N} \\ \|A\| \mapsto \sum_{i=1,I} \|A_i\|\end{gathered}$$

- Emptiness value ($\emptyset$):

$$(A = \emptyset) \iff (\|A\| = 0)$$

Compared to the previous subsection, this subsection also introduces a new operator called $biggest$. This operator receives, as input, an interval-list A and a natural integer n, and returns, as output, another interval-list $biggest(n, A)$ made up of the n largest intervals of A. The intervals are compared with each other using the $\|.\|$ operator.

- Biggest ($biggest$):

$$\mathbb{N} \times \mathbb{I} \to \mathbb{L}$$

$$biggest(n, A) \mapsto \begin{cases} (A_i)/\forall j, \|A_j\| \leq \|A_i\|, & \text{if } n = 1 \\ \Big(biggest(1, A \setminus biggest(1, A)),\ biggest(n-1, A)\Big), & \text{otherwise} \end{cases}$$

7.3 Data structures and operations

This section describes the PB&B@CLUSTER approach. Subsection 7.1 and Subsection 7.3.2 present respectively the data structures of this approach and its operations.

7.3.1 Data structures

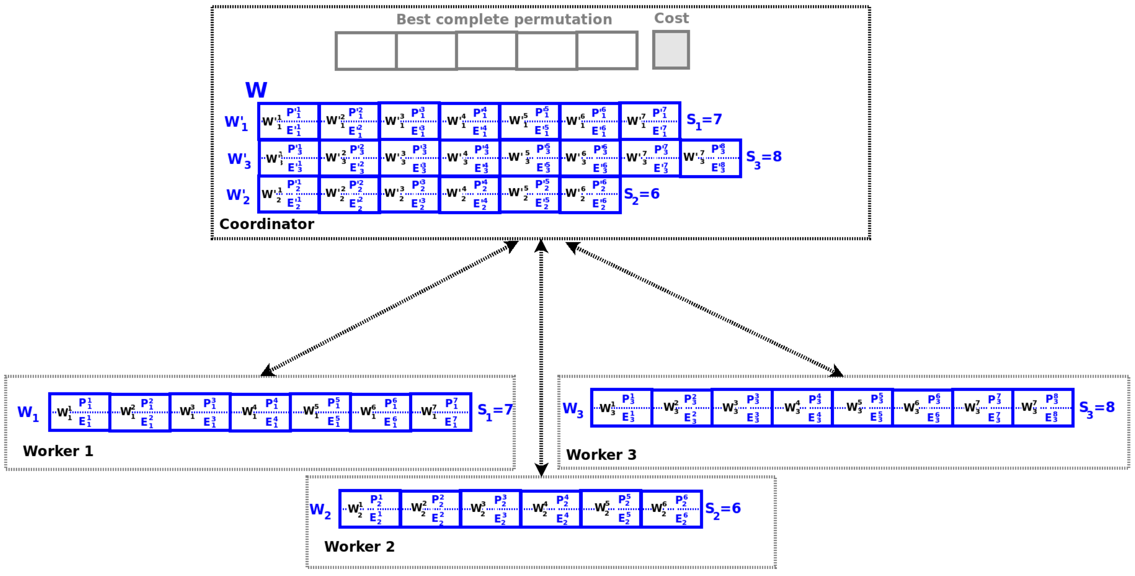


Figure 7.1: The data structure of the coordinator in PB&B@CLUSTER approach.

As shown in Figure 7.1, our approach PB&B@CLUSTER is based on a coordinator-workers architecture. In this architecture, there are a single coordinator process and a large number of workers. The coordinator process has two roles: On the one hand, it allows PB&B workers to exchange the best complete permutation found so far and, on the other hand, this coordinator is used by workers for implementing a work stealing strategy. Compared with the approaches described in previous chapters, the PB&B@CLUSTER approach introduces a new data structure. This structure allows to save the best complete permutation found so far, as well as a copy of the interval-lists being processed on each worker. This data structure is defined as follows:

- **Best found permutation**: This variable saves the best complete permutation found so far and the cost of this permutation.

- **Work of an IVM**: The interval W_i^j is the j^{th} interval of the i^{th} worker. In other words, W_i^j is associated with the j^{th} IVM of the i^{th} worker. This interval W_i^j is equal to $[P_i^j, E_i^j[$ such that:

  - P_i^j : the position vector of IVM j in the worker i;
  - E_i^j : the end vector of IVM j in the worker i;

  A copy of the interval W_i^j, denoted W'^j_i, is always stored at the coordinator. The copy W'^j_i is equal to W_i^j only when the worker sends a request to the coordinator. Between two requests, the copy W'^j_i is probably not equal to the interval W_i^j.

- **Work of worker**: The work of a worker W_i is an interval-list $(W_i^1, ..., W_i^{S_i})$, such that:

  - S_i : #IVM in the worker i
  - W_i^j work of IVM j in the worker i

  For example, the worker 2 in Figure 7.1 has 6 intervals. A copy of the interval-list of worker i, denoted W'_i, is stored at the coordinator. The copy W'_i of the coordinator is equal to W_i only the first time the worker contacts the coordinator to get work.

- **Work of all workers**: W' is the list of all copies $(W'_1, ..., W'_N)$, such that W'_i is the copy of the interval-list W_i of the worker i. For example, the coordinator of Figure 7.1 manages 3 workers. These workers, numbered 1, 2 and 3, respectively handle 7, 6 and 8 IVMs. Therefore, the copy W' has three interval-lists (W'_1, W'_2, W'_3), containing each a list of intervals.

7.3.2 Work processing operations

In addition to the operators explained in the previous chapters, the PB&B@CLUSTER approach essentially introduces three new operations, namely a communication operation of the worker, a checkpointing operation of the coordinator and a work stealing operation of the coordinator. Figure 7.2 shows the relation between these operations on an example of a PB&B@CLUSTER approach using three workers.

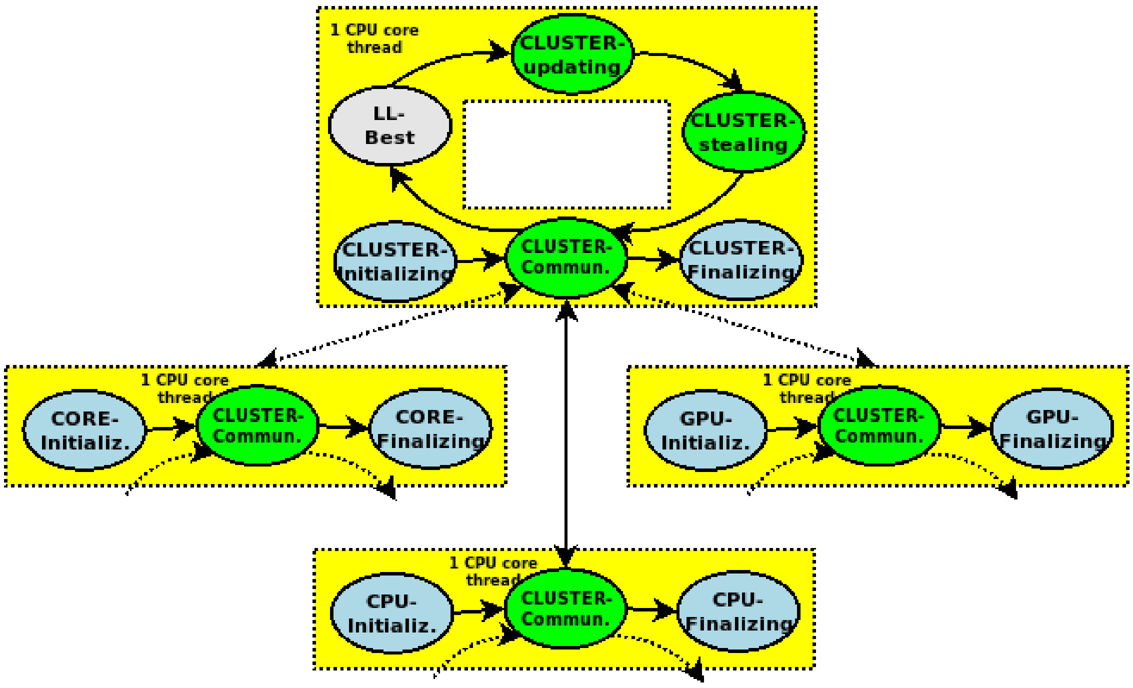


Figure 7.2: Example of a PB&B coordinator with three PB&B workers.

