HAL
open science

Parallel Branch-and-Bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors

Rudi Leroy

To cite this version:

Rudi Leroy. Parallel Branch-and-Bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Lille 1, 2015. English. NNT : . tel-01248563

HAL Id: tel-01248563

https://inria.hal.science/tel-01248563v1

Submitted on 27 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Ecole Doctorale Sciences Pour l'Ingénieur Université Lille 1 Nord-de-France
Centre de Recherche en Informatique, Signal et Automatique de Lille (UMR CNRS 9189)
Centre de Recherche INRIA Lille Nord Europe
Maison de la Simulation

Thèse présentée pour obtenir le grade de docteur
Discipline : Informatique

Parallel Branch-and-Bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors.

Défendue par :

Rudi Leroy

Novembre 2012 - Novembre 2015

Devant le jury composé de:

Rapporteur :	Catherine Roucairol, Professeur, Université de Versailles
Rapporteur :	Pierre Sens, Professeur, Université Paris 6
Examinateur :	Matthieu Basseur, Maître de Conférences HDR, Université d'Angers
Examinateur :	Michel Kern, Chargé de Recherche, Inria Rocquencourt
Directeur de thèse :	Nouredine Melab, Professeur, Université de Lille 1
Co-encadrant :	Mohand Mezmaz, Chargé de Recherche, Université de Mons
Invité :	Daniel Tuyttens, Professeur, Université de Mons

Numéro d'ordre : 41840 | Année : 2015

Abstract

Solving large permutation Combinatorial Optimization Problems (COPs) using Branch-and-Bound (B&B) algorithms results in the generation of a very large pool of subproblems. Therefore, defining a dedicated data structure is crucial to store and manage efficiently that pool. In this Ph.D thesis, we propose an original data structure called Integer-Vector-Matrix (IVM) for permutation COPs based on the factorial number system. Consequently, we redefine the operators of the B&B algorithm acting on it. For performance evaluation in terms of memory footprint and CPU time usage, we conduct a complexity analysis and an extensive experimentation using the permutation Flow-Shop Scheduling Problem (FSP) as a case study. The deque data structure referred to as LL is considered in our work as a basis for comparison with IVM. Compared to LL, IVM requires up to n less memory than LL, n being the size of permutations. Moreover, the IVM-based B&B is up to one order of magnitude faster than its LL-based counterpart in managing the pool of subproblems. Another advantage of IVM over LL is that its memory requirement is constant and predictable.

On the other hand, according to the Top500 international ranking the tendency of HPC technologies is to use multi-core processors and many-core accelerators/coprocessors as key-building blocks. Another contribution of this thesis is to revisit parallel B&B for multi-core processors, GPU accelerators and MIC coprocessors using IVM and LL. As the tree explored by a B&B applied to FSP is highly irregular in shape and size, the thread-based Work Stealing (WS) paradigm is used for parallelization on multi-core processors. Unlike most related works that use concurrent data structures, we propose a private IVM-based and LL-based WS mechanism. In addition, work units are coded in a coalesced thus optimized way using factoradic-based intervals. We also investigate five different WS strategies. Extensive experiments show that in overall IVM outperforms LL in memory footprint and CPU time usage whatever is the used WS strategy. For the many-core parallelization, the proposed approach consists in performing the branching and bounding operators on the coprocessor. Such approach raises some issues, addressed in this thesis, mainly thread mapping, thread/branch divergence and data placement optimization on GPU, and vectorization on Intel Xeon Phi. An extensive experimental study shows that IVM is particularly more efficient than LL within the many-core context. Moreover, even with vectorization (of the lower bound), which allows a significant performance improvement on Intel Xeon Phi, the GPU-based approach is faster. Finally, the many-core approaches are faster than their multi-core counterpart by one order of

magnitude.

Keywords: Branch-and-Bound, Multi-core, Many-core (GPU, MIC), Permutation optimization problems, Flow-Shop.

Résumé

La résolution de problèmes de permutation en optimisation combinatoire au moyen d'algorithmes Branch-and-Bound (B&B) génère un très grand pool de sous-problèmes. Par conséquent, la définition d'une structure de données dédiée est cruciale pour stocker et gérer efficacement ce pool. Dans cette thèse, nous proposons une structure de données originale appelée Integer-Vector-Matrix (IVM) pour les problèmes de permutation basée sur la numération factorielle. En conséquence, nous redéfinissons les opérateurs de l'algorithme B&B agissant sur celle-ci. Pour l'évaluation de performance en termes de consommation mémoire et d'utilisation du CPU, nous avons réalisé une analyse de complexité et une expérimentation intensive en utilisant le problème d'ordonnancement de type Flow-Shop (FSP) comme étude de cas. La structure de données *deque*, désignée par LL dans ce manuscrit, est utilisée comme base de comparaison pour évaluer la performance d'IVM. Par rapport à LL, IVM nécessite jusqu'à n fois moins de mémoire que LL, n étant la taille des permutations. En outre, B&B basé sur IVM est environ 10 fois plus rapide que son équivalent basé sur LL dans la gestion du pool de sous-problèmes. Un autre avantage d'IVM sur LL est que ses besoins en mémoire sont constants et prévisibles.

D'autre part, d'après le classement international Top500 la tendance des technologies HPC est d'utiliser des processeurs multi-coeurs et des accélérateurs/coprocesseurs comme des briques-clés pour la construction de machines. Une autre contribution de cette thèse est de revisiter l'algorithme B&B parallèle pour les processeurs multi-coeurs, les accélérateurs GPU et les coprocesseurs MIC en utilisant IVM et LL. Comme l'arbre exploré par un B&B appliqué au FSP est hautement irrégulier en forme et en taille, le paradigme de vol de cycles (WS) basé sur les threads est utilisé pour la parallélisation multi-coeur. Contrairement à la majorité des travaux existants, qui utilisent des structures de données concurrentes, nous proposons un mécanisme de WS utilisant des structures (IVM et LL) privées ou "distribuées". En outre, les unités de travail sont codées de manière compressée et donc optimisée en utilisant des intervalles de factorielles. Nous étudions également cinq stratégies de WS différentes. Une expérimentation intensive montre que globalement IVM est plus efficace que LL en termes d'empreinte mémoire et d'utilisation du temps CPU et ce quel que soit la stratégie WS utilisée. Pour la parallélisation multi-coeur, l'approche proposée consiste à effectuer le branchement et l'évaluation des bornes sur le coprocesseur.

Cette approche soulève des problèmes, adressés dans cette thèse, notamment le mapping de threads, la divergence de branches/threads et l'optimisation du placement des données sur GPU, et la vectorisation sur Intel Xeon Phi. Une étude expérimentale montre qu'IVM est particulièrement plus efficace que LL dans le contexte many-core. En outre, même avec la vectorisation (de la borne inférieure), ce qui permet une amélioration significative des performances sur les processeurs Intel Xeon Phi, l'approche sur GPU est plus rapide. Enfin, les approches many-core sont environ 10 fois plus rapides que leurs homologues multi-core.

Mots clés: Branch-and-Bound, Multi-coeur, Many-core (GPU, MIC), Problèmes de permutation, Flow-Shop.

Acknowledgments

List of Figures

2.1 Illustration of a permutation FSP where $n = 3$ and $m = 4$. The table shows the processing times of the jobs on each machine. The Gantt diagram shows the optimal solution for this particular instance. 13

3.1 An example of a pool obtained when solving a permutation problem of size 4. 34
3.2 LL-based (or deque-based) representation of a pool of subproblems 36
3.3 IVM representation of a pool of subproblems. 40
3.4 Representation of the IVM data structure within the context of the whole tree. 42
3.5 IVM representation of a pool of subproblems with a direction vector 43
3.6 IVM representation of eliminated subproblems. 43
3.7 Selection Operator. 44
3.8 Branching Operator. 45
3.9 Elimination Operator when the subproblem is not the last of its row. . . . 47
3.10 Elimination Operator when the subproblem is the last of its row. 47

4.1 Illustration of the multi-parametric parallel model. 57
4.2 Illustration of the parallel evaluation of bounds model. 57
4.3 Illustration of the parallel tree exploration model. 59
4.4 IVM representation of a pool of subproblems. 61
4.5 Representation of an interval division at an arbitrary point. 65
4.6 Representation of an interval division at a point chosen to avoid redundancy. 66
4.7 Representation of a generalization of an interval division between subtrees. 68
4.8 Illustration of the LL-B&B and a IVM-B&B parallel algorithms using the same work stealing strategy. 70
4.9 Entity-relationship model of the database containing the results of the experiments. 74
4.10 Comparison of speedup and sharing events for 20 jobs on 20 machines for IVM and LL. 76

5.1 Coprocessor-based B&B 83
5.2 Hardware view: GPU = coprocessor of CPU. 85
5.3 Software view: Parallel program = weakly parallel/serial host code + massively parallel device code. 85

5.4 Software view: Parallel program = grid(s) of block(s) of threads executed as warps of 32 threads. 86
5.5 Hardware view: Intel Xeon Phi = coprocessor of CPU. 89
5.6 Hardware view: Knights Corner core. 90
5.7 Elapsed time vs. Number of IVMs. 94
5.8 Comparison between the linked-list and IVM-based B&B in terms of elapsed CPU time when solving instance 30 using different number of hardware threads. 97
5.9 Comparison between the linked-list and the IVM-based B&B in terms of the number page faults when solving instance 30 using different number of hardware threads. 99

List of Tables

2.1 Roman digits and their values. 14
2.2 Factorial system and its radixes, place-values and digits for the seven first positions. 17
2.3 Encoding a permutation using a Lehmer code. 27
2.4 Decoding a permutation represented by a Lehmer code. 28

3.1 Comparison of serial IVM and LL B&B algorithms in terms of maximum memory. 49
3.2 Comparison of the size of IVM and the average size of LL when solving the ten instances defined with 20 jobs. 50
3.3 Comparison of the size of IVM and the average size of LL when solving the ten instances defined with 50 jobs. 50
3.4 Comparison of serial IVM and LL-based B&B algorithms in terms of the CPU time used for the management of the pool when solving the instances defined with 20 jobs. 52
3.5 Comparison of serial IVM and LL-based B&B algorithms in terms of the CPU time used for the management of the pool when solving the instances defined with 50 jobs. 52

4.1 Number of explored subproblems for 20x20 instances when initialized with the optimal solution 73
4.2 Number of explored subproblems for 50x10 instances when initialized with the optimal solution 73
4.3 Comparison of IVM-B&B and LL-B&B with 16 threads for 20 jobs on 20 machines with an initialization to optimum in terms of speedup and sharing events. 76
4.4 Comparison of IVM-B&B and LL-B&B with 16 threads for 20 jobs on 20 machines in terms of memory usage. 78
4.5 Comparison of IVM-B&B and LL-B&B with 16 threads for 20 jobs on 20 machines in terms of CPU Time. 79

5.1 Hardware execution platform 93
5.2 Exploration time (in seconds) for solving Flow-Shop instances *Ta021-Ta030* initialized at optimal solution 95

5.3 Comparison between the linked-list and the IVM-based B&B in terms of elapsed CPU time when solving instances with 240 threads 98

A.1 Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of memory. 113
A.2 Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of memory. 113
A.3 Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of CPU Time. 114
A.4 Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of CPU Time. 114
A.5 Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to infinity in terms of CPU Time. 115
A.6 Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of CPU Time. 115

B.1 Coding of a permutation using the inversion table code. 118
B.2 Decoding a permutation represented by a code obtained with the inversion table. 120

Contents

1 Introduction **1**

2 Permutation-based optimization problems **7**

2.1 Introduction 7

2.2 Permutation optimization problems 8

2.2.1 Traveling-salesman problem 8

2.2.2 Quadratic assignment problem 9

2.2.3 Job-Shop problem 11

2.2.4 Permutation Flow-Shop problem 11

2.3 Factorial number system 12

2.3.1 Definition of number system 13

2.3.2 Definition of factorial number system 16

2.3.3 Operations based on decimal system 17

2.3.4 Operations based on factorial system 20

2.4 Handling permutations with factorial numbers 22

2.4.1 Basic concepts 22

2.4.2 Permutation to factorial number 25

2.4.3 Factorial number to permutation 27

2.5 Conclusion 29

3 Serial IVM-based B&B **31**

3.1 Introduction 31

3.2 Conventional serial B&B 32

3.2.1 Algorithm description 32

3.2.2 LL data structure 35

3.2.3 B&B operators 37

3.3 IVM-based serial B&B 39

3.3.1 IVM data structure 40

3.3.2 IVM advanced techniques 42

3.3.3 Revisited B&B operators 44

3.4 Experimentation 46

3.4.1 Experimental settings 46

3.4.2 Memory evaluation 48

3.4.3 CPU Time evaluation 51

3.5 Conclusion 53

4 Multi-core IVM-based B&B 55

4.1 Introduction 55
4.2 Parallel models for B&B algorithms 56
4.2.1 Multi-parametric parallel model 56
4.2.2 Parallel evaluation of bounds model 57
4.2.3 Parallel evaluation of a bound model 58
4.2.4 Parallel tree exploration model 58
4.3 Work stealing strategies for multi-core IVM-based B&B 59
4.3.1 WS-based B&B implementation 59
4.3.2 Coalesced work units 61
4.3.3 Dividing one factoradic interval into two intervals 63
4.3.4 Victim selection and granularity policies 67
4.4 Experimentation 72
4.4.1 Experimental settings 72
4.4.2 Strategy and granularity policies evaluation 75
4.4.3 Memory evaluation 77
4.4.4 CPU time evaluation 78
4.5 Conclusion 80

5 Many-core IVM-based B&B 81

5.1 Introduction 81
5.2 Coprocessor-accelerated B&B: the general design 82
5.3 GPU-based implementation of B&B 84
5.3.1 Parallelization on GPU 84
5.3.2 Parallelization of B&B for GPU 86
5.4 MIC-based implementation of B&B 88
5.4.1 Parallelization on Intel Xeon Phi 88
5.4.2 Parallelization of B&B for Intel Xeon Phi 91
5.5 Experimentation 92
5.5.1 Hardware and software testbed and parameter setting 93
5.5.2 Experimental results 94
5.6 Conclusion 96

6 Conclusions and perspectives 101

Bibliography 105

A Other experimental results **113**

B Inversion table **117**

CHAPTER 1

Introduction

The Ph.D thesis presented in this manuscript has been realized within the DOLPHIN [1] research group from Inria Lille-Nord Europe, CNRS/CRIStAL and Université Lille 1. The thesis has been funded by Inria and took place in the premises of Maison de la Simulation at Saclay.

In practice, many problems can be modeled as combinatorial optimization problems (COPs) which consist in maximizing or minimizing a cost function under some constraints. In this thesis, we focus on permutation-based COPs such as the Quadratic Assignment Problem (QAP), the Traveling Salesman Problem (TSP), the Flow-Shop problem, and so on. For those problems permutations are used to code/represent the candidate solutions. For instance, for a scheduling problem such as Flow-Shop the numbers of a permutation coding a given solution may designate jobs. For example, a solution coded as a permutation $(3, 4, 2, 1)$ means that the 4 jobs should be executed in the following order: job 3, then jobs 4 and 2, and finally job 1.

Solving permutation COPs consists in finding an optimal permutation among a large finite set of possible permutations. A wide range of these problems are known to be NP-hard. Therefore, metaheuristics are often used to solve them especially when the considered instances are very large [Mehdi 2011, Luong 2011]. Although these approximate methods allow to reduce the size of the explored search space and to speed up its exploration they fail in general to provide exact solutions. Conversely, exact optimization methods allow to find optimal solutions with proof of optimality. The branch-and-bound (B&B) algorithm is one of the most used exact methods to solve permutation COPs: QAP in [Mautor 1994a], TSP in [Carneiro 2011], Flow-Shop in [Melab 2012], and so on. This tree-based algorithm is based on an implicit enumeration of all the feasible solutions of the problem to be solved using four operators: branching, bounding, selection and pruning. It proceeds in several iterations during which it recursively decomposes the problem into subproblems and progressively improves the best solution found so far. The

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

pool of generated and not yet examined subproblems are kept into some data structure initialized to the original problem. The proposition of an efficient data structure, that is considered in this thesis, is one of the major challenging issues that should be addressed for an efficient implementation of B&B [Mans 2006]. At each iteration, a subproblem is selected from this data structure, according to some strategy (depth-first, best-first,...), using the selection operator. The branching operator performs its decomposition into other subproblems. The bounding operator calculates a lower bound of each generated subproblem. Each subproblem having a lower bound greater than the best solution found so far is discarded using the pruning operator.

The lower bound-based pruning mechanism is a key idea of the B&B algorithm that significantly reduces the number of explored nodes. However, B&B remains time-intensive when it comes to solve very large problem instances. Therefore, only small or moderately-sized instances are often solved in practice in a reasonable amount of time using a single processing core [Garey 1976]. For this reason, over the last decades, parallel computing has been revealed as an attractive way to deal with larger instances of COPs including permutation-based ones. The design and implementation of parallel B&B is strongly influenced by the target execution platform [Bader 2005]. To take into account the associated hardware characteristics, different parallelization approaches have been proposed in the past for Massively Parallel Processors (MPP) [Allen 1997], Networks or Clusters of Workstations [Quinn 1990, Dowaji 1995, Tschöke 1995] and Shared Memory or SMP machines [Mans 2006, Casado 2008]. Recently, the parallelization of B&B has been revisited for multi-core (clusters of) processors [Barreto 2010] and Graphics Processing Units (GPU) [Carneiro 2011, Lalami 2012] and their combination [Chakroun 2013a, Vu 2014]. MIC coprocessors are becoming more and more used in High Performance Computing and serious competitors of GPU accelerators. Indeed, the first ranked machine of Top500 (June 2015) includes Xeon Phi coprocessors. However, to the best of our knowledge the parallelization of B&B for MIC architectures has not yet been considered in the literature. In addition to multi-core and GPU accelerators, the parallelization of B&B on MIC coprocessors is also considered in this thesis.

On the other hand, the focus is put here on parallel B&B algorithms applied to permutation COPs. Without loss of generality, the Flow-Shop Scheduling Problem (FSP) is considered as a case study. The problem consists in scheduling a pool of jobs on a set of machines with respect to two constraints: the jobs are processed on all the machines in the same order and each machine cannot process more than one job at a time. The objective is to find a processing order on each machine such that the time required to complete all

jobs is minimized. The lower bound function considered in our work is that proposed by Johnson in [Johnson 1954] for two machines and generalized in [Lenstra 1978] to more than two machines. The overall major objective of this thesis is twofold: (1) proposing a new data structure for an efficient storage and management of the subproblems generated during the resolution of a permutation COP ; (2) according to this new data structure, revisiting the design and implementation of serial and parallel B&B algorithms for multi-core processors and coprocessors for solving challenging permutation COPs. Rethinking B&B algorithms for multi-core processors and coprocessors raises several design and implementation challenges. These challenges and associated contributions are summarized in the following:

- As for other algorithms, data structures play a major role in the performance of a B&B algorithm [Mans 2006] and irregular applications in general [Acar 2013]. Indeed, the efficiency of its four operators depends strongly on the data structure they act on. Several abstract data structures have been proposed in the literature for B&B algorithms including priority queues for best-first B&B, stacks for depth-first B&B, and so on. Moreover, different implementations have been proposed for their efficient management. For instance, in [Mans 2006] two implementations of the priority queue have been investigated: a skew heap (self-adjusting form of heap) and funnel tree. More generally, efficient implementations of the priority queue are provided in programming languages such as C++ and Java. Another popular data structure provided in those languages is the **d**ouble-**e**nded **que**ue (or deque). A deque is a sequence container with dynamic size that can be expanded or contracted on both ends (either its front or its back). This data structure is important and often used in the multi-core context using the work stealing mechanism in which stealing operations are performed from the back of the queue. This is why it is considered in our work as a basis for comparison with our new data structure. In the literature, deque is also called head-tail linked list. In the rest of this manuscript, it is abbreviated by *LL* which stands for Linked-List. LL is a generic data structure which makes it frequently used. However, we believe that it is important to take into account the specificity of the COPs to be tackled to define more efficient data structures. This is crucial especially when the pool of subproblems generated during their resolution is very big which is the case for permutation COPs. Indeed, the complexity of a single-permutation COP of size n is $n!$ which is very high for large problem instances. For example, for a Flow-Shop with 50 jobs to be scheduled on 20 machines, the number of candidate permutations is 3×10^{64} and the number of potential subproblems to be solved is about 10^{64}. In this thesis, we propose a new

data structure called Integer-Vector-Matrix (or IVM) dedicated to permutation COPs. Consequently, we redefine the B&B operators acting on it at execution. Although IVM is less generic than LL it is more efficient in terms of storage and management of the pool of subproblems generated during the resolution of a COP.

- B&B algorithms applied to permutation problems are highly irregular due to the bounding operator [Chakroun 2013c]. Indeed, the explored tree is unpredictable and highly irregular in size and shape from a run to another of the same problem instance. Therefore, work stealing-based work pool parallel model is well-suited for their parallelization as for irregular applications in general [Acar 2013]. In most existing works related to multi-core B&B, the work pool model is implemented using concurrent data structures [Mans 2006, Chakroun 2013a]. In such approach, each thread continuously picks a subproblem from the shared pool and the generated subproblems are returned back to be inserted in the pool. The major problem with this approach is that the use of concurrent data structures is limited in terms of efficiency and scalability. Instead, private data structures are highly recommended [Shavit 2011, Acar 2013]. In this thesis, we revisit the work stealing paradigm on multi-core processors using private IVM and private LL. The challenge here is twofold: (1) Defining the work units and the way they are communicated ; (2) Defining victim selection and granularity policies to manage the stealing operations performed on these work units. We first propose an efficient coalesced coding of work units using factoradic-based intervals. Then, we investigate various work stealing strategies based on different victim selection and granularity policies.

- An experimental study performed in [Chakroun 2013b] has shown that the time spent by the B&B evaluating the lower bounds of the generated subproblems is on average between 98% and 99% of its total execution time. This result demonstrates that the bounding operator needs massively parallel computing. On the other hand, coprocessors including GPUs and MIC allow to boost the performance of traditional processors through the combination of a larger number of processing cores, vector-SIMD processing and multi-threading. These specific features raise several issues including the optimization of data transfer between the processor and its coprocessor, vectorization, memory optimization, etc. In this thesis, having in mind these issues, we revisit the parallel bounding model combined with the parallel tree exploration model of B&B algorithms for GPU accelerators and Intel MIC Xeon Phi coprocessors.

This thesis is organized into six chapters. Chapter 2 introduces the factorial number system for permutation problems. This later is used to define our IVM data structure. We first give some examples of permutation COPs: TSP, QAP, Job-Shop and Flow-Shop. Then, we recall the factorial number system and remind to the reader some basic concepts in number systems, such as positional numbering systems and mixed radix systems. Finally, we detail the two main techniques used to convert a permutation into a factorial number, and to transform a factorial number into a permutation.

Chapter 3 describes the conventional serial B&B algorithm, its associated four operators, and the implementation of its pool of subproblems using the LL data structure. Then, the implementation of the pool using our newly proposed IVM data structure is detailed. Finally, some experimental results are reported comparing the performance of the IVM-based B&B to the LL-based B&B in terms of memory and CPU time usages.

Chapter 4 presents the different parallel models of B&B algorithms and their investigation for different parallel hardware architectures in some related works. Then, B&B is revisited using the IVM data structure and the work stealing paradigm. Five different strategies are presented, experimented and compared considering the IVM and LL data structures.

In Chapter 5, we first present the general design of the coprocessor-accelerated B&B. Then, we describe the implementation of the GPU-accelerated and Phi-accelerated approaches. After that, we report some experimental results comparing the two coprocessor-based many-core implementations and their multi-core implementation counterpart.

Finally, some concluding remarks are drawn in Chapter 6. In addition, we propose some future extensions of the proposed approaches and some perspectives related to the evolution of the context of High Performance Computing and to the generalization of the proposed approaches to other permutation COPs and other B&X tree-based algorithms.

CHAPTER 2

Permutation-based optimization problems

Contents

2.1 Introduction **7**

2.2 Permutation optimization problems **8**

2.2.1 Traveling-salesman problem 8

2.2.2 Quadratic assignment problem 9

2.2.3 Job-Shop problem 11

2.2.4 Permutation Flow-Shop problem 11

2.3 Factorial number system **12**

2.3.1 Definition of number system 13

2.3.2 Definition of factorial number system 16

2.3.3 Operations based on decimal system 17

2.3.4 Operations based on factorial system 20

2.4 Handling permutations with factorial numbers **22**

2.4.1 Basic concepts 22

2.4.2 Permutation to factorial number 25

2.4.3 Factorial number to permutation 27

2.5 Conclusion **29**

2.1 Introduction

Many industrial and economic problems are permutation combinatorial optimization problems. Solving these problems consists in finding the optimal permutation of elements among a large finite set of possible permutations. In order to find the optimal permutation, one of the used techniques is to explicitly or implicitly list all the set of possible permutations. The factorial number system is well suited for listing and enumerating these

permutations. This special enumeration system is considered as positional mixed radix system in which the numerical base varies from position to position.

The chapter is divided into three sections. Section 2.2 gives four examples of permutations problems: TSP, QAP, Job-Shop and Flow-Shop. Section 2.3 presents the factorial number system. This section reminds to the reader some basic concepts in number systems, such as positional numbering systems and mixed radix systems. Finally, Section 2.4 details the two main techniques used to convert a permutation into a factorial number, and to transform a factorial number into a permutation.

2.2 Permutation optimization problems

2.2.1 Traveling-salesman problem

In the literature, the traveling salesman problem (TSP) is well studied mostly because of its hardness and the number of its applications [Garey 1979]. The TSP as input gives a set of n cities to travel to. The cost of the trip between each pair of cities is given by a cost matrix. The objective of the problem is to find the tour of all cities that has the lowest cost while visiting each city only once. A more formal description of the problem is generally given in the form of the graph $G = (V, A)$, where $V = v_1, ..., v_n$ is the set of vertices and $A = (v_i, v_j)/v_i, v_j \in V$ is the set of edges. Each edge (v_i, v_j) has an associated cost (or weight) c_{ij}. Solving this problem requires finding a Hamiltonian circuit for which the total cost of all the arcs is minimal. A number of variants of this problem exist, such as the symmetric TSP (TSTP) where the cost of each vertex follows this property:

$$c_{ij} = c_{ji} \forall i, j \in 1, 2, ..., n \tag{2.1}$$

When traveling between two cities, the cost of the associated arc is the same no matter the direction, so the graph can be undirected.

The asymmetric TSP (ATSP) on the other hand, takes the form of a complete directed graph where the edges have a different cost (or weight) depending on the direction. The TSP can be represented as a permutation where the order in which each city is visited during the tour is viewed as a sequence which is a permutation of size n. Since this is a tour of all the cities, the last visited city is connected to the first one. Equation 2.2 gives a permutation formulation of the TSP. The cost of a valid tour is the sum of the distances between each pair of adjacent cities in the tour.

$$Minimize \sum_{i=1}^{n-1} C_{\pi(i)\pi(i+1)} + C_{\pi(n)\pi(1)} \tag{2.2}$$

2.2.2 Quadratic assignment problem

One of the hardest combinatorial optimization problems is the quadratic assignment problem (QAP). Koopmans and Beckmann introduced this problem in 1957 [Koopmans 1957], their objective was to provide a mathematical model for the allocation of indivisible economical activities. The problem consists in finding a location for plants while taking into account the cost of transportation between plants. This makes the QAP more complex than the linear assignment problem (LAP). This particular application of the QAP is called facility location.

Later, Sahni and Gonzalez [Sahni 1976] have proven that the QAP is NP-hard. Two factors have given the QAP the attention of the optimization community: the huge number of real life applications in a variety of domains and its complexity making it well known as one of the fundamental problems in optimization. The QAP is applied in various domains outlined in [Loiola 2007] where the authors propose a review of different applications and mathematical formulations as well as a complete state-of-the-art for heuristic and exact methods applied to the QAP.

The problem originally stated by Koopmans and Beckmann can be described this way: a given set of n plants (or facilities) must be assigned to a set of n locations. The distances between each pair of locations are stored in a distance matrix D = (d_{kp}), D is a square matrix of size n. The flows between the facilities (for a hospital, this can be the transportations between different clinics) are stored in a matrix F = (f_{ij}) , F is called a flow matrix. The problem is to assign each facility to a location in a way that minimizes the sum of the flows between each pair of facilities. The QAP is more complex than the LAP because not only the cost of each assignment of a facility to a location depends on the distance and the flow between two adjacent facilities, but it also depends on the rest of the assignments of a facility to a location.

Equation 2.3 shows the general mathematical formulation of the QAP, it is called the Koopmans-Beckmann formulation. The term d_{kp} represents the distance between locations k and p. The term f_{ij} represents the flow between facilities i and j. The binary variables x_{ij} represent the assignment of a facility i to a location j. When considering the cost of the act of assigning a facility to a location (e.g. installation cost), the term b_{ik} represents the cost of the assignment. The sum of the assignment cost for each facility is then added to the overall cost of a solution to QAP. Many researchers omit this term because it is a simple linear assignment that can be optimized easily.

Equation 2.4 shows the uniqueness constraints of QAP (one facility per location, one location per facility).

$$Minimize \sum_{i,j=1}^{n} \sum_{k,p=1}^{n} f_{ij}.d_{kp}.x_{ik}.x_{jp} + \sum_{i,k=1}^{n} b_{ik}.x_{ik} \ , x \in X \tag{2.3}$$

$$x \in X \equiv \begin{cases} x \in \{0,1\} \\ \sum_{i=1}^{n} x_{ij} = 1, j = 1, ..., n \\ \sum_{j=1}^{n} x_{ij} = 1, i = 1, ..., n \end{cases} \tag{2.4}$$

Later, Lawler [Lawler 1963] proposed a different formulation of the QAP, based on integer linear programming (LP). Lawler's LP formulation replaces the terms $f_{ij}.d_{kp}$ by a single term c_{ijkp} representing the cost of assigning a facility i to a location k and assigning a facility j to a location p. These costs are placed in a square matrix of size n^2 called a cost matrix. However, the permutation formulation of the QAP is more pertinent within the context of this thesis. In this case, the QAP can be described as the assignment of n facilities to n adjacent locations, the objective is then to minimize the overall cost of the assignments. It follows that a solution to the QAP can be described as a permutation π of n integers (1, 2, ..., n) which represent the facilities, whereas the positions within the permutation represent the n locations. Equation 2.5 shows this formulation.

$$Minimize \sum_{i,j=1}^{n} f_{ij}.d_{\pi(i)\pi(j)} \tag{2.5}$$

The use of permutations guarantees that the constraints of the assignment are respected.

As said earlier, Sahni and Gonzalez [Sahni 1976] have proven that the QAP is NP-hard. They argued that it is not even possible to find a polynomial f-approximation algorithm for the QAP.

Different authors have studied the practical complexity of the QAP from various points of view. Roucairol *et al.* [Mautor 1994b] studied how difficult it is for exact methods to solve the QAP. They argued that the structure of the instances is a factor in explaining the difficulty of solving the QAP for exact methods such as Branch-and-Bound. The product of the two matrices can produce many solutions of good quality whose cost is within a very narrow range. Under such conditions, the Branch-and-Bound algorithm can not eliminate many branches in the tree. Many authors use the related notion of flow dominance to explain the hardness of some of the instances. Angel *et al.* [Angel 2002] define flow dominance as: flow dominance is when a few facilities exchange a lot of materials between each other but few materials with the rest of the facilities.

2.2.3 Job-Shop problem

The Job-Shop problem (JSP) is a generalized version of the permutation Flow-Shop problem. Brucker [Brucker 2007] provides a definition of JSP. A set of n jobs $J_i, J_2, ..., J_n$ must be scheduled on a set of m machines. Every job J_i is made of n_i operations defined as $O_{i,j}$, $i = 1, 2, ..., n$ and $j = 1, ..., n_i$. There is a precedence constraint which ensures that the sequence of operations $O_{i,1}, O_{i,2}, ..., O_{i,n_i}$ of every job has to be processed in this specific order on all machines. For operation $O_{i,j}$ on machine $M_{i,j}$ with $j = 1, ..., n_i$, we define $p_{i,j}$ as the processing time of this operation. We assume that each operation $O_{i,j}$ will be processed on a different machine $M_{i,j}$, meaning that $M_{i,j} \neq M_{i,j+1}$ for $j = 1, ..., n_{i-1}$. The objective of JSP is to find a valid schedule which minimizes the completion time or the makespan C_{max} of the last operation in the schedule.

The use of multi-permutations makes it possible to have a permutation representation of the JSP [Bierwirth 1996, Ponsich 2010]. Each permutation is a representation of the sequence of operations on one machine. The total number of operations is equal to $n \times m$, n being the number of jobs and m being the number of machines. However, Bierwirth points out in [Bierwirth 1996] that due to the different constraints, it is not possible to find a representation of the JSP that uses standard permutations without including invalid solutions in the coding. For example, using a multi-permutation to describe the problem as described earlier, it is possible that two jobs scheduled on two different machines violate the precedence constraint. In order to overcome this obstacle, some authors use weight penalties to progressively discard invalid solutions from the search, or use reparation to transform an invalid solution into a valid one. However, the search can be disrupted if this happens too often. [Bierwirth 1996] provides a new representation for the JSP. This new representation makes use of a single permutation which uses repetition to represent the sequencing of all the operations on all the machines.

The use of this representation means that each solution requires a phase of decoding (or reading) before it is evolved. Using this representation, each job number appears m times within the permutation, i.e. as many times as there are operations that belong to this job. This representation is called an indirect representation. There is no decoding step required when using classic neighborhood and genetic operators, which means that the permutation should be read according to the properties of the problem.

2.2.4 Permutation Flow-Shop problem

In manufacturing environments, it is common to find Permutation Flow-Shop Scheduling Problems [Bonney 1976, King 1980, Allahverdi 1999] where n jobs have to be processed on m machines where the goal is to optimize an objective function. The objective of the

FSP 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 FSP 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. Within the context of this thesis, each reference to the FSP is actually a reference to the permutation FSP [Allahverdi 1999, Hejazi 2005]. Using Johnson's algorithm [Johnson 1954], it is possible to find an optimal schedule for the FSP in $O(n \log n)$ steps when $m = 2$. The problem is NP-hard when $m \geq 3$ [Garey 1976]. This is why it is often tackled using metaheuristics [Basseur 2005] for solving large problem instances.

These are the constraints that a valid FSP solution should satisfy:

- A machine can not start processing a job before the preceding machines have finished processing 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.

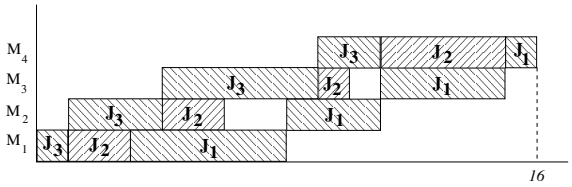
Figure 2.1 shows an example of an FSP instance where $n = 3$ and $m = 4$, it also shows the optimal solution.

2.3 Factorial number system

The proposed IVM data structure (presented in Chapter 3) and the coding of work units in the multi-core context (presented in Chapter 4) are based on the factorial number system. The definition of this system is quite different compared to other known systems such as decimal and binary systems. Subsection 2.3.1 reminds the main concepts used to define any number system. Subsection 2.3.2 presents the factorial number system. Then, Subsection 2.3.3 and Subsection 2.3.4 explain the implementation of some arithmetic operations in the factorial system. The four arithmetic operations used in our work are addition, subtraction, division, and comparison. These factoradic-based operators are used in Chapter 4 for the factoradic interval-based work stealing strategies.

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

Processing Times



Optimal Solution

Figure 2.1: Illustration of a permutation FSP where $n = 3$ and $m = 4$. The table shows the processing times of the jobs on each machine. The Gantt diagram shows the optimal solution for this particular instance.

- **Comparison:** This operation is relatively easy to implement. It consists of comparing all the positions of the two operands starting with the most significant position (i.e. from left to right).

- **Addition:** This operation is directly implemented in the factorial system. Subsection 2.3.4 explains the method used to perform this operation.

- **Subtraction:** The algorithm of this operation is similar to the one used for the addition. Subsection 2.3.4 also explains the implementation of this operation.

- **Division:** Unlike addition and subtraction, the division is more difficult to implement for factorial numbers. In our work, division is implemented by converting factorial numbers to decimal numbers, thereafter, by performing division of these decimal numbers, and finally, by converting the result of the division to a factorial number. Subsection 2.3.3 explains the implementation of division using decimal system.

2.3.1 Definition of number system

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 2.3.2.

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 2.1.

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

Table 2.1: 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 [Golomb 1966].

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 this 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 first and second positions is 60 and the radix of the third 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 (2.6).

$$(a_{n-1}...a_0)_R = a_{n-1}R^{n-1} + ... + a_1R^1 + a_0R^0 \tag{2.6}$$

Since Equation (2.6) 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.

$$0.a_1a_2...a_n = a_{-1}R^{-1} + a_2R^{-2} + ... + a_nR^{-n} \tag{2.7}$$

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 (2.8).

$$a_{n-1}...a_0 = (...((a_{n-1}\ R_{n-1}) + a_{n-2})R_{n-2} + ... + a_1)R_0 + a_0 \tag{2.8}$$

For example, 10 hours 30 minutes 50 seconds is interpreted as

$$10 : 30 : 50 = ((10 \times 24 + 30) \times 60 + 50\ seconds \tag{2.9}$$

2.3.2 Definition of 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 [Knuth 1998] while the French name *numération factorielle* is first used in 1888 [Laisant 1888]. The term "factoradic" appears to be much more recent [McCaffrey 2003]. General properties of mixed radix number systems also apply to the factorial system.