- **Worker communication operation:** The worker i contacts the coordinator when one of the following three conditions occurs:
  - **The rate of empty IVMs is high:** When an IVM of a worker i becomes empty, it is important that the worker, responsible for managing this IVM, operates a work steal to fill the empty IVM. However, if this work stealing operation is done immediately at each empty IVM, then there is a big risk of wasting a lot of time in communication. As a result, the worker i only operates a work stealing when the rate of empty of IVMs $|W_i|/S_i$ is less than a certain threshold α, where $|W_i|$ is the number of empty IVMs, and S_i is the total number of IVMs of the worker i.
  - **New best complete permutation found:** When a new best complete permutation is found by the worker i, this worker communicates to the coordinator this

new permutation. The interest of this communication is to allow the workers to be informed as soon as possible of the new permutation found. This allows these workers to prune more nodes from the PB&B tree. By communicating this new permutation, the worker i communicates its work W_i to the coordinator to update its copy W'_i.

  - **Maximum time limit passed:** After a request to the coordinator, the work W_i of the worker i is equal to the copy W'_i of this work at the coordinator. Between two requests to the coordinator, the more time passes, the larger the gap between W_i and W'_i. Therefore, it is important to configure the worker by a maximum time limit. If one of the two preceding conditions did not occur within this maximum time limit, then the worker i must contact the coordinator to update its copy W'_i. This third condition allows the coordinator to be regularly informed of the progress of the work of the worker i.

- **Work checkpointing operation:** As shown in Algorithm 13, the operation of updating the coordinator's work W' is quite simple. This operation receives as input the work W' of the coordinator, the ID i of the worker and its work W_i. The checkpointing operation performs an intersection of the two lists W_i and W'_i. In the algorithm, the result of this intersection, denoted W_{tmp}, is assigned to the two lists W_i and W'_i.

Algorithm 13 COORDINATOR-WORK-CHECKPOINT(W', i, W_i)

```
1: $W_{tmp} = W_i \cap W'_i$
2: $W'_i = W_{tmp}$
3: $W_i = W_{tmp}$
```

- **Work stealing operation:** Algorithm 14 explains the work stealing operation of the coordinator. As this algorithm shows, this operation is executed only if the interval W_i is empty. In this case, the first instruction is to seek the greatest interval-list, denoted W'_j, of all the works in W'. Once this interval-list is determined, the second instruction calculates the minimum size, denoted S, between the cardinalities of W'_j and W'_i. The cardinality of an interval-list is equal to the number of intervals in this list. The role of the fourth instruction is to take the larger S intervals, denoted W'_{tmp}, from the interval-list W'_j. Then, the intervals of the list W'_{tmp} are divided into 2 using a right division operation. The interval-list, obtained by this division operation, is assigned to the variable W'_i, and the list of intervals $W'_j \setminus W'_i$ is assigned to the variable W'_j. This work stealing operation returns the interval-list W_i to worker i.

Algorithm 14 COORDINATOR-WORK-STEALING(W', i, W_i)

```
1: if $W_i = \emptyset$ then
2:     $j = \arg\max_i(\|W_i\|)$
3:     $S = min(|W_i|, |W_j|)$
4:     $W'_{tmp} = biggest(S, W'_j)$
5:     $W'_i = W'_{tmp} \div 2$
6:     $W'_j = W'_j \setminus W'_i$
7:     $W_i = W'_i$
8: end if
```

7.4 Experiments

This section describes the two experiments performed to validate the PB&B@CLUSTER approach, on a cluster of GPUs and a hybrid cluster of GPUs and CPUs In addition, the section presents the obtained results when solving a large instance.

7.4.1 Experiments on a cluster of GPUs

Experimental protocol

- **Hardware testbed:** The Ouessant cluster is used for this first experiment. This cluster is located at l'Institut du Développement et des Ressources en Informatique Scientifique (IDRIS[1]). The Ouessant Tier1 (national) cluster is composed of 12 computing nodes. Each node consists of 2 CPUs POWER8+ (10 core CPUs, 8 threads per core, so 160 threads per node) and 4 Nvidia GPUs (Generation Pascal P100, and 16 GB of memory). In total, this cluster is composed of about 170,000 cores CPUs and 240 cores CPUs. For reasons of availability, only 9 nodes are used in our experiments.

- **Problem instance:** The ten instances of size 20×20 (20 jobs and 20 machines), noted $Ta021$, $Ta022$, ... and $Ta030$, are used in the previous experiments. Among all instances of this size, the $Ta023$ instance is the most difficult to solve, and requires approximately 22 hours with the PB&B@CORE approach on a single CPU core. Therefore, these instances are not difficult enough for a resolution using a PB&B@CLUSTER on the Ouessant cluster. On the other hand, the ten instances of size 50×20, denoted $Ta051$, $Ta052$, ... and $Ta060$, are difficult for a resolution on the Ouessant cluster. So we used the ten 50×20 instances to generate 50 instances

[1]http://www.idris.fr/

suitable for a resolution on Ouessant. These 50 instances are denoted $Ta0i - h$, with $i \in \{51, 52, ..., 60\}$ and $h \in \{1, 2, ..., 5\}$.

To generate a $Ta0i - h$ instance, the PB&B@CORE approach is deployed with a single worker. This worker solves the $Ta0i$ instance by using a single GPU of Ouessant for h hour (s). In addition, this resolution is initialized with the best known permutation of the instance $Ta0i$. At the beginning of the resolution, the work to be explored is the interval $[0, N![$, with $N = 50$ the size of the instance. At the end of these h hours, it remains a list of intervals L_i^h to continue exploring. Therefore, the interval-list that is already explored is $([0, N![) \setminus L_i^h$. So the instance $Ta0i - h$ is defined as (1) the resolution of the $Ta0i$ instance (2) by initializing the PB&B with the list of intervals $([0, N![) \setminus L_i^h$ and (3) the best known permutation of $Ta0i$.

Obtained results

This section presents the results obtained when solving five instances $Ta0i - h$, with $i \in \{54, 55, 57, 58, 59\}$ and $h \in \{5\}$. Each of these instances is solved using six deployment configurations, namely 1, 4, 8, 16, 24, and 36 GPUs workers. As a result, 30 tests (i.e. $5 * 6$) are performed in total. The metrics measured in this experiment are, for example, percentage of redundantly explored nodes, GPU speedup, number of checkpointing operations, number of work-stealing operations, elapsed-time, etc. For each of these metrics, an average is calculated for each of the six deployment configurations.

- **Redundant nodes:**

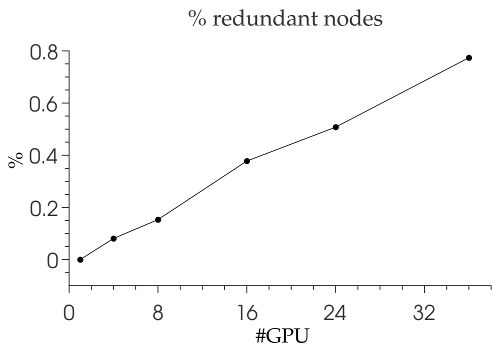


Figure 7.3: Rate of redundant branched nodes.

One of the few disadvantages of the PB&B@CORE approach is that some nodes of the PB&B tree can be explored multiple times. Therefore, it is important to estimate the percentage of nodes explored more than one time. Figure 7.3 shows that with a single GPU worker, no node is explored twice. As the number of GPU workers increases, the percentage of redundant nodes increases. However, the percentage of redundant nodes is on average equal to 0.8% when 36 GPU workers are used. These experiments show therefore that the number of redundant nodes is not significant.

- **Elapsed time and speedup:**

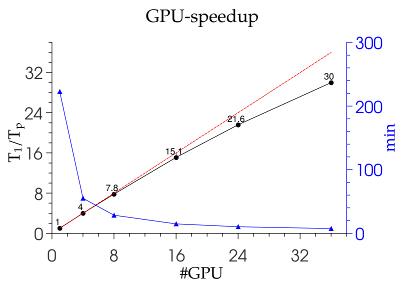


Figure 7.4: Elapsed time and speedup over 1 GPU.

Figure 7.4 shows the elapsed time (in minutes) in blue on the right y-Axis and the speedup with respect to a single GPU in black on the left y-Axis. The red dashed line corresponds to linear speedup with the number of GPUs. The average execution time on a single GPU is about 220 minutes. Using 36 GPUs the average execution time decreases to approximately 7.5 minutes, which corresponds to a relative speedup of $30\times$ over a single GPU. These experiments show that the speedup remains good despite the high used computing power.

- **Work checkpointing and stealing operations:**

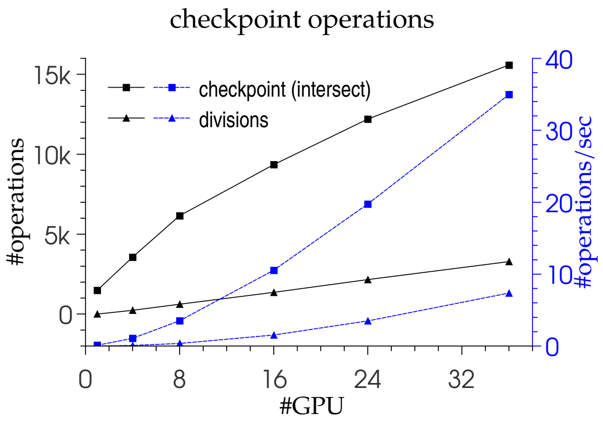


Figure 7.5: Number of checkpointing operations.

As explained previously, the coordinator essentially performs two operations, namely work checkpointing and stealing operations. The objective is to study these two operations. In Figure 7.5 the number of checkpointing operations and work stealing operations is shown in black on the left y-Axis. The number of these two operations per second is shown in blue on the right y-Axis. Naturally, as the number of GPUs increases the number of these two operations also increases. For the considered scale, the rate of increase appears to be constant. A checkpointing operation is counted for both metrics. Therefore the number of checkpointing operations is always greater than the number of work stealing operations. One can see that the number of checkpoints increases faster than the number of work stealing operations. However, using more GPUs does not necessarily increase the number of checkpointing operations. Indeed, if all workers contact the coordinator in fixed and regular intervals, and if the elapsed time decreases linearly with the number of GPUs, then the total number of checkpointing operations would remain constant. In contrast, the results show that workers contact the coordinator more frequently. At near-linear acceleration factors, the rate at which the coordinator performs checkpointing operations increases quadratically. Many checkpointing operations are performed by replacing the coordinator's copy by the current work unit (if the copy wasn't modified remotely).

- **Coordinator operations:**

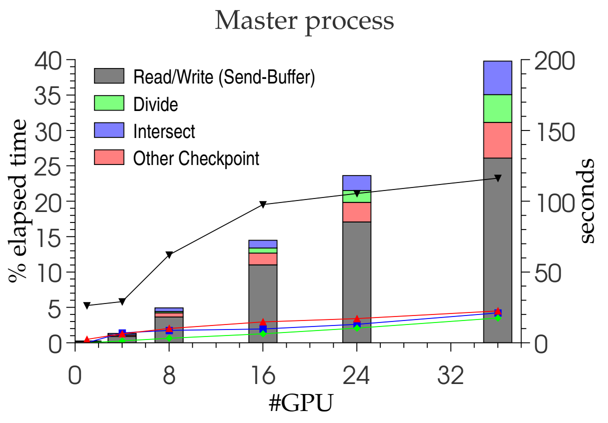
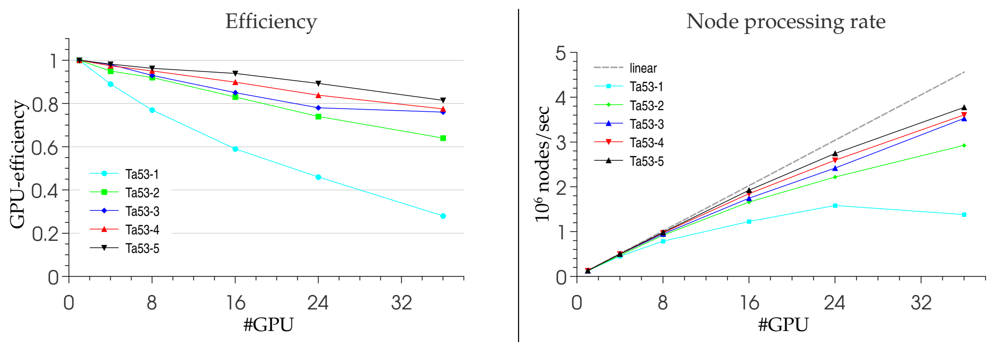


Figure 7.6: Exploitation of coordinator process. Lines show the time (in seconds) and bars the percentage of total elapsed time.

Figure 7.6 focuses on the activity of the coordinator, which is split into four parts measured separately. The absolute time (in seconds) spent in these parts is represented by solid lines. Stacked bars represent this time as a percentage of the total elapsed time. Messages are sent and received in the form of a sequence of characters (*stringstream*), which must be written/read to/from valid work units. This manipulation of the send-buffer consumes 65-90% of the coordinator's processing time (decreasing with the number of GPUs). This overhead is significant and should be reduced. Rewriting communication routines with MPI, instead of socket programming, in order to take advantage of optimized derived data-types [S+03] may be a necessary modification of PB&B@CLUSTER. As the number of GPUs increases, the coordinator spends a greater portion of time actually treating the requests. For 36 GPUs, work unit intersection, division and other checkpoint operations (e. g. periodically saving all work units to the disk) consume each 10-12% of the coordinator's time. In total, the coordinator is exploited 40% of the time when 36 GPUs are used. At this rate it is likely that incoming requests from workers are queuing up, causing the coordinator to become a bottleneck.

- **Scalability:**



(a) Average node processing rate (in Mn/s). (b) GPU efficiency for p GPUs ($\eta = \frac{T_1}{pT_p}$).

Figure 7.7: Average node processing rate and GPU efficiency

In order to evaluate the impact of the instance size on the efficiency of PB&B@CLUSTER the scaling experiment is repeated with instances *Ta053-h*, with $h \in \{1, 2, 3, 4, 5\}$. The obtained node processing rates is shown (in Mn/s) in Figure 7.7b and GPU efficiencies in Figure 7.7a. GPU efficiency is defined analogous to the conventional parallel efficiency definition, replacing processors with GPUs. The smallest of these instances, *Ta053-1* is solved in 25 minutes on a single GPU and the largest, *Ta053-5*, lasts for 220 minutes on a single device.

Unsurprisingly, efficiency and node processing rates increase as the size of the explored tree increases. In order to exploit all 36 available GPUs efficiently ($> 70\%$), the flowshop instance to be solved should at least require 10^9 node branchings (*Ta053-3*, requiring 2 hours of processing on a single GPU). For smaller instances, the repartition of the search space, represented by the interval $[0, N![$, among $36 \times 16\,384 \approx 600\,000$ IVMs incurs too much overhead. Even for instance *Ta053-3*, solved in 4.5 minutes on 36 GPUs, each IVM branches on average only $\approx 1\,500$ permutations.