As explained in Table 2.2, the ith digit from the right has base i and the place-value $i!$. Therefore, the ith digit must be less than i. And in order to compute the value of a number, the value of the ith 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 2.2: 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 thesis 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 correspondent place-values are therefore $1/(0!)$, $1/(1!)$, ..., $1/(n!)$, etc. In our work, factorial fractional numbers are not used.

2.3.3 Operations based on decimal system

This subsection explains the used methods, in our work, to perform the division operation in the factorial system. Unlike other operations, the division is hardly feasible directly in the factorial system. Another way to do this operation is to convert the dividend and the divisor to the decimal system, to perform the division in the decimal system, and thereafter to convert the result to the factorial system. So this subsection explains how to convert a factorial number to a decimal number, and how to convert a decimal number to a factorial number.

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 division as shown in Equation (2.10).

$$\begin{aligned} 349 &= 1 \times 349 + 0 \\ 349 &= 2 \times 174 + 1 \\ 174 &= 3 \times 58 + 0 \\ 58 &= 4 \times 14 + 2 \\ 14 &= 5 \times 2 + 4 \\ 2 &= 6 \times 0 + 2 \end{aligned} \tag{2.10}$$

Euclidean division is the operation of division of two integers, which produces a quotient and a remainder. Each line of Equation (2.10) 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 (2.11), is $F = (R_n ... R_2 R_1)_! = 242010$.

$$\begin{aligned}
349 &= 1 \times 349 + 0 \\
&= 1 \times (2 \times 174 + 1) + 0 \\
&= 1 \times (2 \times (3 \times 58 + 0) + 1) + 0 \\
&= 1 \times (2 \times (3 \times (4 \times 14 + 2) + 0) + 1) + 0 \\
&= 1 \times (2 \times (3 \times (4 \times (5 \times 2 + 4) + 2) + 0) + 1) + 0 \\
&= 1 \times (2 \times (3 \times (4 \times (5 \times (6 \times 0 + 2) + 4) + 2) + 0) + 1) + 0 \\
&= 2\ 5! + 4\ 4! + 2\ 3! + 0\ 2! + 1\ 1! + 0\ 0! \\
&= (242010)_!
\end{aligned} \tag{2.11}$$

Algorithm 1 explains the used method to perform this conversion.

Algorithm 1 DECIMAL-TO-FACTORIAL(D)

```
Place ← 1
while D ≠ 0 do
    F_{i-1} ← D mod i
    D ← D div i
    i ← i+1
end while
return F
```

Factoradic to decimal:

Converting a factorial number to a decimal number is simpler. Let $(R_{n-1} ... R_1 R_0)_!$ 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 ← 1
2: D ← 0
3: for i ← 1 to n do
4:     Place-value ← Place-value × i
5:     D ← D + F_i × Place-value
6: end for
7: return D
```

2.3.4 Operations based on factorial system

It is possible to use the conversion to the decimal system for implementing the addition and subtraction. However, this results in loss of performance. The other technique is to directly implement these operations into the factorial system. This subsection first explains the implementation of the addition and then the implementation of the subtraction.

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 (2.12).

$$\begin{aligned} C_i &= A_i\ i! + B_i\ i! \\ &= [A_i + B_i]i! \\ &= [(i+1) - (i+1) + (A_i + B_i)]i! \\ &= (i+1)i! + [-(i+1) + (A_i + B_i)]i! \\ &= (i+1)! + [(A_i + B_i) - (i+1)]i! \end{aligned} \tag{2.12}$$

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 (2.13) is always true.

$$[(A_i \leq i)\ and\ (B_i \leq i)\ and\ (A_i + B_i > i)] \Rightarrow 0 \leq [(A_i + B_i) - (i+1)] \leq i \tag{2.13}$$

Equation (2.14) gives the value of C_i for any values of A_i and B_i.

$$C_i = \begin{cases} (A_i + B_i) & If\ (A_i + B_i) \leq i \\ (A_i + B_i) - (i+1) & Otherwise \end{cases} \tag{2.14}$$

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)

```
for i ← 0 to (n-1) do
    C_i ← C_i + A_i + B_i
    if C_i > i then
        C_i ← A_i − (i + 1)
        Ci + 1 ← 1
    end if
end for
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 (2.15).

$$\begin{aligned} C_i &= A_i\ i! - B_i\ i! \\ &= [A_i - B_i]i! \\ &= [-(i+1) + (i+1) + (A_i - B_i)]i! \\ &= [-(i+1)]i! + [(i+1) + (A_i - B_i)]i! \\ &= [-1](i+1)! + [(i+1) + (A_i - B_i)]i! \end{aligned} \tag{2.15}$$

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 (2.16) is always true.

$$[(A_i \leq i)\ and\ (B_i \leq i)\ and\ (A_i < B_i)] \Rightarrow 0 \leq [(A_i - B_i) + (i+1)] \leq i \tag{2.16}$$

Equation (2.17) gives the value C_i for any values of A_i and B_i.

$$C_i = \begin{cases} (A_i - B_i) & If\ (A_i - B_i \geq 0) \\ (A_i - B_i) + (i+1) & Otherwise \end{cases} \tag{2.17}$$

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)

```
for i ← 0 to (n − 1) do
    C_i ← C_i + A_i − B_i
    if C_i < 0 then
        C_i ← C_i + (i+1)
        C_{i+1} ← -1
    end if
end for
return C
```

2.4 Handling permutations with factorial numbers

2.4.1 Basic concepts

Representation of a permutation:

In the remainder of this section, three representations are used to describe a permutation π.

- The element-based representation: $\pi = \pi_0\pi_1...\pi_{n-1}$. The positions are implicitly assumed sorted. The presentation gives therefore the position of each element. For example, the element 7 of Equation (2.18) is scheduled at the position 2.

$$\pi = 527038614 \tag{2.18}$$

- The set-based representation: $\pi = \{(0, \pi_0), (1, \pi_1), ...(n-1, \pi_{n-1})\}$. In this representation, the permutation is described as a set of pairs. For each pair, the first part gives the position, and the second part gives the element of this position. The example of Equation (2.18) can be written by Equation (2.19).

$$\pi = \{(0,5), (1,2), (2,7), (3,0), (4,3), (5,8), (6,6), (7,1), (8,4)\} \tag{2.19}$$

- The position-based representation: It is possible to represent a permutation by assuming that the elements are implicitly sorted. Therefore, it is necessary to give the position of each element. The examples of Equation (2.18) and (2.19) can be written using Equation (2.20).

$$\pi = 371480625 \tag{2.20}$$

To facilitate the understanding of the methods of this section, it is therefore important to see a permutation as a bijective relationship between the components of a vector of elements and the components of a vector of positions.

Inversions:

Lehmer code (explained in this chapter) and inversion table (explained in Appendix B) are both based on the inversion concept. Let assume π a permutation. An inversion in a permutation $\pi = \pi_0\pi_1...\pi_{(n-1)}$ is a pair (π_i, π_j) which satisfies the two conditions stated in Equation (2.21).

$$(\pi_i, \pi_j) \; is \; an \; inversion \iff (i < j) \; and \; (\pi_i > \pi_j) \tag{2.21}$$

In this section,

- $Inversions(\pi)$ indicates all the inversions of a permutation π;

$$Inversions(\pi) = \{(\pi_i, \pi_j)/(i < j) \; and \; (\pi_i > \pi_j)\} \tag{2.22}$$

- $Inversions(\pi_i, \pi)$ indicates all the inversions of a permutation π where π_i appears at the left side of the pairs;

$$Inversions(\pi_k, \pi) = \{(\pi_i, \pi_j) \in Inversions(\pi)/i = k\} \tag{2.23}$$

- $Inversions(\pi, \pi_i)$ indicates all the inversions of a permutation π where π_i appears at the right part of the pairs;

$$Inversions(\pi, \pi_k) = \{(\pi_i, \pi_j) \in Inversions(\pi)/j = k\} \tag{2.24}$$

If all elements of a permutation are ordered, then this permutation contains no inversions. A permutation with no inversions is called identity permutation. Equation (2.25) gives some examples to illustrate all these concepts.

$$\begin{aligned}
&Inversions(527038614) = \{(5,2),(5,0),(5,3),(5,1),(5,4),(2,0),(2,1),...,(6,4)\}\\
&Inversions(5,527038614) = \{(5,2),(5,0),(5,3),(5,1),(5,4)\}\\
&Inversions(527038614,0) = \{(5,0),(2,0),(7,0)\}\\
&Inversions(012345678) = \emptyset
\end{aligned} \tag{2.25}$$

Basic vector operations:

In the explanation of the two methods of the conversion between factorial numbers and permutations, three basic operations for handling vectors are used.

- The first operation, called VECTOR-INITIALIZE-ZERO accepts as input a vector, and initializes each position of the vector by zero. This operation is used in Subsection 2.4.2

- The second operation, called VECTOR-INITIALIZE accepts as input a vector, initializes each position of this vector by a number equal to its position, and returns the initialized vector. This operation is used in Subsection 2.4.3

- The third operation, called VECTOR-SELECT, receives as input a vector v and a position i, reads the integer r located at position i of vector v (i.e. $r = v_i$), shifts all the elements located at the right of i with one position towards the left, and returns the integer r. This operation is used in Subsection 2.4.3

Algorithm 5 Vector and some of its basic operations

```
1: procedure VECTOR-INITIALIZE-ZERO(V)
2:     for i ← 0 to (N-1) do
3:         V_i ← 0
4:     end for
5: end procedure
6:
7: procedure VECTOR-INITIALIZE(V)
8:     for i ← 0 to (N-1) do
9:         V_i ← i
10:    end for
11: end procedure
12:
13: procedure VECTOR-SELECT(V,I)
14:    r ← V_I
15:    for i ← I to (N-1) do
16:        V_i ← V_{i+1}
17:    end for
18:    return r
19: end procedure
```

2.4.2 Permutation to factorial number

When encoding and decoding a permutation, it is important that the used code has to be compact and the coding and decoding operations have to be fast. With a set of n elements, it is possible to get $n!$ permutations, and from 0 to $(n! - 1)$, there are n numbers. So the idea is to associate for each of these permutations one and only one number. In other words, the objective is to encode each permutation with one and only one number. There are two main methods, almost equivalent, which are used to code a permutation as a factorial number. The first is based on the Lehmer code (explainted in this section) and the second on the inversion table (explained in Appendix B).

Lehmer code, introduced by Derrick Lehmer [Lehmer 1960], allows to encode a permutation as a factorial number. The example of Table 2.3 shows how to transform a permutation $\pi = \pi_0\pi_1...\pi_8$ in order to obtain its Lehmer code $L = (L_0L_1...L_8)_!$. This example assumes that the permutation π is equal to 527038614. In the figures of this table, the elements of the permutation are represented by black circles, positions by blue triangles topped with a bow, and the obtained Lehmer code by a red rectangle. Each of the nine

cells of the table describes the obtained value L_i of a Lehmer code position.

As shown in the first cell of the table, to obtain the first code L_0, it is necessary to count the number of inversions where the element $\pi_0 = 5$ appears on the left side of a pair. In this example and as shown in Equation (2.26), there are 5 inversions of this type. This means that L_0 is equal to 5.

$$Inversions(5, 41032) = \{(5,2),(5,7),(5,0),(5,3),(5,4)\} \tag{2.26}$$

In the same way, L_1 is equal to 2 since π_1 is equal to 2 and there are 2 pairs of inversions where 2 appears at the left side of the pair. It is possible also to deduce the other values L_i as shown in Equation (2.27). Therefore, the Lehmer code of the permutation $\pi = 527038614$ is equal to $L = (525013200)_!$.

$$L_i = |Inversions(\pi_i, \pi)| \tag{2.27}$$

The figure of the first cell shows that the number of inversions associated to π_0 can not exceed 8 since there are only 8 elements at the right of π_0. In the same way, the number of inversions associated to π_1 can not exceed 7. Those associated to π_2 can not exceed 6, etc. So the general rule stipulates that the associated number of inversions for π_i can not exceed $8 - i$. Since the value L_i is always smaller than $8 - i$, the obtained Lehmer code is a factorial code.

Algorithm 6 describes the used method for obtaining the Lehmer code of any permutation. This algorithm has a complexity $O(n\ log(n))$ and contains two loops. The outer loop allows the algorithm to point the code i to compute, and the inner loop counts the number of inversions where the element π_i appears at the left side of an inversion of a permutation π.

Algorithm 6 PERMUTATION-TO-LEHMER(π)

```
1: Vector-initialize-zero(L);
2: for i ← 0 to (N-1) do
3:     for j ← (i+1) to (N-1) do
4:         if π_i > π_j then
5:             L_i ← L_i + 1;
6:         end if
7:     end for
8: end for
9: return L
```

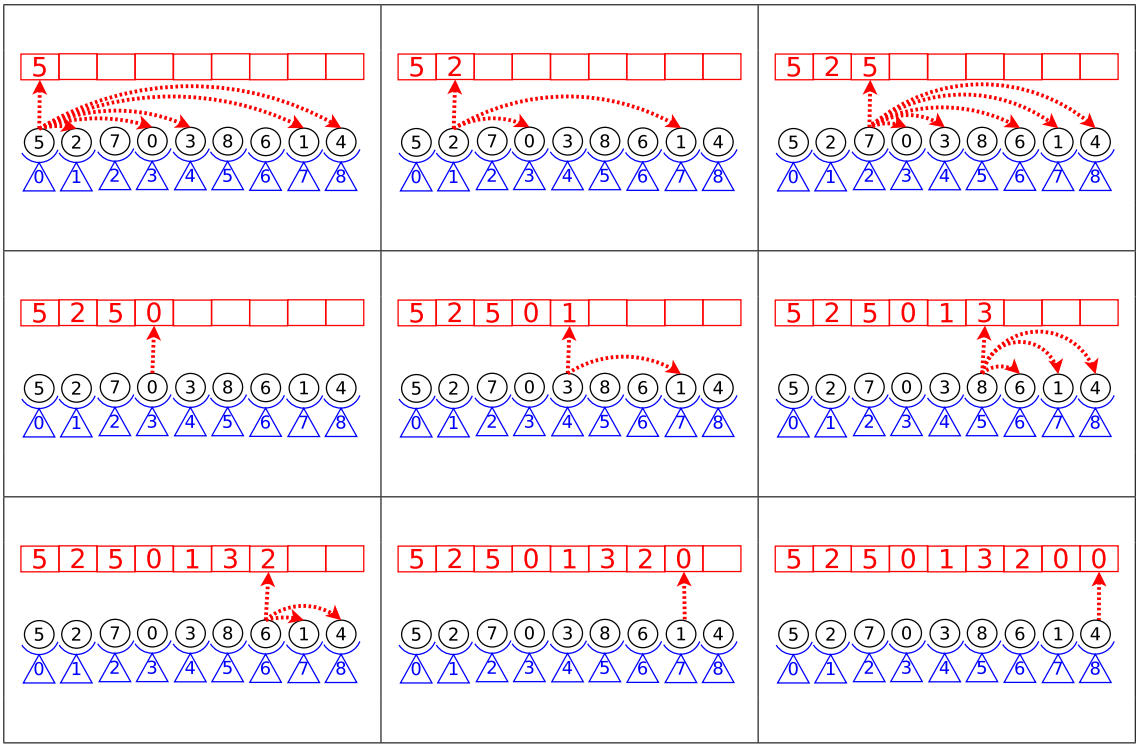


Table 2.3: Encoding a permutation using a Lehmer code.

2.4.3 Factorial number to permutation

Figures of Table 2.4 explain the procedure for obtaining a permutation π from its Lehmer code $L = (525013200)_!$. In these figures, a position is represented by a blue triangle, and an element is illustrated by a black circle.

This algorithm does not operate on the vector of positions but on the vector of elements. In this algorithm, the Lehmer code L_0 is decoded first to find π_0, then L_1 to find π_1, ..., until decoding L_8 to find π_8. In other words, decoding L_i allows to find the element π_i of the permutation π.

At the beginning, L_0 is equal to 5. The decoding is thus performed by taking the element that is located at position 5 of the vector of elements, put this element at position 0 of the permutation π, and shift with one position to the left all the elements which are at the right of the position 5 of the vector of elements. Decoding a code L_i consists therefore to take the element which is located at position i of the vector of elements, put this element at the position i of the permutation π, and shift with one position to the left all the elements which are at the right of the position i of the vector of elements. Decoding the Lehmer code $L = (525013200)_!$ provides therefore the permutation $\pi = 527038614$.

The method of decoding a Lehmer code is summarized by Algorithm 7. The algo-

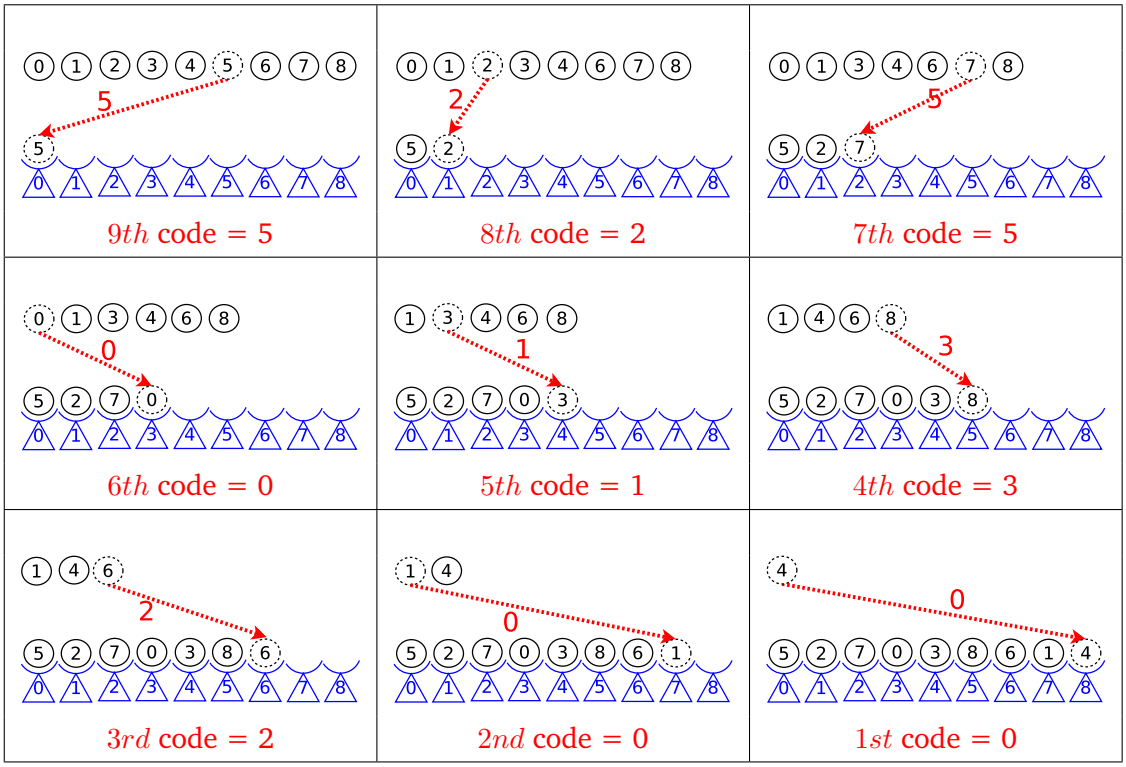


Table 2.4: Decoding a permutation represented by a Lehmer code.

rithm receives as input a factorial number corresponding to a Lehmer code, and returns as output the permutation which corresponds to this code. The algorithm begins by initializing a vector of elements called $Items$. This initialization is performed by the function VECTOR-INITIALIZE explained in Subsection 2.4.1. The vector $Items$ contains therefore all the elements sorted. Then, the algorithm performs the generation of the permutation π starting with the element π_0 and ending with the element π_{n-1}.