7.4.2 Experiments on a cluster of GPUs and CPUs

Experimental protocol

- **Hardware testbed:** In order to evaluate the scalability of PB&B@CLUSTER with GPUs *and* CPUs further experiments are performed on the **Kepler** cluster located at the Mathematics and Operations Research Department at UMONS University. *Kepler* is a cluster of 20 low-power system-on-a-chip (SoC) devices. Each of the 20 nodes is a Nvidia Tegra K1 SoC featuring a 32-bit quad-core ARM Cortex-A15 CPU and a Kepler GK20A GPU containing 192 CUDA cores. Tegra K1 is primarily designed for graphics intensive mobile applications, like gaming, and is used in different tablet computers. Therefore, battery lifetime is a main design objective and Tegra K1 has a TDP of less than 10 W.

- **Problem instance:** As the Kepler cluster is not very powerful, one of the 20 jobs and 20 machines instances is chosen to validate the PB&B@CLUSTER approach. The selected instance is $Ta022$ which can be solved in 196 minutes on a CPU core.

- **GPU-CPU efficiency definition:** We evaluate the scalability of PB&B@CLUSTER as follows. For a given problem instance, the node processing rate (in nodes/sec) on a single CPU (resp. GPU) is measured. Based on these rates the expected node processing rate on a system using n GPUs and m CPUs is deduced. The efficiency on a (n GPU+m CPU) system is expressed as a percentage of this achieved rate.

  Formally, let α and β be the node processing rates achieved by a single CPU (resp. GPU), and let $\tau_{m,n}$ be the processing speed measured for a system composed of m CPUs and n GPUs. The efficiency $\eta_{m,n}$ for this configuration is computed as

$$\eta_{m,n} = \frac{\tau_{m,n}}{m\alpha + n\beta} \times 100\%$$

  For instance, solving $Ta022$ on a single CPU (resp. GPU) a node processing rate of $\alpha = 4.90\ kn/s$ (resp. $14.12\ kn/s$) is achieved. Using 8 GPUs and 4 CPUs, the same instance is solved with an average node processing rate of $131.6\ kn/s$. Supposing linear scalability with CPUs and GPUs one can expect to achieve a node processing rate of $8 \times 14.12 + 4 \times 4.90 = 132.6\ kn/s$. In this case, PB&B@CLUSTER reaches an efficiency of $\eta_{4,8} = 99.2\%$.

Obtained results

Table 7.1 reports the efficiency $\eta_{m,n}$ achieved for m, n=4, 8, 12, 16, 20 solving flowshop instance *Ta022*. The results shown in this table are averages over 5 independent runs and the relative standard deviation (RSD) is shown on the right-hand side. The initial runs on one CPU (resp. GPU) were also performed 5 times. The two tables on top (Tables 7.1a and 7.1b) show results using 128 IVMs per GPU, the two bottom tables (Tables (7.1c and 7.1d) show results for 1024 IVMs per GPU. Each CPU PB&B process is a 4-threaded (4-IVM) exploration process. The coordinator process runs on a reserved node (without concurrent exploration processes) except for the runs where #CPUs=20 or #GPUs=20. The node processing rates for individual workers are the following:

- (1) $4.90\,\text{kn/s}$ for one CPU worker,
- (2) $14.12\,\text{kn/s}$ for one GPU worker with $T = 128$
- and (3) $22.10\,\text{kn/s}$ for one GPU worker with $T = 1024$.

For reference, the sequential processing rate on a Intel E5-2630v3 CPU is about $1.93\,\text{kn/s}$. For $T = 128$, resp. $T = 1024$ IVMs, the maximal rate of the hybrid system is therefore $380\,\text{kn/s}$, resp. $540\,\text{kn/s}$. Using all 20 nodes, the achieved processing rates is $293\,\text{kn/s}$ ($\eta_{20,20} = 77\%$) , resp. $373\,\text{kn/s}$ ($\eta_{20,20} = 69\%$), meaning that *Ta022* is solved in 75, resp. 59 seconds.

For all configurations less than 2% of nodes is explored redundantly. However, one can observe that execution time variability is significant, especially when the number of CPUs is high and the number of GPU is low. A more detailed analysis of exceptionally slow explorations reveals that a high number of work allocations occur in the shutdown phase. While this indicates a better robustness for larger instances, experimental confirmation is needed. Although a threshold below which work units are not divided, an efficient handling of the shutdown phase revealed challenging.

7.4.3 Resolution of a big instance

Experimental protocol

- **Hardware testbed:** The Ouessant cluster, described in Subsection 7.4.1, is used for the experiments of this subsection.
- **Problem instance:** The objective is to solve the instance *Ta056*. To the best of our knowledge, of the 10 Taillard instances defined by 50 jobs and 20 machines (*Ta051-Ta060*), *Ta056* is currently the only one for which the best known complete permutation is proven to be optimal.

		GPU x					
	Eff	**0**	**4**	**8**	**12**	**16**	**20**
	0		95	91	86	82	79
	4	100	103	93	87	82	78
GPU x	**8**	98	99	95	87	81	77
	12	96	91	89	89	84	79
	16	94	89	89	84	84	78
	20	92	86	85	81	79	77

(a) $\eta_{m,n}$ - *Ta022* - **128** IVM/GPU.

		GPU x					
	Eff	**0**	**4**	**8**	**12**	**16**	**20**
	0		0.1	0.8	0.6	0.7	2.3
	4	0.1	3.4	0.2	2.0	1.8	1.2
GPU x	**8**	0.2	4.3	5.0	2.5	4.2	1.8
	12	0.4	8.9	6.9	3.4	6.9	5.2
	16	0.2	11.4	7.7	8.9	7.5	4.5
	20	0.7	6.5	11.2	3.6	9.2	5.9

(b) RSD - *Ta022* - **128** IVM/GPU.

		GPU x					
	Eff	**0**	**4**	**8**	**12**	**16**	**20**
	0		96	90	86	81	81
	4	100	88	87	83	78	75
GPU x	**8**	102	84	83	82	78	73
	12	100	79	88	76	75	71
	16	101	77	87	77	78	72
	20	99	83	77	79	73	69

(c) $\eta_{m,n}$ - *Ta022* - **1024** IVM/GPU.

		GPU x					
	Eff	**0**	**4**	**8**	**12**	**16**	**20**
	0		0.8	1.3	0.9	1.1	0.9
	4	0.1	2.1	2.9	2.4	2.3	1.5
GPU x	**8**	0.2	2.5	7.7	2.0	4.7	4.2
	12	0.4	1.0	1.5	4.5	4.6	4.7
	16	0.2	9.6	3.7	4.7	3.8	2.3
	20	0.7	13.2	11.7	3.1	8.0	3.0

(d) RSD - *Ta022* - **1024** IVM/GPU.

Table 7.1: Mixed GPU-CPU efficiency and Relative Standard Deviation (RSD) for resolution of flowshop instance *Ta022* (22.1×10^6 nodes) using 20 Tegra K1. The upper (resp. lower) row shows results for $T = 128$ (resp. $T = 1024$) IVMs/GPU. Average efficiency and RSD over 5 runs.

Obtained results

- **Elapsed time:**

	Serial PB&B	*B&B@Grid*	*PB&B@CLUSTER*
#GPUs	0	0	$9 \times 4 = 36$
#CPU cores	1	Aver. 328	$9 \times 2 = 18$
Elapsed time	22 years	25 days	9 hours

Table 7.2: Exploration statistics for resolution of flowshop instance *Ta056*.

The optimal complete permutation of *Ta056* was found and proven in 2006 using B&B@Grid [MMT07c]. The resolution required 25 days of processing, exploiting on average 328 CPU cores distributed on 9 clusters of the French experimental testbed Grid'5000 [2]. This result is used as a reference for the three resolutions of *Ta056* which are performed under identical initial conditions. As in the B&B@Grid experiment reported in [MMT07c] the initial upper bound is set to the optimal cost plus one

[2]https://www.grid5000.fr/

unit, i.e. $3\,680$, which allows one to verify the correctness of the algorithm. Our new resolution of *Ta056* using PB&B@CLUSTER found the same optimal cost $(3\,679)$, and the same optimal complete permutation.

On Ouessant and using PB&B@CLUSTER, *Ta056* is solved in 9 hours. Compared with the B&B@Grid resolution, the execution time is reduced by a factor of about $65\times$. Compared to the estimated sequential execution time of 22 years, the execution time is reduced by a factor of about $20,000\times$.

- **Energy consumption:**

	B&B@Grid	*PB&B@CLUSTER*
#GPUs	0	$9 \times 4 = 36$
TDP of a GPU	-	300 W
#CPUs	Aver. 328	$9 \times 2 = 18$
TDP of a CPU	30 W+	225 W
Elapsed time	25 days	9 hours
Energy consumption	≈6000 kWh	≈130 kWh

Table 7.3: Exploration statistics for resolution of flowshop instance *Ta056*.

Table 7.3 indicates an approximate value for the energy consumption of each resolution. These values are based on the Thermal Design Power (TDP) of CPUs and GPUs, as listed by the respective vendors. For example, the GTX 980 GPU is listed with a TDP of 165 W and the host Xeon CPUs with 85 W, so an indicative value for the energy consumption is given by $(36 \times 300 + 18 \times 225)\mathrm{W} \times 9h \approx 130\,\mathrm{kWh}$.

For the 2006 resolution using B&B@Grid the energy consumption can only be roughly estimated. About $\frac{2}{3}$ of CPU cores in the computational pool exploited by B&B@Grid in 2006 are AMD Opteron dual-core CPUs, $90\,\mathrm{nm}$ feature size, with clock rates between 2.0 and $2.2\,\mathrm{GHz}$. The remaining $\frac{1}{3}$ are Intel Pentium 4 and Celeron single-core processors with similar clock rates. The most energy efficient models of this type of CPUs are listed with TDP values above 30 W. Taking into account that most CPUs are dual-core, an optimistic estimation for the energy consumption is 328 CPU cores $\times\ 30W \times 25$d $\times\ 24$h/d $\approx 6000\,\mathrm{kWh}$.

- **Load balancing:**

Figure 7.8 illustrates the workload repartition among GPU and CPU based workers, in terms of branched nodes. In addition to four GPU workers, on each Ouessant node one multi-threaded CPU based worker with 160 IVM is used ($2\times$ 10 cores$\times$ 8 threads).

In this configuration, each multi-core PB&B process branches on average about $\frac{1}{10}$ the amount of nodes branched by an average GPU PB&B. Using 4 times as many GPU based workers as CPU workers, in total less than 3% of node branchings are performed on a CPU. One can see in Figure 7.8 that workers of the same type perform a roughly equal amount of work. However, a node branching represents a variable amount of work. Therefore, the number of branched nodes is only an approximative indicator for load balancing.

	PB&B@CLUSTER
Branched nodes	175.8×10^9
$T_{coordinator}$	38.2 min
Coord. exploitation (%)	7.1%
#checkpoints	3 568 368
#work allocations	33 387

Table 7.4: Exploration statistics for resolution of flowshop instance *Ta056*.

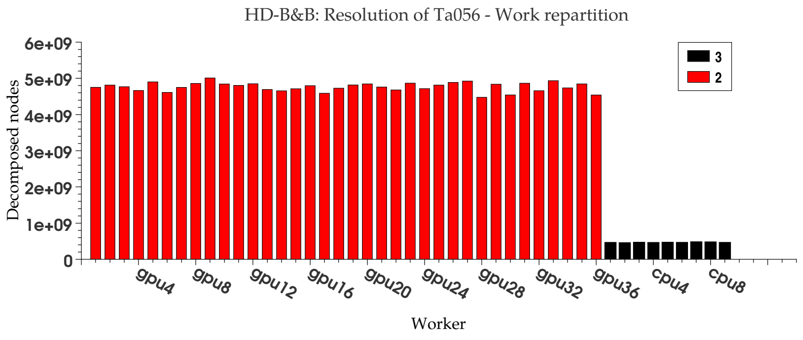


Figure 7.8: Number of nodes branched per worker. Resolution of *Ta056* on 9 Minsky nodes (Ouessant), using $9\times(4$ GPU based workers with $16\,384$ IVM + 1 CPU based worker with 160 IVM).

Another indicator for the load imbalance throughout the execution is the number of work allocations (work stealing operations). A worker only requests new work from the coordinator when no more work is locally available. The results indicate that workers run out of work very rarely.

These metrics change significantly for the hybrid resolution on Ouessant. In order to balance the workload between 36 GPUs and 9 multi-core CPUs, more than $33\,000$ work unit allocations and $3\,500\,000$ checkpointing operations are performed. In that case the farmer processor is exploited 7% of the time, i. e. 38 minutes out of 9 hours.

7.5 Conclusions

In this chapter we revisited the design of B&B@Grid to enable the integration of GPU- and CPU-based workers. When B&B@Grid was designed in mid-2000 most components of computational grids were single-core and dual-core CPUs. Today, more than 15 years later, large-scale HPC platforms are becoming increasingly heterogeneous, integrating GPUs and CPUs with larger core-counts.

The extension of B&B@Grid to PB&B@CLUSTER, the proposed PB&B for hybrid distributed HPC clusters, includes the redefinition of work units and a modification of the communication scheme to allow asynchronous checkpointing operations that overlap with worker computations.

A very large flowshop instance defined by 50 jobs and 20 machines, *Ta056*, was successfully solved on GPU powered clusters with a total of up to 130 000 GPU cores. A first resolution of this instance was performed in 2006, using B&B@Grid to exploit on average 328 CPU cores in a computational grid during 25 days. Using PB&B@CLUSTER the resolution of *Ta056* was performed in 9 hours on a cluster composed of 36 GPUs.

Low exploitation rates of the coordinator process and experiments performed with smaller instances are indicators for the good scalability of PB&B@CLUSTER. For a set of 50-job flowshop instances requiring 3.7 hours of computation on a single GPU, a relative speedup of $30\times$ is achieved on 36 GPUs, solving this instance in 7.5 minutes on average.

PB&B@CLUSTER was also experimented on a mini-cluster of 20 systems-on-a-chip designed for mobile devices. Experimental results show that the availability of efficient data types for inter-node communication is a key component for the performance of the central coordinator process.

CHAPTER 8

Conclusions and perspectives

The research works, presented in this manuscript, are focused on the exact resolution of a certain type of economic problems, where the objective is to find the best permutation. A large number of methods, often serial, have been developed to solve these problems. A wide range of exact methods can be classified as PB&B algorithms. The solution space is explored by dynamically building a tree, whose size in terms of number of nodes is huge. Therefore, solving these problems often requires the use of a high performance computing system.

This last chapter recalls the main contributions, presented in this manuscript, and the conclusions to draw from these contributions. In addition, the chapter gives an overview of our other contributions in PB&B algorithms which are not described in this manuscript. And finally, the chapter indicates our main future works in the field PB&B algorithms.

8.1 Main contributions

Our works led to the development of four approaches, renamed in this manuscript PB&B@CORE, PB&B@CPU, PB&B@GPU and PB&B@CLUSTER. The first two methods are developed during the thesis of Rudi Leroy, and the two others in the thesis of Jan Gmys.