Generating the element of position i is done by decoding the Lehmer code L_i. The element of position L_i of the vector $Items$ is read using the function VECTOR-SELECT. As explained in Subsection 2.4.1, this function also performs a left shift with one position. The element read by this function corresponds to the element of position i of the permutation. This algorithm also has a complexity $O(n\ log(n))$.

Algorithm 7 LEHMER-TO-PERMUTATION(L)

```
Vector-initialize(Items)
for i ← 0 to (n-1) do
    li ← L_i
    ITEM ← Vector-select(Items,li)
    π_i ← ITEM
end for
return π
```

2.5 Conclusion

Many problems encountered in combinatorial optimization are permutation problems. This chapter provides some examples of basic NP-hard problems which are often studied in the combinatorial optimization literature. These problems are the TSP, where the objective is to find the shortest route that connects a certain number of cities, the QAP, where the objective is to assign facilities to locations in order to minimize the sum of the distances multiplied by the corresponding flows, the Flow-Shop, where the objective is to find the same schedule on all machines for a set of jobs in order to minimize the processing time, and the Job-Shop, where unlike the Flow-Shop, jobs may have distinct schedules for each machine. These examples are given to illustrate the large number of permutation problems encountered in combinatorial optimization. In the rest of this report, our approaches are validated on the flow-shop problem.

As explained in this chapter, it is easy to transform permutations to factorial numbers, and factorial numbers to permutations. Indeed, the Lehmer code and the inversion table are effective methods to perform these conversions. The approaches developed in this thesis are based on seeking the optimal permutation using the factorial number system instead of searching this optimal permutation directly in the permutation space. As shown in the following chapters, we have developed approaches based on the factorial system which are more efficient than those working directly in the permutation space.

CHAPTER 3

Serial IVM-based B&B

Contents

3.1 Introduction **31**

3.2 Conventional serial B&B **32**

3.2.1 Algorithm description 32

3.2.2 LL data structure 35

3.2.3 B&B operators 37

3.3 IVM-based serial B&B **39**

3.3.1 IVM data structure 40

3.3.2 IVM advanced techniques 42

3.3.3 Revisited B&B operators 44

3.4 Experimentation **46**

3.4.1 Experimental settings 46

3.4.2 Memory evaluation 48

3.4.3 CPU Time evaluation 51

3.5 Conclusion **53**

3.1 Introduction

One of the most popular exact methods, for solving to optimality permutation combinatorial optimization problems, is the Branch-and-Bound (B&B) algorithm. This algorithm is based on an implicit enumeration of all the feasible solutions of the problem to be tackled. Building the B&B tree and its exploration are performed using four operators which work on a pool of subproblems. One of the challenging issues related to the implementation of B&B is to define an efficient data structure for the management of that pool of subproblems. In this chapter, we present a new data structure to implement the B&B pool. This data structure, called IVM which stands for Integer Vector Matrix, uses an integer, a vector and a matrix in order to explore the tree. IVM is based on factorial number systems explained in the previous chapter.

The chapter is divided into three sections. Section 3.2 describes the conventional B&B algorithm, its associated four operators, and the implementation of the pool of subproblems using LL. Section 3.3 presents the implementation of the pool using our new IVM data structure. Finally, Section 3.4 compares the performance of LL-based B&B and the IVM-based B&B in terms of memory and CPU time usages.

From now on, the LL-based Branch-and-Bound algorithm will be referred to as LL-B&B, whereas the IVM-based Branch-and-Bound algorithm will be referred to as IVM-B&B.

3.2 Conventional serial B&B

Before introducing our new approach based on IVM, this section reminds to the reader how a conventional B&B works using LL data structure. The section is divided into three subsections. The first subsection provides a general overview about the B&B algorithm, the second subsection explains the role of the pool based on LL data structure, and the third subsection details the four operators of this algorithm.

3.2.1 Algorithm description

Several exact resolution methods used in combinatorial optimization are Branch-and-Bound (B&B)-like algorithms. These methods are mainly divided into three basic variants: simple B&B, Branch-and-Cut (B&C), and Branch-and-Price (B&P). There are other B&B variants less known such as Branch-and-Peg [Goldengorin 2004], Branch-and-Win [Pastor 2004], and Branch-and-Cut-and-Solve [Climer 2006]. This list is certainly not exhaustive. It is also possible to consider divide-and-conquer algorithm as a B&B algorithm. It is enough to remove the pruning operator, explained below, from the B&B to get a divide-and-conquer algorithm. Some authors consider B&C, B&P, and the other variants as different algorithms from B&B. These authors use B&X to refer to algorithms like B&B, B&C, B&P, etc. In what follows, B&B algorithm refers to simple B&B or any other variant of this algorithm.

This algorithm is based on an implicit enumeration of all the solutions of the problem being solved. The space of potential solutions (search space) is explored by dynamically building a tree where:

- The **root node** represents the initial problem to solve.
- The **leaf nodes** are the possible solutions of this initial problem.
- And the **internal nodes** are subspaces of the total search space. Internal nodes can be also considered as subproblems of the initial problem.

The size of the subspaces is smaller and smaller as one gets closer and closer to the leaves. The construction of such a tree and its exploration are performed using four operators: branching, bounding, selection and pruning. The algorithm proceeds in several iterations. The best solution found is saved and can be improved from an iteration to another. Subproblems generated and not yet processed are kept in a pool. In the beginning, this pool contains the initial problem.

The LL version of the Branch-and-Bound algorithm is shown in Algorithm 8. The pool of Figure 3.1 is represented as a tree in order to visualize the problem/subproblem relationships between nodes, and as a matrix to facilitate the comparison with the IVM-based approach described in Section 3.3. In our work, this pool is implemented using a deque (head-tail linked-list), referred to as LL, as shown in Figure 3.2. In Figure 3.1, for instance, the node *24/13* means that job 2 is scheduled at the first position, job 4 at the second position, and jobs 1 and 3 are not yet scheduled. In this figure, strike-through nodes represent subproblems which are added into LL and selected from it. At each B&B iteration, the algorithm points to a node of the B&B pool. In the example of Figure 3.1, the algorithm is currently pointing to the solution *2314/*. Therefore, Figure 3.2 represents the state of the pool just before removing *2314/*. Before selecting this solution, LL contains five nodes, namely *3/124*, *4/123*, *24/13*, *234/1* and *2314/*.

Before having LL in this state, some operations are applied. At the beginning of the B&B, none of the four jobs is scheduled (i.e. */1234*). The node */1234* is branched/decomposed into four nodes which are *1/234*, *2/134*, *3/124* and *4/123*. In each of these nodes, one job is scheduled and the three other jobs are not yet scheduled. This example assumes that the first node *1/234* is processed or pruned, and the algorithm branches the second node *2/134*. The decomposition of this node generates three nodes, namely *21/34*, *23/14* and *24/13*. The example also assumes that the first node *21/34* is processed or pruned. Therefore, the algorithm decomposes the second node *23/14*, and obtains two new nodes which are *231/4* and *234/1*. The node *231/4* represents a simple subproblem and accepts only one solution *2314/*.

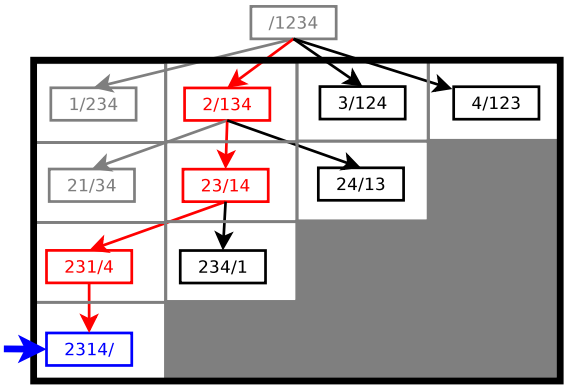


Figure 3.1: An example of a pool obtained when solving a permutation problem of size 4.

Algorithm 8 LL-based B&B algorithm.

```
procedure LL-B&B-MAIN
    best=+∞ (or −∞)
    insert(pool, root)
    while (pool is not empty) do
        subproblem p←take(pool)
        if leaf(p) then
            cost←evaluate(n)
            if best>cost then
                best←cost
            end if
        else
            cost←bound(n)
            if (best>cost) then
                subproblems ps←decompose(p)
                for all p ∈ ps do
                    insert(pool, p)
                end for
            end if
        end if
    end while
end procedure
```

3.2.2 LL data structure

The pool of subproblems:

In order to explore all the possible subproblems, the B&B algorithm must maintain a pool of subproblems that are ready to be processed. At the beginning, the B&B algorithm places the root problem in the pool of subproblems. The root problem is the subproblem where no job has been scheduled yet. So, the pool of subproblems contains [*1234*].

The selection operator then takes the first subproblem from the pool and sends it to the branching operator which will decompose this subproblem in a new set of subproblems where one more job has been scheduled. The branching operator needs to know which subproblems to add to the pool. For this, the branching operator first calls the bounding operator for each possible new subproblem in order to determine whether the new job should be placed at the beginning or at the end. In this example the branching operator will compute the bounds of the following subproblems:

- **1***234*, **2***134*, **3***124*, **4***123* as the first set of subproblems.
- *234***1**, *134***2**, *124***3**, *123***4** as the second set of subproblems.

Then the branching operator calls the elimination operator in order to remove the jobs whose bound is higher than the best known bound. Then the branching operator can finally place the new subproblems in the pool of subproblems. Let's suppose that the first set has the highest sum and is the one that is chosen, so the job will be placed at the beginning, then the elimination operator removes the first subproblem and keeps the remaining three. Those three subproblems are placed back into the pool of subproblems which now contains the following [**2***134*, **3***124*, **4***123*]. Then a new round of the B&B algorithm begins with the selection operator, which chooses the **2***134* subproblem and sends it to the branching operator.

The pool of subproblems is sorted so that the first subproblems are the ones that are deepest in the tree, this means that the new subproblems are not simply placed at the beginning or the end of the pool of subproblems, but at a location determined by a sorting algorithm which considers two criteria: first the depth, then the order of the scheduled jobs. For example if the **2***134* subproblem is decomposed into **2***14***3** and **2***13***4**, when these subproblems are placed into the pool, it will contain [**2***14***3**, **2***13***4**, **3***124*, **4***123*] where the first two subproblems are one level deeper in the tree than the last two. The B&B algorithm stops when the selection operator finds an empty pool of subproblems.

Size of the pool of subproblems:

While the size of the pool of subproblems is variable, we can find a formula that gives an upper bound for the number of subproblems in the pool of subproblems. When the best solution is unknown a worst case scenario occurs where the branching operator of the B&B algorithm has to decompose each node from the root node to the first explored leaf. This means that every time the B&B algorithm selects a node, it will be removed from the pool of subproblems, but all its children will be added to the pool of subproblems because the elimination operator will never be used as long as the best solution is unknown. This means that for a problem of size N, the maximum number of subproblems stored in the pool will be $\frac{N \times (N-1)}{2}$.

LL data structure for serial B&B:

The LL data structure used to store the subproblems is shown in Figure3.2. When selecting a new subproblem to process, the B&B algorithm takes one element from LL and uses the bounding operator in order to determine whether it should be either pruned or decomposed into subproblems. If the lower bound computed by the bounding operator is greater than the best known solution, the subproblem is eliminated by simply removing it from LL. On the other hand, if the lower bound computed by the bounding operator is smaller than the best known solution, the branching operator decomposes the subproblem. The branching operator then inserts the new subproblems into LL. LL can therefore be seen as a queue of subproblems waiting to be processed.

The subproblems in LL are not inserted randomly, or even put at the beginning or the end of LL. There is a comparison operator which ensures that the subproblems are sorted and inserted in the right position within LL. This sorting algorithm applied to LL allows to explore the tree of subproblems in a certain way. For example, if LL is sorted in such a way that the deepest subproblems are taken first by the selection operator, the tree is explored in a Depth First Search manner. The tree can also be explored in a Breadth First Search manner if LL is sorted in such a way that the deepest subproblems are taken last by the selection operator.

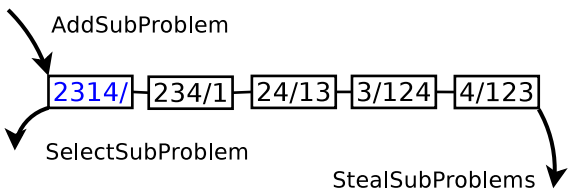


Figure 3.2: LL-based (or deque-based) representation of a pool of subproblems

Growth of the pool of subproblems:

When the branching operator inserts new subproblems into the pool of subproblems, the size of this latter will vary depending on the number of subproblems to insert into the pool. Considering that the selection operator already removed one subproblem from the pool, three cases may appear:

- If the elimination operator removed all subproblems, then no new subproblem will be inserted and the size of the pool of subproblems decreases by one.
- If the elimination operator removed all but one subproblem, then the size of the pool of subproblems remains unchanged.
- If more than one subproblems are inserted, then the size of the pool of subproblems grows by at most $N - 1$, N being the size of the problem.

The first two cases are unlikely to happen in the early stages of the B&B algorithm when the subproblems explored are still close to the root problem, especially with a higher number of jobs. It is easy to see why the pool of subproblems will grow rapidly and in the worst cases reaches its upper bound of $\frac{N \times (N-1)}{2}$ subproblems as established in Section 3.2.2. The growth of the pool can become an issue when it goes beyond the available memory, which is why a data structure which would allow for the execution of the B&B algorithm within a constant amount of memory is a desirable goal. This is the objective of the IVM data structure which is presented in Section 3.3.

3.2.3 B&B operators

Selection operator:

The selection operator often uses two main strategies, namely the *depth-first* and *best-first* strategies. In the best-first strategy, the algorithm explores the tree by expanding the most promising node chosen according to its upper or lower bounds. While in the depth-first strategy, the algorithm explores as far as possible along each branch before backtracking.

Compared to the depth-first strategy, the best-first strategy enables the algorithm to produces good solutions. With these good solutions, the algorithm can quickly prune subproblems. However, the depth-first strategy manages a smaller pool than the the best-first strategy. Therefore, the B&B uses a hybrid depth-first and best-first strategy. Our algorithm selects the deepest subproblems, and if there are several subproblems with the same depth, then the algorithm selects the best subproblem.

The used selection operator is quite simple to implement: It chooses the subproblem that is located at the top of LL. However, LL is assumed to be sorted according to depths and bound costs of the subproblems.

Branching operator:

The role of the conventional branching operator is to decompose a subproblem into new subproblems that are candidates for further exploration and to insert them back into the pool of subproblems at the right positions. The decomposition itself is accomplished by a combination of the bounding operator and the elimination operator, therefore the only concrete responsibility of the branching operator is the insertion of the new subproblems into the pool of subproblems. The time complexity for an insertion in LL is linear and depends on the number of elements of that pool. In this case, the time complexity depends on the number of subproblems in the pool of subproblems.

As shown in Subsection 3.2.2, $\frac{N\times(N-1)}{2}$ is the maximum size of the pool of subproblems, where N is the size of the problem. The sorting algorithm needs to compare two arrays of N elements to determine whether a subproblem can be inserted at a particular position, this means that the number of comparisons required for each insertion is at most $\frac{N^2\times(N-1)}{2}$. Finally there are at most N subproblems to insert into the pool, so the maximum number of comparisons required to insert all the subproblems is $\frac{N^3\times(N-1)}{2}$. The time complexity of the branching operator in the worst case is therefore $O(N^4)$.

In the B&B implementation that was investigated in this thesis, the jobs are not simply put next to each other from the beginning to the end of the subproblem. When scheduling the next job, this version of the B&B algorithm first evaluates whether it is better to put the job at the beginning or at the end of the subproblem that will be sent to the bounding operator. For example, if the first job chosen is 2 and is placed at the beginning, and the second job chosen is 3 and placed at the end, the resulting subproblem will be **2***14***3** where the bolded jobs are scheduled and the jobs in italics are not yet scheduled. The branching operator accomplishes this by computing the bound of each possible job that has to be scheduled, first at the beginning, then at the end. For example, if job 2 is already scheduled at the beginning, then the following subproblems will be sent to the bounding operator:

- **21***34*, **23***14*, and **24***13* as the first set of subproblems.
- **2***34***1**, **2***14***3**, and **2***13***4** as the second set of subproblems.

The branching operator then computes the sum of the bounds for the subproblem where the jobs were placed at the beginning, and do the same for the subproblem where the jobs were put at the end. The highest sum determines which one will be chosen. In this

example, if the sum of the second set of subproblems is higher, then the first set is discarded completely. After that, the elimination operator removes the subproblems whose bound is higher than the best known solution. The remaining subproblems are then inserted in the pool of subproblems.

Elimination operator:

The conventional elimination or pruning operator takes a set of subproblems and removes all the subproblems whose bound is greater than the best known solution. The time complexity of a deletion in LL is $O(1)$. Each element of LL will be considered for deletion and there are at most N elements in the list, N being the size of the problem. This means that the elimination operator will perform at most N deletions. Therefore, the time complexity of the elimination operator is $O(N)$.

Bounding operator:

As indicated by its name, the Branch-and-Bound algorithm requires two main operations: branching and bounding. The bounding operator is a procedure that computes upper and lower bounds for a given subproblem. This operator is the core of the B&B algorithm since the quality of the bounding largely determines the resolution time. Besides, the bounding operator often consumes the biggest part of this resolution time [Chakroun 2013c].

3.3 IVM-based serial B&B

The use of LL to store a pool of subproblems described in Subsection 3.2.2 has a number of disadvantages. The insertion of new subproblems in LL can be very costly, since it implies finding the right position in LL for each subproblem, which involves the comparison of each subproblem to the ones already present in LL until its place is found. Considering that LL can grow and get very large, the branching operator can take a significant amount of time due to the insertions in LL. Another problem to consider is the memory usage of LL, which can grow rapidly. The use of a Depth First Search approach to explore the tree can make this problem less significant than in a Breadth First Search, but LL can still grow rapidly for problems of a significant size.

It is therefore necessary to create a new data structure which exhibits a better behavior in the management of the subproblems of a permutation problem. The IVM (Integer Vector Matrix) data structure allows to store and manage subproblems in a more efficient way than LL. It uses a constant amount of memory which makes its behavior much more predictable than LL. Its branching operator is also less costly than the one described in

Subsection 3.2.2. It is however less generic than the LL data structure when used in the B&B algorithm. It can only be used for permutation problems and only allows for a Depth First Search exploration of the tree of subproblems.

3.3.1 IVM data structure

The IVM version of the Branch-and-Bound algorithm (IVM-B&B) is shown in Algorithm 9. Figure3.3 shows the representation of the state of the pool of Figure 3.1 using an integer, a square matrix of integers and a position-vector instead of LL used in Section 3.2. The size of these square matrix and vector is equal to the size of the problem. In this new version, the values of the position-vector belong to factorial number system, and this vector behaves like a counter in the factorial system. In this example, their size is equal to four. Each cell of the matrix represents a subproblem from the B&B pool. In other words, a single integer represents a subproblem rather than a permutation of integers. In this matrix, a cell with a row number strictly greater than its column number is never used (upper triangular matrix). In each cell, only the new not yet scheduled jobs are represented. For example, the three subproblems obtained after the decomposition of the second cell of the first row are written in the second row. In this second row, only job 1 is written in the first cell, job 3 in the second cell, and job 4 in the third cell.

For each row, the associated position vector index always points to the cell which is currently explored. In other words, the index points to the last decomposed subproblem of its row. In Figure 3.3, the first and second indexes point to the second cell, while the third and fourth indexes point to the first cell of their rows. The last index always points to the first cell. Jobs already scheduled are deduced from previous rows using the position-vector. And of course, non-scheduled jobs are those which do not belong to the list of scheduled jobs. For example, when the algorithm points to the first cell of the last row, the scheduled job in this cell is 4. Before this job, the indexes of the position-vector indicate that the jobs scheduled are 1 in the third row, 3 in the second row, and 2 in the first row. All jobs are scheduled. Therefore, this cell of the matrix encodes the subproblem *2314/* which is a solution.

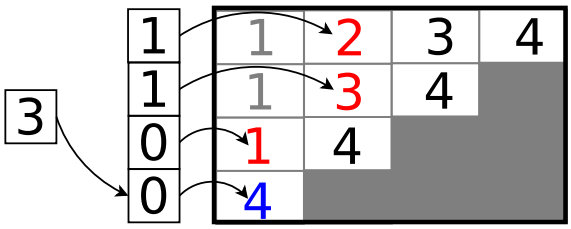


Figure 3.3: IVM representation of a pool of subproblems.

Algorithm 9 IVM-based B&B algorithm.

```
1:  procedure IVM-B&B-MAIN
2:      while (true) do
3:          if (row-end) then
4:              if (first-row) then
5:                  exit-program
6:              else
7:                  cell-upward
8:              end if
9:          else
10:             if (cell-eliminate) then
11:                 cell-rightward
12:             else
13:                 next-row-process
14:                 cell-downward
15:             end if
16:         end if
17:     end while
18: end procedure
19: procedure NEXT-ROW-PROCESS
20:     cell-branch
21:     for all cell of the next row do
22:         cell-selection
23:         cell-bound
24:     end for
25: end procedure
```

The IVM data structure within the context of the tree:

In order to clarify the relationship between the IVM data structure and the tree that is explored, Figure 3.4 shows the whole tree and the parts that are stored in the IVM data structure. It shows the same solution *2314/* as Figure 3.3 within the context of the entire tree, the root of this tree is the starting problem */1234*.

In this tree, the job numbers are shown in black inside the nodes, the blue number below and to the left of each node is the index of that job with respect to its parent node. The red numbers are the jobs which have been scheduled, those jobs are the ones whose indices correspond to the *1100* position vector of Figure 3.3. As can be seen, the numbers in the position vector represent the index of the job that is currently scheduled. In this example, the position vector *1100* points to the jobs *2314*.

Each red box shown in the tree corresponds to one line in the matrix of the IVM data structure. This shows that at any given moment, the IVM data structure can store only one complete branch of the whole tree. However by modifying the values of the position vector, it is possible to store any of the possible branches, therefore it is possible to explore the tree in its entirety using an IVM data structure.

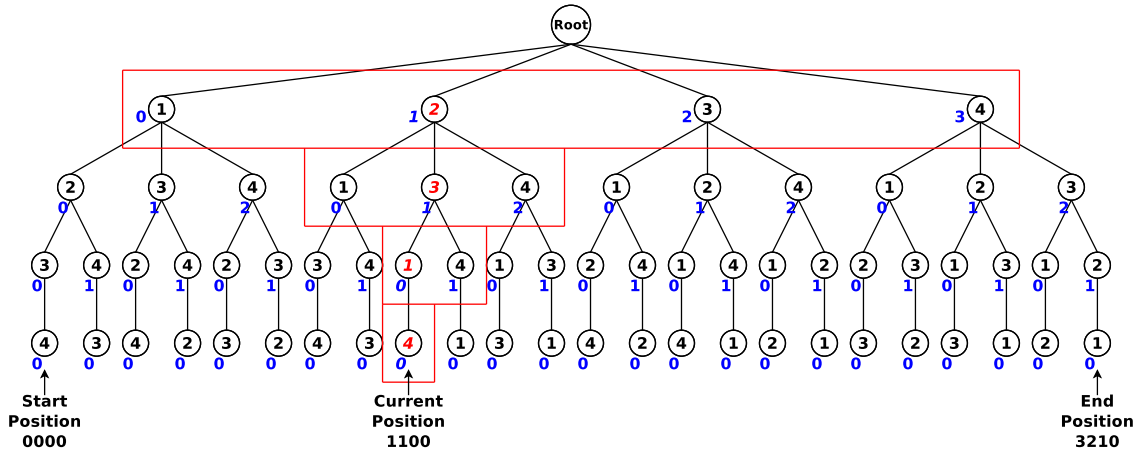


Figure 3.4: Representation of the IVM data structure within the context of the whole tree.

3.3.2 IVM advanced techniques

Scheduling a job at the beginning or at the end with IVM:

As explained in Section 3.2.3, this particular implementation of the B&B algorithm chooses whether it will put a newly scheduled job at the beginning or at the end of the permutation. The choice is made in the same way as the Linked LL version, however IVM-B&B stores these choices in a different manner. IVM-B&B uses a direction vector which contains either "BEGIN" or "END" for each row of the matrix. This way the selection operator knows whether a job should be scheduled at the beginning or at the end.

In the example shown in Figure 3.5, job 2 is scheduled at the beginning and job 3 is scheduled at the end, job 1 and job 4 are non-scheduled. Therefore, the subproblem represented by this instance of IVM is **2***14***3**. Since the Integer is pointing at the second row of the matrix, only the first two rows have a BEGIN or END value, the other two rows do not contain scheduled jobs yet, so a BEGIN or END value would have no meaning.

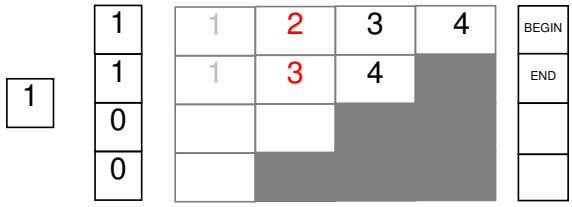
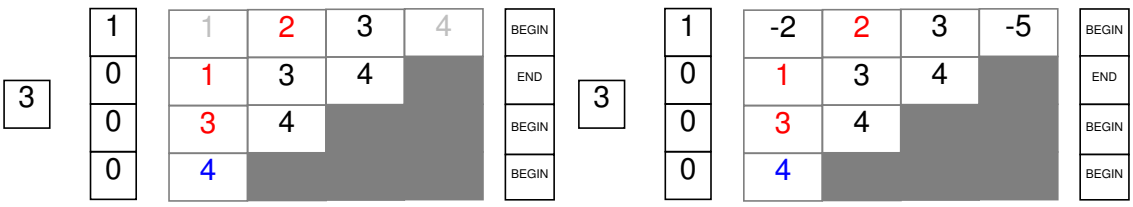


Figure 3.5: IVM representation of a pool of subproblems with a direction vector

Keeping track of which subproblems have been eliminated:

The B&B algorithm eliminates the subproblems whose cost is superior to the best known solution. However, keeping track of the cost of each subproblem would require a second matrix containing these costs. In order to avoid that, we need a way to mark subproblems as eliminated that does not use more memory. The subproblems are all stored as integer greater than or equal to zero, which leaves the negative integers unused. By using the negative integers to represent the eliminated subproblems, we can keep track of the eliminated subproblems without having to store their cost, which means we do not use more memory for a second matrix. If a subproblem is stored as the integer N in the matrix, then it will be stored as $-N-1$ if it is eliminated (Or as $-N$ in the case where job 0 is assumed does not exist). For example, subproblem 1 will be stored as -2 if it is eliminated, then by doing the same operation, we can find out what the original subproblem was.

In the example shown in Figure 3.6a, let's suppose that on the first row, the bound computed for subproblems 1 and 4 is superior to the best known solution. This means that subproblems 1 and 4 have been removed by the elimination operator. The actual contents of the Matrix are shown in Figure 3.6b, subproblems 1 and 4 are stored as the negative integers -2 and -5. This allows IVM-B&B to know that those two subproblems should be skipped when exploring the tree of subproblems.



(a) Abstract representation.

(b) In-memory representation.

Figure 3.6: IVM representation of eliminated subproblems.

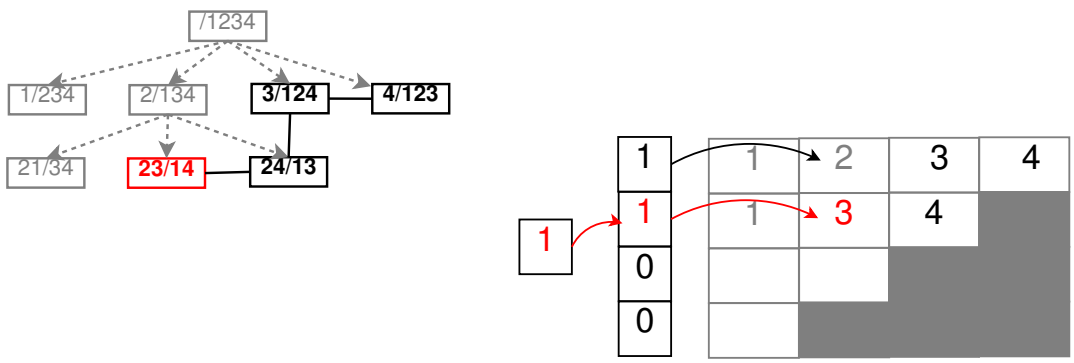
3.3.3 Revisited B&B operators

The B&B algorithm consists of four operators: selection, branching, elimination and bounding. The IVM data structure does not affect the bounding operator. However the selection, branching and elimination operators have to be revisited to work with this new data structure.

Selection Operator:

As shown in Figure 3.7b, the integer always points to one component of the vector, and this component points to a cell of the matrix. The selection operator simply has to decode the subproblem associated to this cell. The list of scheduled jobs are those that are pointed by the vector: *2 3*, and the list of non-scheduled jobs are those of the last row: *1 4*, job 3 is excluded because it is pointed by the vector and thus is one of the scheduled jobs. Therefore, the selected subproblem is *23/14*.

In order to generate the permutation, the vector must be read up to the index stored in the Integer, which means at most N reads, N being the number of jobs. In this example, 2 and 3 are read, not 1 and 4 because the integer points to 1. The matrix must also be read, however only one element on each row is read, except for the last row pointed by the integer, where the other elements contain the non-scheduled jobs. This means that N elements are read from the Matrix. In this example, only 2 is read in the first row, then 3 in the second row, the remaining jobs of the second row 1 and 4 are the unscheduled jobs. So, at most $2N$ elements are read for the selection operator and N elements are written to the permutation, which means that its time complexity is $O(N)$.



(a) LL representation. (b) IVM representation.

Figure 3.7: Selection Operator.

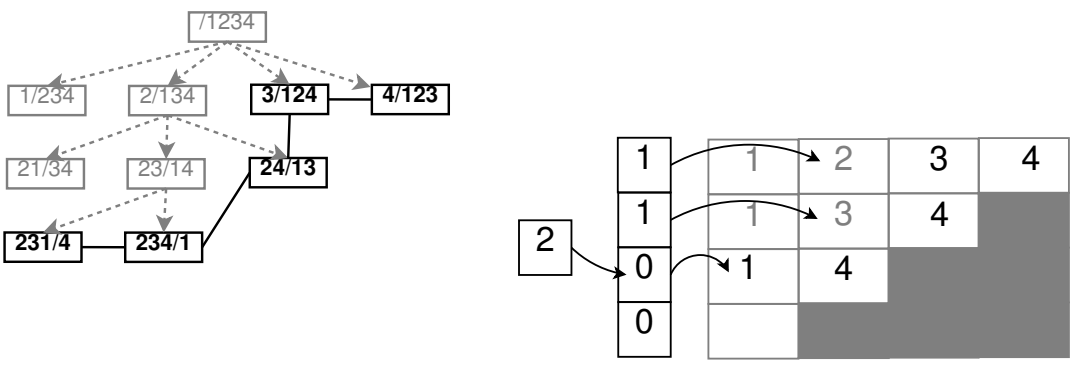
Branching Operator:

To divide the subproblem *23/14* seen in Figure 3.7b, the branching operator only has to copy the row pointed by the Integer to the next row excluding the job that is pointed by the position Vector. The Integer is then incremented to point to the new row.

As shown in Figure 3.8b, two new subproblems have been created, *231/4* and *234/1*. To be able to fill the new row, all the elements of the previous row must be read. This means that there will be at most N reads and $N - 1$ writes, N being the number of jobs. In this example, jobs 1 and 4 are read then written to the next line, while job 3 is ignored because it is pointed to by the Vector.

After the new row is created, it needs to be sorted according to the lower bound of each subproblem. When multiple subproblems have the same lower bound, they are sorted according to the number of the subproblem. The better behaved sorting algorithms generally have a time complexity of $O(N\ log(N))$ and the number of elements to sort is at most N, therefore the time complexity of the branching operator is $O(N\ log(N))$, where N is the size of the problem.

The reason for sorting the subproblems is to make sure that the subproblems are explored in the same order for both LL-B&B and IVM-B&B. While this does not change the number of explored subproblems when the best known solution is initialized with the optimal solution, it makes a difference when it is initialized with infinity. In general use, the sorting of subproblems is not required, which reduces the time complexity of the branching operator to $O(N)$.



(a) LL representation. (b) IVM representation.

Figure 3.8: Branching Operator.

Elimination Operator:

As shown in Figure 3.9b, to discard the subproblem *231/4*, the elimination operator only has to increment the component of the Vector which is pointed by the Integer. The selected subproblem is now *234/1*.

If the Vector component points to the end of the row, the Integer is decremented and this component of the Vector is incremented. Figure 3.10b shows that the selected subproblem is now *24/13*.

The incrementation of an element in an array has an $O(1)$ complexity. When the subproblem is the last of its row, the elimination must decrement the Integer and increment the pointed element of the Vector. However, it is possible that the subproblem pointed to on the previous row is also at the end of the row, leading to another decrementation of the Integer. In the worst case, the Integer can go from the last row all the way up to first row, which requires $N - 1$ decrementations. Therefore, the time complexity of the elimination operator is $O(N)$.

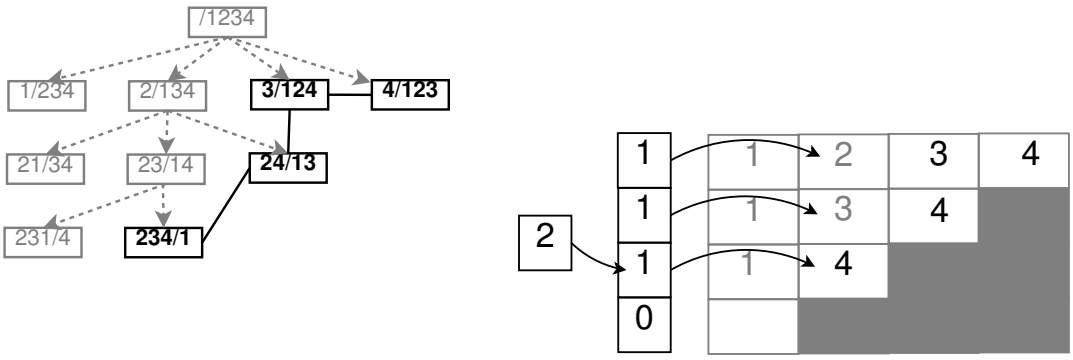
3.4 Experimentation

This section compares the LL-based B&B and the IVM-based B&B in terms of memory and CPU time usage.

3.4.1 Experimental settings

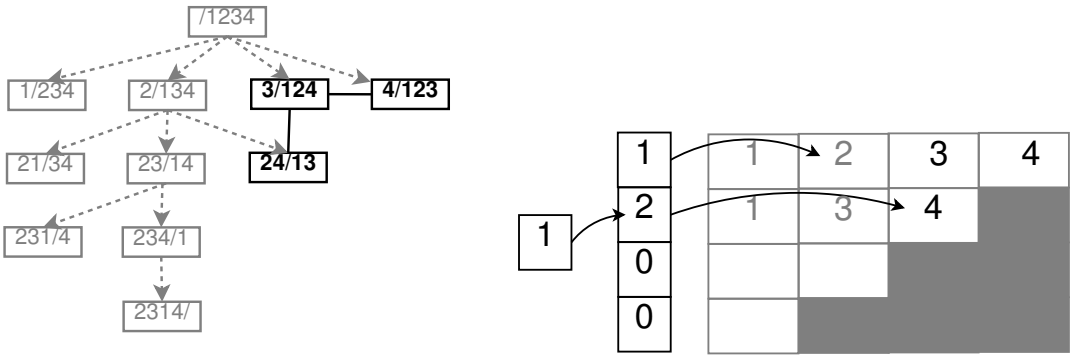
The performance of a B&B algorithm depends mainly on the efficiency of the used bounding operator. The lower bound proposed by Lageweg *et al.* [Lenstra 1978] is used in our bounding operator. This bound is known for its good results and has a complexity of $O(M^2 N log(N))$, where N is the number of jobs and M the number of machines. This lower bound is mainly based on Johnson's theorem [Johnson 1954] which provides a procedure for finding an optimal solution for Flow-Shop scheduling problem with 2 machines.

In our experiments, we used the Flow-Shop instances defined by Taillard [Taillard 1993]. 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. For example, the resolution of the instance Ta056 (The sixth instance with 50



(a) LL representation. (b) IVM representation.

Figure 3.9: Elimination Operator when the subproblem is not the last of its row.



(a) LL representation. (b) IVM representation.

Figure 3.10: Elimination Operator when the subproblem is the last of its row.

jobs and 20 machines), performed in [Mezmaz 2007b], lasted 25 days with an average of 328 processors and a cumulative computation time of about 22 years.

The serial LL-B&B and IVM-B&B have been implemented in C++ and compiled using GCC 4.6 with $-O3$ option. All the experiments were run on the computer Poincaré which belongs to Maison de la Simulation. 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. For each Flow-Shop instance the computational time spent in managing the pool of subproblems is measured using the *clock_gettime* function with a nanosecond precision.