- The PB&B@CORE approach rethinks the conventional data structure used in PB&B. When exploring the PB&B tree, the nodes, generated but not yet processed, are often stored in a linked-list. **For a single CPU core, we have developed a new data structure, called IVM, to replace the linked-list.** Compared to the linked-list, the experiments, performed for the resolution of the 50×10 (i.e. 50 jobs and 10 machines) flowshop problem instances [Tai93], show that the management of an IVM pool requires on average ≈ 3 times less memory and ≈ 35 times less computing power.

- The PB&B@CPU approach, obtained using the IVM structure of PB&B@CORE, allows the parallelization of the PB&B on multi-core CPU processors. **For such architectures, and unlike conventional parallelization, where the work unit is often a**

set of nodes, the work unit of our approach is an interval of factoradics. A factoradic is an integer encoded in the factorial enumeration system. The experiments performed on a multi-core CPU accelerator, for the resolution of the ten 20×20 flowshop instances, show that the management of an IVM pool is on average ≈ 10 times faster than the management of linked-list pool.

- The PB&B@GPU approach, based on the factoradic intervals of PB&B@CPU, allows the parallelization of PB&B on GPU, which is a processor with a SIMD architecture. Previously, there have been many attempts in the literature to run the most time-consuming part (hot spot) of the PB&B on GPU and the rest of the algorithm on CPU. **The advantage of our approach, based on a revisited work stealing method, is to offload the whole execution of PB&B, whose tree has an irregular structure, to the GPU accelerator, which has a regular SIMD architecture.** When solving the ten 20×20 flowshop instances, experiments show that on average the resolution time is reduced on average by ≈ 3.3 times compared to a GPU linked-list PB&B.

- The PB&B@CLUSTER approach, which is based on the three previous approaches, allows the deployment of a PB&B on a cluster of CPU processors and GPU accelerators. **For a distributed memory architecture, such as a cluster, our approach uses and redefines a certain number of basic operators known in set theory**. This approach is validated on Ta056, which is a difficult 50×20 flowshop instance.

  The resolution of this instance required ≈ 22 days of computing using on average 328 CPU cores and a total of ≈ 6000 kWh[MMT07a]. Our new approach solved this instance in ≈ 9 hours with an energy consumption of ≈ 130 kWh.

The PB&B@CLUSTER approach is not yet published while the others are published mainly in four journals and one conference, namely two CCPE [GMMT17, GLM+16], one Parallel computing [GMMT16] and one FGCS [MGMT18a], and the 28th IEEE IPDPS conference [MLMT14a]. The PB&B@GPU approach is also awarded with the Best Paper Award at the 11th PPAM conference [GMM+15]. As future research directions for this work, we have identified some challenging perspectives summarized in the following:

8.2 Other contributions

In addition to these four contributions, we have several other contributions in PB&B algorithms. Some of these contributions are important, but are not presented in this manuscript. In particular, we would like to describe two of them.

- **Efficient branching and bounding operations:** In this contribution, we present a new node decomposition scheme that combines dynamic branching and lower bound refinement strategies in a computationally efficient way. To alleviate the computational burden of the two-machine bound used in the refinement stage, we propose an online learning-inspired mechanism to predict promising couples of bottleneck machines. The algorithm offers multiple choices for branching and bounding operations and can explore the search tree either sequentially or in parallel on multi-core CPUs.

  In order to empirically determine the most efficient combination of these components, a series of computational experiments with 600 flowshop benchmark instances is performed. A main insight is that the problem size, as well as interactions between branching and bounding operations substantially modify the trade-off between the computational requirements of a lower bound and the achieved tree size reduction. Moreover, we demonstrate that parallel tree search is a key ingredient for the resolution of large problem instances, as strong super-linear speedups can be observed. An overall evaluation using two well-known benchmarks indicates that the proposed approach is superior to previously published PB&B algorithms.

  For the first benchmark we report the exact resolution - within less than 20 minutes - of two instances defined by 500 jobs and 20 machines that remained open since more than 25 years, and for the second a total of 88 improved best known upper bounds, including proofs of optimality for 71 of them. This contribution is under a minor revision in EJOR journal.

- **PB&B on many-core processors:** On the road to exascale, coprocessors are increasingly becoming key building blocks of high performance computing platforms. In addition to their energy efficiency, these many-core devices boost the performance of multi-core processors. In this contribution, we revisit the design and implementation of PB&B algorithms for multi-core processors and Intel Xeon Phi coprocessors considering the offload mode as well as the native one. In addition, two major parallel models are considered: the master-worker and the work pool models. We address several parallel computing issues including processor-coprocessor data transfer optimization and vectorization.

  The proposed approaches have been experimented using the flowshop and two hardware configurations equivalent in terms of energy consumption: Intel Xeon E5-2670 processor and Intel Xeon Phi 5110P coprocessor. The reported results show that: (1) the proposed vectorization mechanism reduces the execution time by (resp.) in the many-core (resp. multi-core) approach; (2) the offload mode allows a faster execution on MIC than the native mode for most flowshop problem instances; (3) the many-core approach (offload or native) is in average twice faster than the multi-core approach; (4) the work pool parallel model is more suited for many/multi-core PB&B applied to flowshop than the master-worker model because of its irregular nature. This contribution is published in the FGCS journal [MGMT18b].

8.3 Perspectives

The contributions presented in this document open up new perspectives. These perspectives are mainly related to the validation of our approaches using other permutation problems, and a cluster powered with a larger number GPU nodes. Our future works can be summarized with the following points:

- As a short-term future work, we will validate our approaches on other single permutation permutation problems such as TSP, Job-Shop and QAP, and permutation problems with more than one permutation like Q3AP [Meh11].
- The experimental results, obtained with PB&B@CLUSTER, indicate that the algorithm is scalable on larger GPU-enhanced clusters. We plan to validate this by attempting the resolution of previously unsolved flowshop instances on a large GPU-powered supercomputer. To resolve these instances, we obtained 50,000 hours of computing on the 1,000 GPU nodes of Jean Zay supercomputer. Jean Zay is the converged platform acquired by the French Ministry of Higher Education, Research and Innovation through the intermediary of the French civil company, GENCI (Grand Equipement National De Calcul Intensif). The Jean Zay computer was installed at IDRIS, national computing centre for the French National Centre for Scientific Research (CNRS), in 2019.
- In order to further improve scalability of the approach, specially in an exascale environment [CM19], the coordinator process, usually running on a multi-core CPU, should be parallelized. Also, the checkpointing mechanism should be revisited. PB&B@CLUSTER uses the checkpointing mechanism inherited from B&B@Grid, making the approach tolerant against node failures. However, as a large portion is shifted to lower levels it becomes important to make the approach fault-tolerant against failures at the GPU and multi-core CPU level.
- The IVM data structure revealed itself particularly well suited for fine-grained permutation problems. For example, sampling methods based on the optimization of

latin hypercubes can be modeled as (multi-)permutation problems. As a future research direction we plan to revisit the IVM-based algorithm to enable the resolution of multi-permutation problems.

- Experimental results have shown strong performance variations according to the used node evaluation function. Having different bounds for the same problem matching implementations with underlying hardware. A challenging improvement of the PB&B algorithm consists in implementing a library of lower bounds for the same problem, in order to enable the different workers to use the node evaluation function which is the best fit for the underlying hardware.

Bibliography

[ABEB+16] Dabah Adel, Ahcène Bendjoudi, Didier El-Baz, Ait Zai Abdelhakim, et al. Gpu-based two level parallel b&b for the blocking job shop scheduling problem. 2016.

[ABGL02] Kurt Anstreicher, Nathan Brixius, Jean-Pierre Goux, and Jeff Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3):563–588, 2002.

[ACR] ACRO. A common repository for optimizers, sandia national laboratories. `https://software.sandia.gov/trac/acro/`. Accessed: 2017-09-23.