3.4.2 Memory evaluation

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 data structure is always constant. In bytes, this size can be calculated using Equation (3.1).

$$\begin{aligned} Maximum - size(IVM) &= [\frac{N\ (N+1)}{2} + 3\ N + 1] \text{ bytes} \\ &= [\frac{1}{2}N^2 + \frac{7}{2}N + 1] \text{ bytes} \end{aligned} \tag{3.1}$$

Unlike IVM, the advantage of LL is to not require additional CPU calculation time for generating a subproblem. In the LL approach, a conventional coding of a subproblem 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 subproblem of an instance defined by N jobs is always equal to N bytes, and therefore, the size of a pool LL, which contains X subproblems, is equal to $N \times X$ bytes. In LL, a pool reaches its maximum size when visiting the first solution. At this moment, the pool contains $N-1$ subproblems with 1 job scheduled, $N-2$ subproblems with 2 jobs scheduled, ..., until $N-(N-1)$ (i.e. 1) subproblem with $N-1$ jobs scheduled. The maximum size of a pool LL can be calculated using Equation (3.2).

$$\begin{aligned} Maximum - size(LL) &= [\sum_{i=1}^{i=N-1} (N-i)] \text{ subproblems} \\ &= [\sum_{k=1}^{k=N-1} k] \text{ subproblems} \\ &= [\frac{N\ (N-1)}{2}] \text{ subproblems} \\ &= [\frac{N^2\ (N-1)}{2}] \text{ bytes} \\ &= [\frac{1}{2}N^3 - \frac{1}{2}N^2] \text{ bytes} \end{aligned} \tag{3.2}$$

In terms of space, this means that the IVM data 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 3.1 concretely shows the ratio between the two approaches. These ratios are computed using Equation (3.1) and Equation (3.2). 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 3.1: Comparison of serial IVM and LL B&B algorithms in terms of maximum memory.

However, the comparison of Table 3.1, 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 (3.2), 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.

Tables 3.2 and 3.3 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 subproblems 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.

Instance	LL average size (Subproblems)	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 3.2: 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 (Subproblems)	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 3.3: Comparison of the size of IVM and the average size of LL when solving the ten instances defined with 50 jobs.

According to Table 3.1, 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.

More results are available in the appendix in Tables A.1 and A.2.

3.4.3 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 subproblems 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. So the objective of this subsection is to compare the LL and IVM approaches in terms of pool management CPU time.

In our comparative evaluation, the pool management time does not only include the CPU time spent by reading and writing operations in the LL and IVM data 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 B&B algorithm except the bounding operation. As indicated in this chapter, the selection, pruning and branching operators are adapted to the IVM data structure. Therefore, these three operators are not implemented in the same way for the LL and IVM approaches. So, it is fair to include their CPU time cost in the pool management time.

Table 3.4 gives the average time obtained for the instances defined with 20 jobs, and Table 3.5 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.

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	$\sim$ **2.75**

Table 3.4: Comparison of serial IVM and LL-based B&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	$\sim$ **3.00**

Table 3.5: Comparison of serial IVM and LL-based B&B algorithms in terms of the CPU time used for the management of the pool when solving the instances defined with 50 jobs.

The previous subsection shows 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 3.4 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 3.5 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 three operators. The selection, branching and pruning operators are more optimized in the IVM approach compared to the LL approach. For example, the LL branching operator involves creating a certain number of subproblems, and each subproblem must contain a permutation almost similar to the subproblem which is branched. Unlike this LL operator, the IVM branching operator merely copies the content of a matrix row to another row.

More results are available in the appendix in Tables A.3, A.4, A.5 and A.6.

3.5 Conclusion

In this chapter, we proposed a new data structure, called IVM (Integer Vector Matrix), to implement the pool of subproblems generated by a B&B algorithm for solving permutation optimization problems. The position-vector of IVM behaves like a counter in the factorial number system. In this new approach, it is necessary to transform the value of this position-vector from a factorial value to a permutation. The IVM data structure does not affect the bounding operator since this operator does not work directly on the pool. However, the selection, branching and elimination operators have been revisited to operate on this new data structure.

This new approach is validated using standard instances of the Flow-Shop 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 significantly 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.

CHAPTER 4

Multi-core IVM-based B&B

Contents

4.1 Introduction **55**

4.2 Parallel models for B&B algorithms **56**

4.2.1 Multi-parametric parallel model 56

4.2.2 Parallel evaluation of bounds model 57

4.2.3 Parallel evaluation of a bound model 58

4.2.4 Parallel tree exploration model 58

4.3 Work stealing strategies for multi-core IVM-based B&B **59**

4.3.1 WS-based B&B implementation 59

4.3.2 Coalesced work units 61

4.3.3 Dividing one factoradic interval into two intervals 63

4.3.4 Victim selection and granularity policies 67

4.4 Experimentation **72**

4.4.1 Experimental settings 72

4.4.2 Strategy and granularity policies evaluation 75

4.4.3 Memory evaluation 77

4.4.4 CPU time evaluation 78

4.5 Conclusion **80**

4.1 Introduction

Thanks to their bounding operator B&B algorithms can significantly reduce the computing power needed to explore the whole solution space. However, such power may still be huge, especially when solving large instances. That is why the use of increasingly powerful hardware is also necessary to solve these larger instances. For years CPUs have been made faster by increasing the clock frequency. However that method has reached a physical barrier due to energy consumption and heat dissipation. Hardware manufacturers such

as Intel or AMD have worked around these problems by building multi-core CPUs which provide more computing power while not increasing the clock frequency.

Work stealing has been proven to be an effective method for scheduling irregular parallel programs such as Branch-and-Bound (B&B) on multi-core processors [Shavit 2011, Acar 2013]. In this chapter, the focus is put on multi-core B&B algorithms for solving large scale permutation-based optimization problems. Five work stealing (WS) strategies are investigated using the IVM data structure presented in the previous chapter. In these strategies, each thread has a private IVM allowing the local management of a set of subproblems enumerated using a factorial system presented in the first chapter. The WS strategies differ in the way the victim thread is selected and the granularity of stolen work units (intervals of factoradics).

The chapter is divided into three sections. Section 4.2 describes the different approaches proposed in the literature to parallelize B&B algorithms. Section 4.3 presents the five WS strategies investigated in our work. Finally, Section 4.4 compares this five IVM WS strategies to their conventional linked-list-based counterparts.

4.2 Parallel models for B&B algorithms

Many approaches to parallelize B&B algorithms are proposed in the literature. A taxonomy of these models is presented in [Melab 2005]. This taxonomy is based on the classifications proposed in [Cung 1994] and [Gendron 1994]. Four models are identified: the multi-parametric parallel model, the parallel evaluation of bounds model, the parallel evaluation of a bound model, and the parallel tree exploration model. This later, illustrated in Figure 4.3, is the most frequently used in the literature and it is also the focus of this thesis.

4.2.1 Multi-parametric parallel model

The multi-parametric parallel model (Figure 4.1), relatively less studied in the literature, is based on the use of several B&B algorithms run in parallel. This is a coarse-grained model. Several variants of this model may be considered according to the choice of one or more parameter(s) of the B&B algorithm. The parallel B&B algorithms differ only by the branching operator in [Miller 1993]. These parallel algorithms are different only by the selection operator in [Janakiram 1988] where a variant of the depth-first exploration strategy is used. Each algorithm randomly selects the next subproblem to be addressed among the last generated subproblems. In [Kumar 1984], each algorithm uses a different upper bound in their tests. The idea is that one algorithm uses the best upper bound found

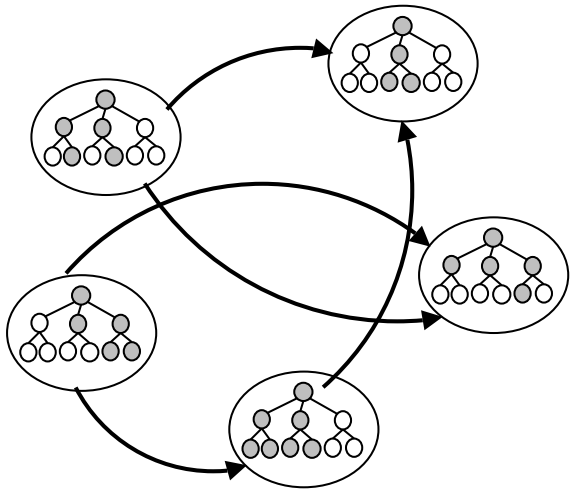


Figure 4.1: Illustration of the multi-parametric parallel model.

while the others use this bound reduced by an ϵ value(ϵ-optimal, where $\epsilon > 0$). Another variant of this parallel model consists of decomposing the interval defined by the lower and upper bounds of the subproblem to be solved into subintervals. Each subinterval is assigned to one of these algorithms.

The main advantage of the multi-parametric parallel model is its genericity allowing its use in a transparent manner to the end-user. Its disadvantage is the overhead in the computation it generates since some subproblems in the tree are explored in a redundant manner. However, this extra computing cost has less consequence when the model is deployed on a high performance computing system since it has many computing resources.

4.2.2 Parallel evaluation of bounds model

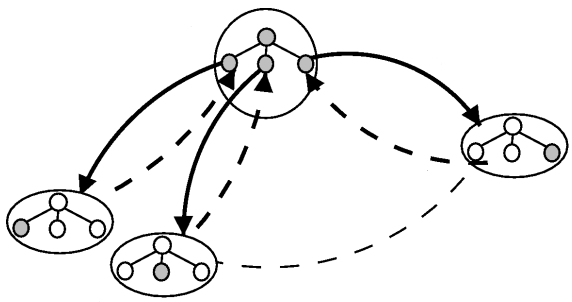


Figure 4.2: Illustration of the parallel evaluation of bounds model.

The parallel evaluation of bounds model (Figure 4.2) allows the parallelization of the

bounding of subproblems generated by the branching operator. This model is used in the case where the bounding operator is performed many times after the branching operator. The model does not change the order nor the number of explored subproblems in the parallel B&B algorithm compared to the serial B&B. Besides the fact that the bounding phase is faster in the parallel evaluation of bounds B&B than the serial B&B, the main advantage of this model is its genericity. However, this model can be inefficient in some parallel computing environments for the following reasons:

- The model is synchronous and therefore unsuitable for heterogeneous and volatile contexts.
- Its granularity (the cost of the bounding operator) can be fine and therefore unsuitable for high performance computing systems.
- The degree of parallelism of this model depends on the addressed problem. For the Flow-Shop problem, the more a subproblem is deep in the tree, the more the number of its subproblems decreases, so the less this model is suitable.

The combination of this model with the parallel tree exploration model can generate a higher degree of parallelism than using this model alone.

4.2.3 Parallel evaluation of a bound model

This model does not change the semantics of the algorithm because it is similar to the serial version except that the bounding operator is faster. The efficiency of this centralized and synchronous model depends on the addressed problem. Because of its scalability, the efficiency of this model depends on its combination with another model.

4.2.4 Parallel tree exploration model

The parallel tree exploration model consists of simultaneously exploring several subproblems that define different research subspaces of the initial problem (Figure 4.3). This means that selection, branching, bounding and pruning operators are executed in parallel synchronously or asynchronously by different processes exploring these subspaces. In synchronous mode, a B&B algorithm has different phases. During each phase, the B&B processes of the algorithm do their exploration independently. Between the phases, the B&B processes are synchronized to exchange information, such as the best solution found so far. In asynchronous mode, the B&B processes communicate in an unpredictable manner.

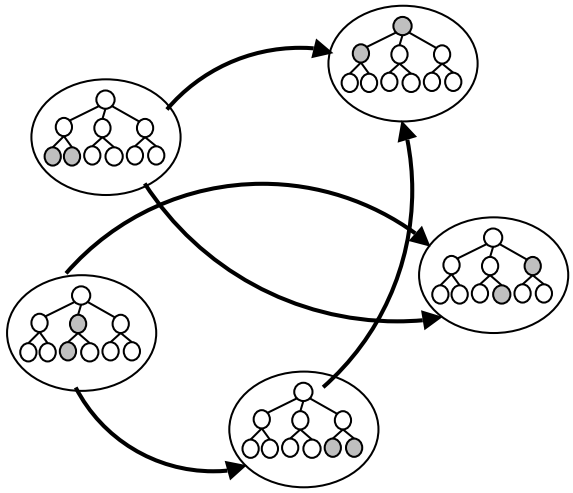


Figure 4.3: Illustration of the parallel tree exploration model.

Compared to other models, the parallel tree exploration model is more frequently used and is the subject of much research for two main reasons. On the one hand, the degree of parallelism of this model may be important when solving large problem instances justifying the use of multi-core computing or a high performance computing system. On the other hand, the implementation of the model raises several issues that constitute interesting research challenges in parallel computing. Among these issues, we can include the placement and management of the set of subproblems to be solved, the distribution and sharing of the load (generated subproblems), the communication of the best solution found so far, detecting the termination of the algorithm, and fault tolerance.

4.3 Work stealing strategies for multi-core IVM-based B&B

4.3.1 WS-based B&B implementation

Work stealing has proven to be an effective method for scheduling irregular parallel programs such as Branch-and-Bound on shared memory computers. In addition, the efficiency of parallel B&B depends strongly on the implementation of the work stealing paradigm and the data structures used to store the generated subproblems at runtime. The design of concurrent data structures on multi-core computers is becoming increasingly challenging with the advent of multi-core processors as the standard computing platform [Shavit 2011]. For B&B algorithms, the choice of the data structure depends on the exploration strategy. For instance, in [Chakroun 2013a] the authors propose a parallel depth-first B&B for multi-core processors combined with GPUs. The multi-core part is implemented using Pthreads library. The work stealing mechanism is based on

a concurrent stack which is made non-blocking by using the *try* primitive instead of *lock*. For the best-first B&B, the priority queue data structure is often used. For instance, in [Le Cun 1995] this data structure is used for parallel tree search algorithms (B&B and A*) on shared memory computers. Partial locking is used to allow non-blocking concurrent accesses and speed up the exploration process.

Unfortunately, concurrent data structures suffer from a memory issue. For instance, in [Acar 2013], it is reported that the deque (doubly-ended queue) operations require expensive memory fences in modern weak-memory architectures. Therefore, there has been a lot recent interest in implementations of work stealing with non-concurrent (private) data structures such as deques [Acar 2013, Shavit 2011]. In this thesis, we investigate and compare two different implementations of a private data structure conceptually similar to deque. The first one is implemented as an IVM data structure. The other one is based on the implementation of deque provided in C++ (referred to as LL in this document) and used for the B&B.

Algorithm 10 describes a thread of our multi-threaded IVM-based B&B. Each thread explores its interval using its own integer, position-vector and matrix. In this algorithm, each one of the T threads of the algorithm runs *IVM-B&B-thread* procedure.

At the beginning, each thread initializes its position-vector and matrix. Threads are numbered from 0 to $T-1$. Each thread R is initialized with the interval $[R*N!/T,(R+1)*N!/T[$. In other words, the vector-position is set to $R*N!/T$. Its matrix is initialized by calling *factoradic-set* described in Subsection 4.3.2. While there is at least one non-empty interval, *IVM-B&B-thread* applies a new iteration. In this iteration, *IVM-B&B-thread* checks the status of its interval. Three scenarios can occur:

- ***interval-empty***: An interval is said empty when its beginning is greater than or equal to its end. In this case, the thread R sends a work request to the thread R' chosen by the *choose-thread* function. Once an interval $[B',E'[$ is received, the role of *factoradic-set* is to initialize the matrix according to the value of B'.

- **Not *interval-empty* and *interval-request***: The interval is not empty and a work request is received. In this case, the thread R divides the interval using *interval-steal* and sends the resulting interval to the thread that made the request.

- **Not *interval-empty* and not *interval-request***: The interval is not empty and no work request is received. In this case, the thread processes the current cell of the

matrix by calling *cell-process*.

In our approach, the B&B algorithm points always to the cell $[i, P_i]$ of the matrix such that *i* is the row which is currently being processed. As indicated in the code of *cell-process*, three scenarios may occur when processing a cell. These scenarios are described in the following:

- ***row-end***: The algorithm reaches the end of the current row. This means that the indexes of the current cell $[i, P_i]$ satisfy the condition $P_i > i$. In this case, the next cell to process is located in the previous row and this cell is $[(i-1), P_{i-1}+1]$ (i.e. *cell-upward*).

- ***cell-eliminate***: The bound of the node associated to the current cell indicates that this node can be eliminated. In this case, $[i, P_i+1]$ (i.e. *cell-rightward*) is the next cell of the matrix to process.

- ***cell-promising***: The bound of the node associated to the current cell indicates that this node cannot be eliminated. In this case, $[i+1, 0]$ (i.e. *cell-downward*) is the next cell of the matrix to process. Before processing this cell, the thread runs *next-row-process* to initialize the row *i+1*.

The procedure *next-row-process* decomposes the node associated to $[i, P_i]$. This decomposition is done by copying all the jobs of the row i to the row $i+1$ except the job of $[i, P_i]$. Then for each cell of the row $i+1$, the thread reads and decodes the node associated to this cell, and computes its bound.

4.3.2 Coalesced work units

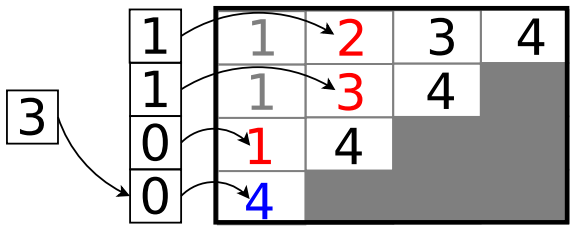


Figure 4.4: IVM representation of a pool of subproblems.

Dynamic load balancing is strongly required for parallel B&B algorithms [Dowaji 1995, Mezmaz 2007a]. To the best of our knowledge, except in rare works such as [Mezmaz 2007b], work units exchanged between threads (or processes)

Algorithm 10 Multi-core IVM-based B&B algorithm.

```
1:  T: Number of threads
2:  D ∈ {2, T}: Divisor of an interval
3:  procedure IVM-B&B-THREAD(R)
4:      [B, E[ ← [R * N!/T, (R + 1) * N!/T[
5:      factoradic-set([B, E[)
6:      while (exist-no-empty-intervals) do
7:          if (interval-empty) then
8:              R' ← choose-thread(R)
9:              [B', E'[ ← interval-receive(R')
10:             factoradic-set([B', E'[)
11:         else
12:             if (interval-request) then
13:                 [B', E'[ ← interval-steal([B, E[)
14:                 interval-send([B', E'[, requester)
15:             else
16:                 cell-process
17:             end if
18:         end if
19:     end while
20: end procedure
21: function INTERVAL-STEAL([B, E[)
22:     [B', E'[ ← [(B + E)/D, E[
23:     [B, E[ ← [B, (B + E)/D[
24:     return [B', E'[
25: end function
26: procedure CELL-PROCESS
27:     if (row-end) then
28:         cell-upward
29:     else
30:         if (cell-eliminate) then
31:             cell-rightward
32:         else
33:             next-row-process
34:             cell-downward
35:         end if
36:     end if
37: end procedure
38: procedure NEXT-ROW-PROCESS
39:     cell-branch
40:     for all cell of the next row do
41:         cell-selection
42:         cell-bound
43:     end for
44: end procedure
```

are often sets of nodes. In our approach the tree nodes (partial or full permutations) are numbered according to the factorial system number presented in Chapter 2. Therefore, it is possible to define an interval of node numbers as the work unit. In the example of the serial B&B of Figure 4.4, the interval explored by the algorithm is $[0000, 3210[$. It is therefore possible to have two threads *T1* and *T2* such as *T1* explores $[0000, X[$ and *T2* explores $[X, 3210[$. 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 all $[0000, 3210[$.