[AGA99] A. Allahverdi, J.N.D Gupta, and T. Aldowaisan. A review of scheduling research involving setup considerations. *Omega*, 27(2):219–239, 1999.

[AMTM17] E. Alekseeva, M. Mezmaz, D. Tuyttens, and N. Melab. Parallel multi-core hyper-heuristic GRASP to solve permutation flow-shop problem. *Concurrency and Computation: Practice and Experience*, 29(9), 2017.

[ASW+14] Tae-Hyuk Ahn, Adrian Sandu, Layne T Watson, Clifford A Shaffer, Yang Cao, and William T Baumann. A framework to analyze the performance of load balancing schemes for ensembles of stochastic simulations. *International Journal of Parallel Programming*, 43(4):597–630, 2014.

[Bas05] Matthieu Basseur. Conception de métaheuristiques coopératives pour l'optimisation multi-objective: Application au flowshop. *PhD Thesis from Université Lille 1*, 2005.

[BG76] M.C. Bonney and S.W. Gundry. Solutions to the constrained flowshop sequencing problem. *Operational Research Quarterly*, page 869, 1976.

[BHP05] David A Bader, William E Hart, and Cynthia A Phillips. Parallel algorithm design for branch and bound. *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at INFORMS 2004, Denver, CO*, 76, 2005.

[BL99] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[BMMT18] Guillaume Briffoteaux, Nouredine Melab, Mohand Mezmaz, and Daniel Tuyttens. An adaptive evolution control based on confident regions for surrogate-assisted optimization. In *HPCS 2018 - International Conference on High Performance Computing & Simulation*, Orléans, France, July 2018.

[BMT12a] A. Bendjoudi, N. Melab, and E. G. Talbi. Hierarchical branch and bound algorithm for computational grids. *Future Gener. Comput. Syst.*, 28(8):1168–1176, October 2012.

[BMT12b] Ahcène Bendjoudi, Nordine Melab, and El-Ghazali Talbi. An adaptive hierarchical master–worker (ahmw) framework for grids—application to b&b algorithms. *Journal of Parallel and Distributed Computing*, 72(2):120–131, 2012.

[BMT14] Ahcene Bendjoudi, Nouredine Melab, and El-Ghazali Talbi. Fth-b&b: A fault-tolerant hierarchicalbranch and bound for large scaleunreliable environments. *IEEE Transactions on Computers*, 63(9):2302–2315, 2014.

[Bob] Bobpp. Bobpp framework, université de versailles. `http://www.prism.uvsq.fr/~blec/bobpp/main.html`. Accessed: 2017-09-23.

[Cha13] I. Chakroun. *Parallel heterogeneous Branch and Bound algorithms for multi-core and multi-GPU environments*. PhD thesis, Université Lille 1, 2013.

[CM13] I. Chakroun and N. Melab. Operator-level gpu-accelerated branch and bound algorithms. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, pages 280–289, 2013.

[CM19] Tiago Carneiro and Nouredine Melab. Productivity-aware design and implementation of distributed tree-based search algorithms. In João M. F. Rodrigues, Pedro J. S. Cardoso, Jânio Monteiro, Roberto Lam, Valeria V. Krzhizhanovskaya, Michael H. Lees, Jack J. Dongarra, and Peter M.A. Sloot, editors, *Computational Science – ICCS 2019*, pages 253–266, Cham, 2019. Springer International Publishing.

[CMGH08] Leocadio G Casado, JA Martinez, Inmaculada García, and Eligius MT Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods & Software*, 23(5):689–701, 2008.

[CMMB13] 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*, 25(8):1121–1136, 2013.

[CMMT13] Imen Chakroun, Nordine Melab, Mohand Mezmaz, and Daniel Tuyttens. Combining multi-core and gpu computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing*, 73(12):1563–1577, 2013.

[CMNLdC11] 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)*, pages 41–47, 2011.

[CNNdC12] Tiago Carneiro, Ricardo Holanda Nobre, Marcos Negreiros, and Gustavo Augusto Lima de Campos. Depth-first search versus Jurema search on GPU branch-and-bound algorithms: A case study. *NVIDIA's GCDF - GPU Computing Developer Forum on XXXII Congresso da Sociedade Brasileira de Computação (CSBC)*, 2012.

[CO] COIN-OR. Computational infrastructure for operations research. `https://www.coin-or.org/documentation.html`. Accessed: 2017-09-23.

[Cra06] Teodor Gabriel Crainic. Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*, 1:1–28, 2006.

[CZ06] S. Climer and W. Zhang. Cut-and-solve: an iterative search strategy for combinatorial optimization problems, artificial intelligence. 170:714–738, 2006.

[EHP15] Jonathan Eckstein, William E Hart, and Cynthia A Phillips. Pebbl: an object-oriented framework for scalable parallel branch and bound. *Mathematical Programming Computation*, 7(4):429–469, 2015.

[EPH00] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch-and-bound. Research Report 40–2000, RUTCOR, 2000.

[EPH01] Jonathan Eckstein, Cynthia A Phillips, and William E Hart. Pico: An object-oriented framework for parallel branch and bound. *Studies in Computational Mathematics*, 8:219–265, 2001.

[EPS09a] Y. Evtushenko, M. Posypkin, and I. Sigal. A framework for parallel large-scale global optimization. *Computer Science-Research and Development*, 23(3):211–215, 2009.

[EPS09b] Yuri Evtushenko, Mikhail Posypkin, and Israel Sigal. A framework for parallel large-scale global optimization. *Computer Science-Research and Development*, 23(3-4):211–215, 2009.

[FRvLP10] Frank Feinbube, Bernhard Rabe, M von Löis, and Andreas Polze. Nqueens on cuda: Optimization issues. In *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*, pages 63–70. IEEE, 2010.

[GC94] B. Gendron and T.G. Crainic. Parallel Branch and Bound Algorithms: Survey and Synthesis. *Operations Research*, 42:1042–1066, 1994.

[GGS04] B. Goldengorin, D. Ghosh, and G. Sierksma. Branch and peg algorithms for the simple plant location problem. *Computers & Operations Research*, 31:241–255, 2004.

[GJS76] M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flow-shop and job-shop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

[GLM$^+$16] J. Gmys, R. Leroy, M. Mezmaz, N. Melab, and D. Tuyttens. Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms. *Concurrency and Computation: Practice and Experience*, 28(18):4463–4484, 2016.

[GMM$^+$15] J. Gmys, M. Mezmaz, N. Melab, R. Leroy, and D. Tuyttens. IVM-based Work Stealing for Parallel Branch-and-Bound on GPU . In *Proc. of Int. Conference on Parallel Processing and Applied Mathematics*, Krakow, Poland, September 2015.

[GMMT16] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. A gpu-based branch-and-bound algorithm using integer-vector-matrix data structure. *Parallel Computing*, 59:119–139, 2016.

[GMMT17] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. Ivm-based parallel branch-and-bound using hierarchical work stealing on multi-gpu systems. *Concurrency and Computation: Practice and Experience*, 29(9), 2017.

[Gmy17] J. Gmys. *Heterogeneous Cluster Computing for Many-Task Exact Optimization : Application to Permutation Problems*. PhD thesis, Université de Mons et Université de Lille, 2017.

[Gol66] S. Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399–401, Jul 1966.

[HS05] S. Reza Hejazi and S. Saghafian. Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research*, 43(14):2895–2929, 2005.

[HSH$^+$17] Juan F. R. Herrera, José M. G. Salmerón, Eligius M. T. Hendrix, Rafael Asenjo, and Leocadio G. Casado. On parallel branch and bound frameworks for global optimization. *Journal of Global Optimization*, Mar 2017.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with cuda. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007.

[IF00] A. Iamnitchi and I. Foster. A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems. *29th International Conference on Parallel Processing (ICPP), Toronto, Canada, August*, pages 21–24, 2000.