An original load balancing strategy was presented in [Mezmaz 2007b] where the work unit is an interval. Compared to [Mezmaz 2007b], the new strategy presented in this thesis brings three new contributions. The first contribution is that the B&B is based on a matrix of integers instead of a linked list of permutations of integers. The second is that the intervals are expressed with factoradic numbers instead of decimal numbers. And the third is that it is not necessary to use the *fold* and *unfold* operators defined in [Mezmaz 2007b] to transform an interval into a linked list of nodes and *vice versa*.

To implement this new strategy, it is therefore necessary to allow a thread to explore any interval $[A, B[$. A B&B thread must be able to initialize its position-vector to A, to begin the exploration of its interval by incrementing its position-vector value, and to stop when the value of the position-vector is equal to B. To initialize its position-vector to A, a thread must also initialize its associated matrix to the right values. The role of *factoradic-set* in Algorithm 10 is to initialize this matrix. Let's assume that the value of the position-vector is $P_1P_2P_3, ..., P_n$. The initialization of the matrix is done by starting with the first row, followed by the second row, ..., and stops at the nth row. To fill the first row, all jobs are written in this row from the smallest to the highest number. To fill the second row, all jobs of the first row are copied to the second row except the job of position P_1. To fill the third row, all jobs of the second row are copied to the third row except the job of position P_2, etc. Therefore, filling the row $i + 1$ is done by copying all jobs of the row i except the job of position P_i and by keeping the same order for jobs.

4.3.3 Dividing one factoradic interval into two intervals

Dividing the interval at an arbitrary point:

The most obvious way to divide a thread's remaining interval is to add the current position to the end position and divide the result by two. Since the position vectors are factoradic numbers, it is possible to apply algebraic operators to them.

Let's suppose that a thread T1 is exploring an interval $[0000, 3210[$. There is a second thread T2 that is started with an empty interval. T1 explores its interval until T2 tries to steal work from T1. By computing the average between its current position and the end of the interval, position 2100 is determined to be the point where the interval will be divided in two. Thread T1 now explores the interval $[0000, 2100[$ while thread T2 explores the interval $[2100, 3210[$.

Figure 4.5 shows what the tree looks like after T2 steals work from T1. In this figure, T1 is represented in green and T2 is represented in blue. The blue arrows show the nodes that are going to be explored by thread T2 at the start of its interval, the green arrows show the nodes that are going to be explored by thread T1 at the end of its interval.

T1 and T2 are going to explore the same nodes alongside the branch 2100 of the tree, which means that for these nodes the bounds are going to be computed twice. This is a waste of computation time and should be avoided in order to achieve the best possible performance, especially as the number of jobs and therefore the cost of the bound increase.

The red boxes in Figure 4.5 correspond to the lines of the matrix in the IVM data structure. As can be seen in this figure, they will be the same for both T1 and T2, however they will not be the same at the same time. When T1 reaches the point where it explores the branch 2100, it is very likely that T2 will have moved on to another branch of the tree. This means that the top line is the only one that can be shared between T1 and T2. In fact, this top line will always be the same for all threads and for any branch of the tree. This is due to the fact that the first line is a decomposition of the root node which is always the same for a given problem. This means that once one thread has computed the first line, it can then be copied to the IVM data structure of all other threads.

Another problem that arises from this is that the total number of explored nodes is wrong because of the nodes on branch 2100 which are counted twice. Since we use that number in the case where the B&B algorithm is initialized with the optimal solution to verify that the exploration did not skip any nodes, this means we can not be sure anymore whether the exploration was correct or not.

Fortunately, it is possible to fix that problem by making sure that T2 does not start counting the nodes until it has left the branch 2100. However, this still leaves us with the problem of the inefficient exploration of the branch where the tree was cut.

It is possible to avoid this problem by choosing the position where the tree will be cut in two in such a way that T1 ends its exploration just before the position where T2 starts its own exploration without any redundant node.

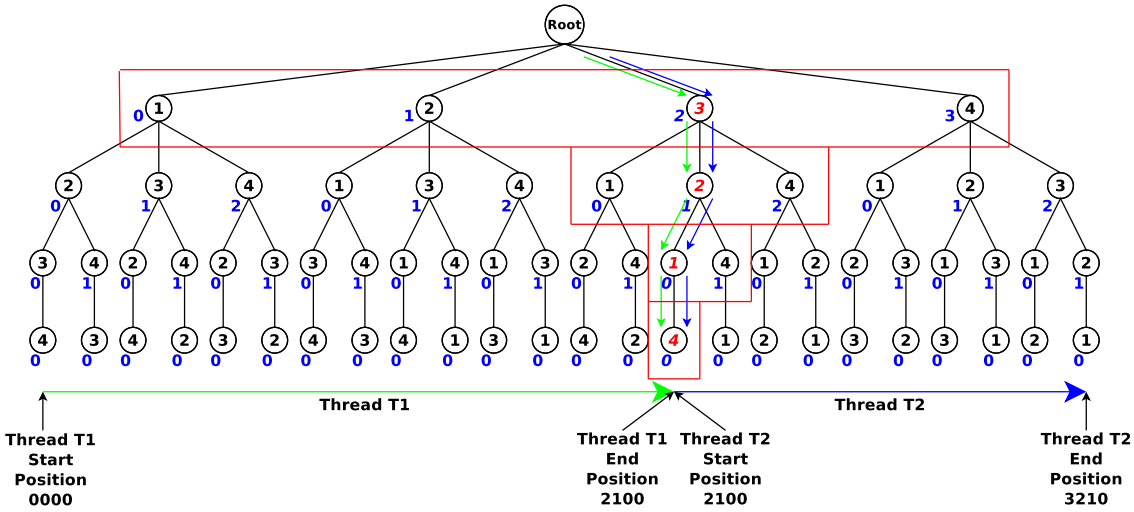


Figure 4.5: Representation of an interval division at an arbitrary point.

Dividing the interval between two subtrees:

Figure 4.6 shows that the ideal position to cut the tree is at the separation between two subtrees. The thread T1 explores the tree until it reaches the branch 1210 as shown by the green arrows, the thread T2 starts its exploration at branch 2000 as shown by the blue arrows and continues until the end of the tree. As can be seen on the figure, there is no longer any redundancy between T1 and T2.

In this example T1 explores the interval $[0000, 1210]$ and T2 explores the interval $[2000, 3210[$. However since in factoradic 1210 plus one is equal to 2000, this means that T1 explores the interval $[0000, 2000[$ and T2 explores the interval $[2000, 3210[$ which covers the entirety of the interval $[0000, 3210[$.

Once again the red boxes correspond to the lines in the matrix of the IVM data structure. As in the previous example, the first line is shared between the two threads, the three other lines however are not since the two threads explore completely separate subtrees. In this case T1 explores subtrees $0XXX$ and $1XXX$ while T2 explores subtrees $2XXX$ and $3XXX$. With subtree $0XXX$ being defined as all the branches from 0000 to

0210.

This only leaves the matter of choosing where to cut the tree to produce an efficient division of the interval. Let's suppose that before T2 stole work from T1, T1 was exploring nodes inside subtree $0XXX$. This means that subtrees $1XXX$, $2XXX$ and $3XXX$ were completely unexplored. When T2 tries to steal work from T1 it sees that three subtrees are left unexplored, three divided by two equals one so T2 leaves one subtree to T1 and takes the rest of the subtrees on the right hand side. T1 is left with subtrees $0XXX$ and $1XXX$ and T2 starts exploring subtrees $2XXX$ and $3XXX$. Now, T1 and T2 need to compute their interval:

- T2 takes T1's end position which is 3210.
- T2 starts its exploration at the first branch of subtree $2XXX$ whose position can easily be computed by taking the position 2 and filling the rest with 0, the result is 2000.
- T1 keeps its starting position.
- T1 ends its exploration at the last branch of subtree $1XXX$ whose position can be computed by taking the position 1 and completing it with a countdown to 0, the result is 1210.

This gives us the two intervals $[0000, 1210]$ for T1 and $[2000, 3210[$ for T2, which as shown earlier, covers the entirety of the interval $[0000, 3210[$.

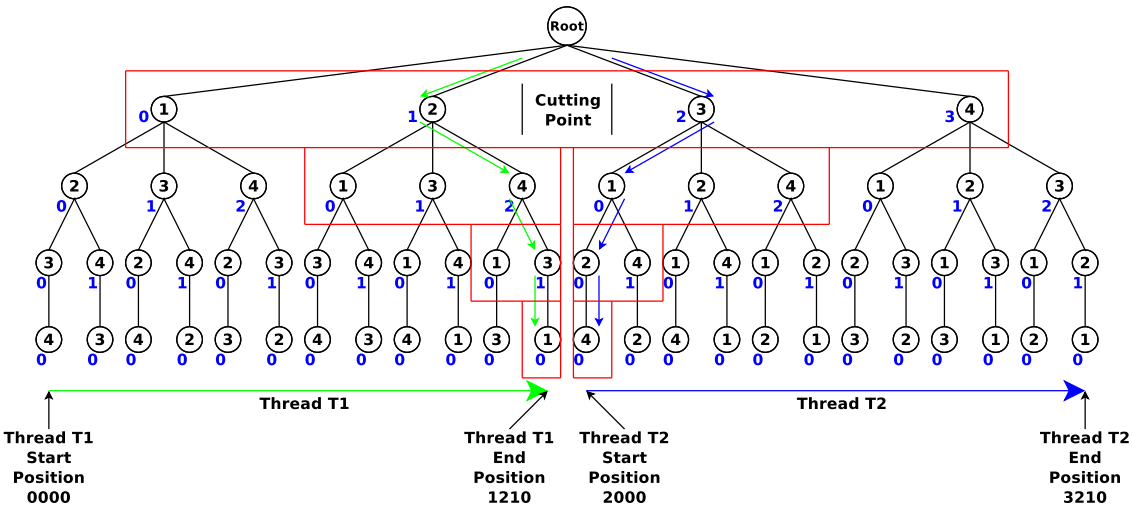


Figure 4.6: Representation of an interval division at a point chosen to avoid redundancy.

Generalization of the division between subtrees:

In Figure 4.7, we show a problem of size 8. A thread T1 is exploring the subtree $7654XXXX$, which means that its starting position is 76540000 and its end position is 76543210, its current position is somewhere in the subtree $76540XXX$.

A thread T2 tries to steal work from T1, it is not possible to do so at the first line because for that line the current and end position are both equal to 7, which means that there is no other subtree for T2 to steal at this depth. The same goes for all the following lines until it reaches a depth where 4 subtrees exist, $76540XXX$, $76541XXX$, $76542XXX$ and $76543XXX$. Subtree $76540XXX$ is being explored by T1, which leaves three subtrees available for a division $76541XXX$, $76542XXX$ and $76543XXX$.

Those three subtrees are not necessarily all candidates for further exploration, it is possible that the branching operator marked one or more of them as having a cost superior to the best known solution. In this example, let's suppose that subtree $76543XXX$ was marked as not requiring anymore exploration, this means that the only two candidates left for T2 to steal are $76541XXX$ and $76542XXX$. Two divided by two equals one, so T2 will leave one subtree and take the rest. In order to not create more than two new intervals, T2 will leave the subtree on the left and take the subtrees on the right. In this case this means that T1 keeps subtrees $76540XXX$ and $76541XXX$, while T2 takes subtrees $76542XXX$ and $76543XXX$. The reason T2 takes subtree $76543XXX$ is to make sure that the two new intervals cover the entirety of the interval that was divided. In this case T1's new interval is $[76540000, 76541210]$ which is equivalent to $[76540000, 76542000[$ and T2's new interval is $[76542000, 76543210[$.

The red boxes in Figure 4.7 correspond to the lines of the matrix of the IVM data structure. In this example, the first five lines are common to T1 and T2, this means that T2 does not have any branching to do and can simply copy the first five lines of T1's matrix into its own. This avoids a lot of unnecessary computation for T2. The lines in the matrix are counted from top to bottom starting from 0, so if T2 finds a position where it can divide the interval at line L then it can simply copy the first $L + 1$ lines of T1's matrix.

4.3.4 Victim selection and granularity policies

Figure 4.8 illustrates LL-B&B and IVM-B&B parallel algorithms using the same work stealing strategy. A strategy can be defined by its victim selection policy and its granularity policy. The victim selection policy indicates the thread victim R' that a thread R can steal,

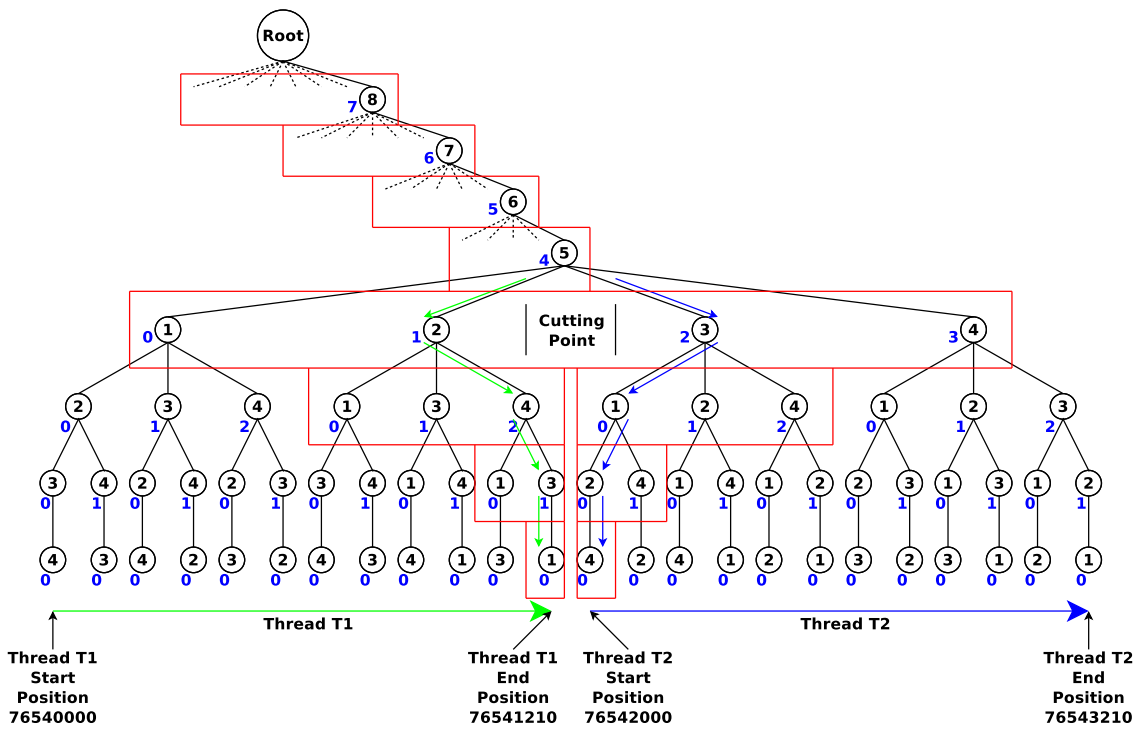


Figure 4.7: Representation of a generalization of an interval division between subtrees.

and the granularity policy determines the amount of work of thread R' stolen by the thread R. Algorithm 11 shows the ways a thread R can choose its thread victim R'. An ideal strategy is the one which (1) chooses the thread victim R' with the largest amount of work, (2) and makes this choice as rapidly as possible. In this thesis, we define four victim selection policies, explained in Subsection 4.3.4, and two granularity policies described in Subsection 4.3.4. Our IVM-B&B parallel algorithm is compared to the LL-B&B parallel algorithm using work stealing strategies defined by these victim selection policies and granularity policies.

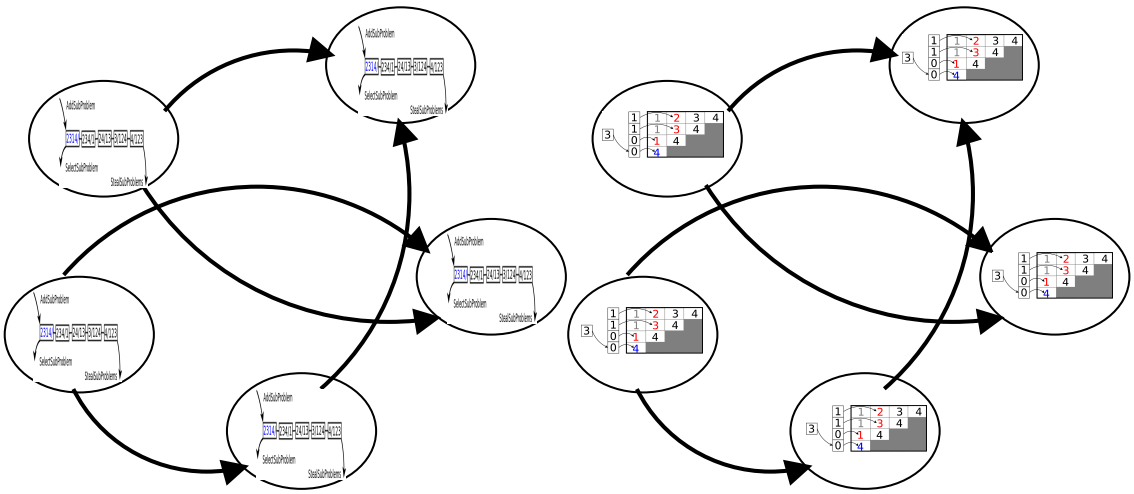
Victim selection policies:

In this section, four victim selection policies are described: Two of them with a small computational complexity, namely the ring and the random policies, and the two others with a greater computational complexity, namely the largest and the honest policies.