[JAO$^+$11] John Jenkins, Isha Arkatkar, John D. Owens, Alok Choudhary, and Nagiza F. Samatova. Lessons learned from exploring the backtracking paradigm on the gpu. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 425–437, Berlin, Heidelberg, 2011. Springer-Verlag.

[Joh54] S.M. Johnson. Optimal two and three-stage production schedules with setup times included. *Naval Research Logistis Quarterly*, 1:61–68, 1954.

[Knu97] D.E. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. *Reading, Ma*, page 192, 1997. ISBN=9780201896848.

[Knu98] Donald E Knuth. The art of computer programming, volume 2: Seminumerical algorithms, addison-wesley. *Reading, Massachusetts*, 1998.

[KRR88] Vipin Kumar, V. N Rao, and K. Ramesh. Parallel depth first search on the ring architecture. Technical report, Austin, TX, USA, 1988.

[KS80] J. R. King and A. S. Spachis. Heuristics for flow-shop scheduling. *International Journal of Production Research*, 18(3):345–357, 1980.

[Lai88a] C-A. Laisant. Sur la numération factorielle, application aux permutations. *Bulletin de la Société Mathématique de France*, 16:176–183, 1888.

[Lai88b] C-A. Laisant. Sur la numération factorielle, application aux permutations. *Bulletin de la Société Mathématique de France*, 16:176–183, 1888.

[Lai88c] C-A Laisant. Sur la numération factorielle, application aux permutations. *Bulletin de la Société Mathématique de France*, 16:176–183, 1888.

[LEB12] M.E. Lalami and D. El-Baz. GPU Implementation of the Branch and Bound Method for Knapsack Problems. In *IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, pages 1769–1777, Shanghai, CHN, May 2012.

[Ler15a] R. Leroy. *Parallel Tree-based Exact Algorithms using Heterogeneous Many and Multi-core Computing for Solving Challenging Problems in Combinatorial Optimization*. PhD thesis, Université de Lille, 2015.

[Ler15b] Rudi Leroy. *Parallel Branch-and-Bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors*. PhD thesis, Université Lille 1, 2015.

[LLK78] J.K. Lenstra, B.J. Lageweg, and A.H.G. Rinnooy Kan. A General bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[LLW$^+$15] Liang Li, Hong Liu, Hao Wang, Taoying Liu, and Wei Li. A parallel algorithm for game tree search using GPGPU. *Parallel and Distributed Systems, IEEE Transactions on*, 26(8):2114–2127, 2015.

[MCA13] X. Meyer, B. Chopard, and P. Albuquerque. A branch-and-bound algorithm using multiple gpu-based lp solvers. In *20th Annual International Conference on High Performance Computing*, pages 129–138, Dec 2013.

[MCB14] Nouredine Melab, Imen Chakroun, and Ahcène Bendjoudi. Graphics processing unit-accelerated bounding for branch-and-bound applied to a permutation problem using data access optimization. *Concurrency and Computation: Practice and Experience*, 26(16):2667–2683, 2014.

[McC03] James McCaffrey. Using permutations in .net for improved systems security. *Microsoft Developer Network*, 2003.

[Meh11] Malika Mehdi. Parallel hybrid optimization methods for permutation based problems. *PhD Thesis from Université Lille 1 and Université du Luxembourg*, 2011.

[Men17] Tarek Menouer. Solving combinatorial problems using a parallel framework. *Journal of Parallel and Distributed Computing*, 2017.

[MGMT18a] Nouredine Melab, Jan Gmys, Mohand Mezmaz, and Daniel Tuyttens. Multi-core *versus* many-core computing for many-task branch-and-bound applied to big optimization problems. *Future Generation Comp. Syst.*, 82:472–481, 2018.

[MGMT18b] Nouredine Melab, Jan Gmys, Mohand Mezmaz, and Daniel Tuyttens. Multi-core versus many-core computing for many-task branch-and-bound applied to big optimization problems. *Future Generation Computer Systems*, 82:472–481, 2018.

[MLMT14a] M. Mezmaz, R. Leroy, N. Melab, and D. Tuyttens. A Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. In *Proc. of 28th IEEE Intl. Parallel and Distributed Processing Symp.*, Phoenix, Arizona, USA, May 2014.

[MLMT14b] Mohand Mezmaz, Rudi Leroy, Nouredine Melab, and Daniel Tuyttens. A Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. In *28th IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS)*, pages 1203–1212, Phoenix, AZ, May 2014.

[MMK$^+$11] M. Mezmaz, N. Melab, Y. Kessaci, Y.C. Lee, E-G. Talbi, A.Y. Zomaya, and D. Tuyttens. A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems. *Journal of Parallel and Distributed Computing*, 71(11):1497 – 1508, 2011. ISSN 0743-7315, DOI: 10.1016/j.jpdc.2011.04.007.

[MMT07a] M. Mezmaz, N. Melab, and E-G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In *21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–9, Long Beach, CA, March 2007.

[MMT07b] 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.

[MMT07c] M. Mezmaz, N. Melab, and E. G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems.

In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–9, March 2007.

[MMT13] M. Mezmaz, N. Melab, and D. Tuyttens. *A multithreaded branch-and-bound algorithm for solving the flow-shop problem on a multicore environment*, chapter 3, pages 53–70. Large Scale Network–Centric Distributed Systems. John Wiley & Sons, July 2013. ISBN-10: 0470936886, ISBN-13: 978-0470936887.

[PC04] R. Pastor and A. Corominas. Branch and win: Or tree search algorithms for solving combinatorial optimisation problems. *Top*, 1:169–192, 2004.

[PŽ09] R. Paulavičius and J. Žilinskas. Parallel branch and bound algorithm with combination of lipschitz bounds over multidimensional simplices for multicore computers. *Parallel Scientific Computing and Optimization*, pages 93–102, 2009.

[RLS04] Ted K Ralphs, Laszlo Ladányi, and Matthew J Saltzman. A library hierarchy for implementing scalable parallel search algorithms. *The Journal of Supercomputing*, 28(2):215–234, 2004.

[RS10] Kamil Rocki and Reiji Suda. *Parallel Minimax Tree Searching on GPU*, pages 449–456. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[S^+03] Xian-He Sun et al. Improving the performance of mpi derived datatypes by optimizing memory-access cost. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 412–419. IEEE, 2003.

[SECG11] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures. *The Journal of Supercomputing*, pages 1–9, 2011.

[SKK^+11] Vijay. A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In *Proc. of the 16th symposium on Principles and pratice of parallel programming*, PPoPP ’11, pages 201–212, New York, NY, USA, 2011. ACM.

[Tai93] E. Taillard. Benchmarks for basic scheduling problems. *Journal of Operational Research*, 64:278–285, 1993.

[TFMJ13] D. Tuyttens, H. Fei, M. Mezmaz, and J. Jalwan. Simulation-based Genetic Algorithm towards an Energy-efficient Railway Traffic Control. *Mathematical Problems in Engineering, Hindawi*, 2013:12 pages, 2013. DOI: 10.1155/2013/805410.

[TRDM17] James McCracken Trauer, Romain Ragonnet, Tan Nhut Doan, and Emma Sue McBryde. Modular programming for tuberculosis control, the “autumn” platform. *BMC Infectious Diseases*, 17(1):546, Aug 2017.

[VDM13] Trong-Tuan Vu, Bilel Derbel, and Nouredine Melab. Adaptive Dynamic Load Balancing in Heterogenous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search. In *7th International Learning and Intelligent OptimizatioN Conference (LION)*, Catania, Italy, January 2013. Lecture Notes in Computer Science.

[ZSW11] Tao Zhang, Wei Shu, and Min-You Wu. Optimization of n-queens solvers on graphics processors. In *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, APPT’11, pages 142–156, Berlin, Heidelberg, 2011. Springer-Verlag.