- ***Ring victim selection policy***: In this policy, threads are connected to each other with a unidirectional ring. A thread R always steals from 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

Algorithm 11 Pseudocode of the victim selection policies.

```
1: function CHOOSE-THREAD(R,STRATEGY)
2:     switch STRATEGY do
3:         case RING:
4:             return choose-ring(R)
5:         case RANDOM:
6:             return choose-random()
7:         case LARGEST:
8:             return choose-largest(R)
9:         case HONEST:
10:            return choose-honest(R)
11: end function
12: function CHOOSE-RING(R)
13:     if (R=1) then
14:         return T
15:     else
16:         return (R − 1)
17:     end if
18: end function
19: function CHOOSE-RANDOM(R)
20:     while true do
21:         R'←random(1,T)
22:         if (has-work(R') AND (R'≠R)) then
23:             return R'
24:         end if
25:     end while
26: end function
27: function CHOOSE-LARGEST(R)
28:     max-size←0
29:     for all R" ∈ {1, 2, ..., T} AND (R"≠R) do
30:         if (size(R")> max-size) then
31:             R'←R"
32:             max-size←size(R")
33:         end if
34:     end for
35:     return R'
36: end function
37: function CHOOSE-HONEST(R)
38:     remove(rank-threads, R)
39:     while not-empty(rank-threads) do
40:         R'←pop-front(rank-threads)
41:         if (has-work(R') then
42:             push-back(rank-threads, R)
43:             return R'
44:         end if
45:     end while
46: end function
```



(a) LL-B&B parallel algorithm. (b) IVM-B&B parallel algorithm.

Figure 4.8: Illustration of the LL-B&B and a IVM-B&B parallel algorithms using the same work stealing strategy.

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. As shown in function *choose-ring*, this policy has a smaller computational complexity than the three other victim selection policies.

- ***Random victim selection policy***: In this policy, a thread victim R' is randomly selected when a thread R sends 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. As shown in function *choose-ring*, this policy has a higher computational complexity than the ring policy but a smaller computational complexity than the two other victim selection policies.

- ***Largest victim selection policy***: In a B&B algorithm, it is often impossible to determine how a work unit is hard to solve. This policy is based on a simple heuristic to choose the thread with the most difficult work unit to finish. Indeed, the largest policy assumes that probably the larger the size of a work unit 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 LL-based approach, the size of LL 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.

- ***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 probably has a work unit which is more difficult than the work unit of the thread R_2. Therefore, the thread R steals the work from the thread victim R' which did the least recent work stealing operation. 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.

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 thesis, two granularity policies are used and described in the following:

- ***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 LL-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. 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 Tth policy***: Theoretically, steal half policy may not be appropriate for certain victim selection policies. Assuming that four threads where thread 1 has a certain amount of work W, and threads 2, 3 and 4 complete 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 allocate the amount of works $W/2$, $W/4$, $W/8$ and $W/8$ 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.

4.4 Experimentation

This section compares the multi-core LL and IVM-based B&B approaches using different work stealing strategies and granularity policies in terms of speedup, sharing events, memory space, and CPU usage.

4.4.1 Experimental settings

When an instance is solved twice using a multi-threaded B&B, the number of explored subproblems is often different between the two resolutions. To compare the Linked List based and the IVM-based strategies, the number of explored subproblems should be exactly the same between the different tests. Therefore, we chose to always initialize our B&B by the optimal solution of the instance to be solved. Such initialization makes sure that the number of explored subproblems is the same in both approaches, leading to a fair comparison. Therefore, the objective of the resolution is to prove the optimality of the initial solution. Obviously, to provide the optimal solution the same algorithm is used. One has just to intitialize the best solution found so far to infinity for a minimization problem.

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 operator gives such good lower bounds that it is possible to solve them in few seconds using a multi-core B&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. Table 4.1 (resp. Table 4.2) gives the number of explored subproblems for 20x20 (resp. 50x10) instances when initialized with the optimal solution.

Instance	#Subproblems (in millions)
ta021	41.4
ta022	22.0
ta023	140.8
ta024	40.0
ta025	41.4
ta026	71.4
ta027	57.1
ta028	8.0
ta029	6.8
ta030	1.6
Average	**43.1**

Table 4.1: Number of explored subproblems for 20x20 instances when initialized with the optimal solution

Instance	#Subproblems (in millions)
ta041	0.4
ta042	159.0
ta043	40.6
ta044	0.3
ta045	0.8
ta046	2.1
ta047	9.1
ta048	3.8
ta049	0.4
ta050	73.7
Average	**29.0**

Table 4.2: Number of explored subproblems for 50x10 instances when initialized with the optimal solution

Hardware and software testbed:

The multi-core versions of LL-B&B and IVM-B&B have been implemented using C++ and the pthread Posix threads library. Their compilation has been done using GCC 4.6 and $-O3$ optmization option. All the experiments were run on the computer Poincaré which belongs to Maison de la Simulation. 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. For each Flow-Shop instance the computational time spent in managing the pool of subproblems is measured using the *clock_gettime* function with a nanosecond precision.

Experimentation Results Database:

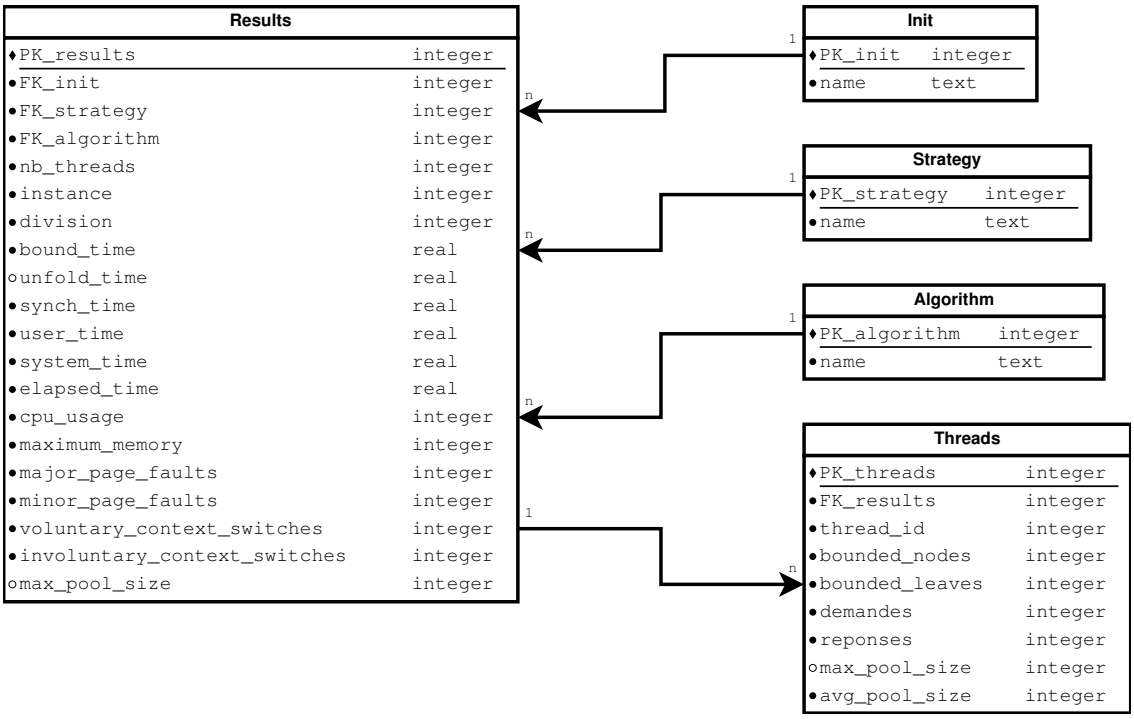


Figure 4.9: Entity-relationship model of the database containing the results of the experiments.

Figure 4.9 is a representation of the relational database which has been used to store the results of the experiments. Each line of the $Results$ table corresponds to one B&B process, it contains all process-level data that is relevant to the experimentation. Each line of the

$Threads$ table corresponds to one thread inside a B&B process, it contains all thread-level data that is relevant to the experimentation. The three remaining tables, $Init$, $Strategy$, and $Algorithm$ contain text data that should not be duplicated in the $Results$ table.

- $Init$ contains the two possible initialization types, $optimal$ and $infinite$.
- $Strategy$ contains the five different combinations of victim selection and granularity policies, $Largest1/2$, $Honest1/2$, $Random1/2$, $Ring1/2$, and $Ring1/T$.
- $Algorithm$ contains the two tested algorithms: IVM-B&B and LL-B&B.

4.4.2 Strategy and granularity policies evaluation

Tables 4.3 reports the experimental results for two metrics for both IVM-B&B and LL-B&B and for the 5 evaluated strategies $Largest1/2$, $Honest1/2$, $Random1/2$, $Ring1/2$, and $Ring1/T$:

- The speedup which compares the wallclock time measured for a serial single-core execution of the B&B algorithm to a multi-core execution using 16 threads.
- The number of sharing events during an execution using 16 threads, i.e. the number of times an interval has been divided in order to be shared with another thread.

A ratio comparing IVM-B&B to LL-B&B is reported. The ratios are computed as LL divided by IVM, which means that in the case of the speedup, a ratio inferior to 1 means that IVM does better than LL, whereas for the number of sharing events, a ratio superior to 1 means that IVM does better than LL.

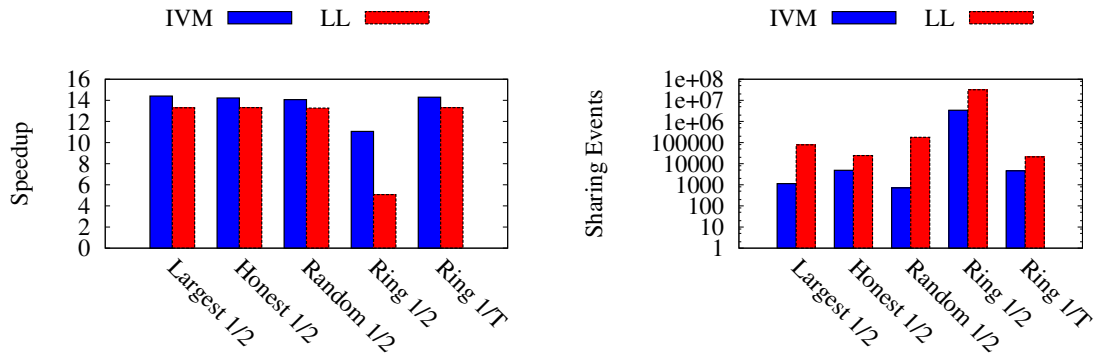
Table 4.3 shows the results for 20 jobs on 20 machines. For the $Largest1/2$ strategy, IVM-B&B has an average speedup of 14.41, and the threads steal an average of 1140.6 intervals. LL-B&B has an average speedup of 13.30, and the threads steal on average 78357.7 intervals. On average, the LL speedup is 0.92 times the speedup of IVM, while the number of sharing events for LL is 68.70 times the sharing events of IVM.

Table 4.3 shows that IVM performs better than LL for all 5 evaluated strategies. Within those strategies, $Ring1/2$ gives the worst speedup for both IVM and LL. For IVM the $Largest1/2$ strategy provides the best speedup, while the $Random1/2$ strategy provides the lowest amount of sharing events. For LL the $Honest1/2$ and $Ring1/T$ strategies provide the best speedup, while the $Ring1/T$ strategy provides the lowest amount of sharing events.

	IVM		LL		Ratio	
Strategy	**Speedup**	**Sharing Events**	**Speedup**	**Sharing Events**	**Speedup (LL/IVM)**	**Sharing Events (LL/IVM)**
Largest 1/2	14.41	1140.60	13.30	78357.70	0.92	68.70
Honest 1/2	14.23	4851.80	13.31	24252.70	0.94	5.00
Random 1/2	14.07	715.90	13.27	177749.10	0.94	248.29
Ring 1/2	11.06	3394303.40	5.07	31707844.60	0.46	9.34
Ring 1/T	14.30	4596.10	13.31	21100.00	0.93	4.59

Table 4.3: Comparison of IVM-B&B and LL-B&B with 16 threads for 20 jobs on 20 machines with an initialization to optimum in terms of speedup and sharing events.

The evaluation of the $Largest1/2$, $Honest1/2$, $Random1/2$, $Ring1/2$, and $Ring1/T$ strategies shows a clear advantage in favor of IVM for all strategies. Figures 4.10 shows a visual representation of the comparison of the speedup and sharing events for all evaluated instances.
The experiments clearly show that in the $Ring1/2$ strategy produces a much higher number of sharing events than the other 4 strategies. The 3 remaining strategies $Honest1/2$, $Random1/2$, and $Ring1/T$ give speedups that are very close to each other within their own version of the B&B algorithm.



(a) Speedup. (b) Sharing Events.

Figure 4.10: Comparison of speedup and sharing events for 20 jobs on 20 machines for IVM and LL.

In order to simplify the memory evaluation in Subsection 4.4.3 and the CPU time evaluation in Subsection 4.4.4, only the $Random1/2$ strategy will be taken into account.

4.4.3 Memory evaluation

Tables 4.4 reports the results for IVM-B&B using the $Random1/2$ strategy with 16 threads. The columns give :

- The size of the IVM data structure in bytes.
- The maximum number of subproblems stored in the pool of subproblems.
- The maximum size of the LL data structure in bytes.
- A ratio comparing IVM-B&B to LL-B&B. The ratios are computed as LL divided by IVM, which means that for both the total memory and the data structure size, a ratio superior to 1 means that LL uses more memory than IVM, whereas a ratio inferior to 1 means that LL uses less memory than IVM.

Table 4.4 reports the results for FSP instances defined by 20 jobs on $Ta020$ machines. For instance 21, IVM-B&B uses a maximum of 14106624 bytes of memory (i.e. 13.5 MB), and the IVM data structure 4320 bytes of memory (i.e. 4.2 KB). LL-B&B uses a maximum of 18481152 bytes of memory (i.e. 17.6 MB), the pool of subproblems stores a maximum of 1799 subproblems, and the LL data structure used a maximum of 35980 bytes of memory (i.e. 35.1 KB). The total memory footprint of LL is 1.31 times the total memory footprint of IVM, while the size of the LL data structure is 8.33 times the size of the IVM data structure.

Table 4.4 shows that IVM has a smaller memory requirement than LL for all 10 instances. The memory requirements for both LL and IVM do not seem to vary much depending on the instance.

Instance	IVM Data Structure size (Bytes)	LL Maximum Pool size (#subproblems)	LL Data Structure size (Bytes)	Data Structure ratio (LL/IVM)
ta021	4336	1799	35980	8.33
ta022		1552	31040	7.19
ta023		1898	37960	8.79
ta024		1686	33720	7.81
ta025		1650	33000	7.64
ta026		1881	37620	8.71
ta027		1698	33960	7.86
ta028		1629	32580	7.54
ta029		1457	29140	6.75
ta030		1466	29320	6.79
Average	4336	1672	33432	**7.74**

Table 4.4: Comparison of IVM-B&B and LL-B&B with 16 threads for 20 jobs on 20 machines in terms of memory usage.

The experiments show a clear advantage for IVM which can be explained by the fact that the space complexity of the IVM data structure is $O(N^2)$, whereas the space complexity of the LL data structure is $O(N^3)$, as shown in the previous chapter. Another advantage of IVM is that its memory footprint does not vary much for problems of a given size, whereas the memory footprint of LL-B&B can vary a lot for larger problems. This predictability makes IVM a good candidate for architectures where the performance is very dependent on memory, such as GPUs for example.

4.4.4 CPU time evaluation

Table 4.5 reports the results for four metrics for both IVM-B&B and LL-B&B using the $Random1/2$ strategy with 16 threads:

- The total time, also known as the wallclock time or the elapsed time, is the time between the start of the program running the B&B algorithm and its end.
- The user time and system time is the sum of the time spent in user space and in kernel space by the program.
- The bound time is the time spent on the bounding operator.

- The pool management time is the user time and system time minus the bound time.

Table 4.5 shows a ratio comparing IVM-B&B to LL-B&B. The ratios are computed as LL divided by IVM, which means that for both the total time and the pool management time, a ratio superior to 1 means that LL uses more CPU time than IVM, whereas a ratio inferior to 1 means that LL uses less CPU time than IVM.

Table 4.5 shows the results for 20 jobs on 20 machines. For instance 21, IVM-B&B needs 1354 seconds to complete its execution, it spends 21585 seconds in user time and system time. The bound computation lasts 21411 seconds which leaves the remaining 174 seconds for pool management. LL-B&B needs 1517 seconds to complete its execution, it spends 24037 seconds in user time and system time. The bound computation lasts 22834 seconds which leaves the remaining 1203 seconds for pool management. LL-B&B needs 1.12 more time to complete its execution and spends 6.92 more time managing the pool than IVM-B&B.

On average, LL-B&B uses 7.16 more time to manage its pool of subproblems, but uses only 1.10 more time than IVM-B&B to complete its execution. This is due to the fact that the vast majority of the time is spent computing the bound, and very little time is spent on actual pool management. IVM-B&B is faster but not by a significant margin.

	IVM				LL					
Inst.	**Total Time (Sec.)**	**User + Sys Time (Sec.)**	**Bound Time (Sec.)**	**Pool Mgmt Time (Sec.)**	**Total Time (Sec.)**	**User + Sys Time (Sec.)**	**Bound Time (Sec.)**	**Pool Mgmt Time (Sec.)**	**Total Time Ratio (LL/IVM)**	**Pool Mgmt Ratio (LL/IVM)**
ta021	1354	21585	21411	174	1517	24037	22834	1203	1.12	6.92
ta022	667	10441	10351	90	742	11708	11055	653	1.11	7.27
ta023	4419	70521	69929	592	4981	79033	74774	4259	1.13	7.20
ta024	1143	17655	17496	159	1263	19990	18823	1167	1.11	7.33
ta025	1508	22024	21857	167	1537	24385	23303	1082	1.02	6.48
ta026	1928	30761	30482	279	2186	34684	32704	1980	1.13	7.09
ta027	1595	25445	25209	236	1822	28866	27077	1789	1.14	7.57
ta028	266	4033	3999	34	286	4501	4259	242	1.08	7.12
ta029	223	3262	3233	28	235	3673	3470	204	1.05	7.21
ta030	51	799	792	7	58	900	848	52	1.14	7.42
Avg.	1315	20653	20476	177	1463	23178	21915	1263	1.10	**7.16**

Table 4.5: Comparison of IVM-B&B and LL-B&B with 16 threads for 20 jobs on 20 machines in terms of CPU Time.

Despite IVM-B&B being consistently faster than LL-B&B for both the management of the pool of subproblems and the total time, the overall difference in performance is not very significant. This is due to the fact that pool management uses far less CPU time than the computation of the bound. The difference in performance would probably be more clear when using a less expensive bound than the one used in these experiments for the Flow-Shop problem.

4.5 Conclusion

The work stealing approaches described in this chapter are based on factorial number system which is presented in Chapter 2, and the use of an Integer-Vector-Matrix (IVM) data structure introduced in the previous chapter. A work stealing (WS) strategy can be defined by its victim selection policy and its granularity policy. The selection policy indicates the thread victim that a thread thief can steal, and the granularity policy determines the amount of stolen work. The four victim selection policies presented in this chapter are ring, random, largest, and honest. In addition, the chapter describes two granularity policies which are the Steal Half and Steal T^{th} policies. Combining these selection and granularity policies, five IVM-based work stealing strategies are defined and compared to their conventional LL-based counterparts on the Flow-Shop scheduling permutation problem. This evaluation is performed in terms of strategy, granularity, CPU time usage and memory.

In terms of speedup and sharing events, the evaluation shows a clear advantage in favor of IVM compared to LL for all strategies except *Largest 1/2*. In addition, the experiments show that in all situations the *Ring 1/2* strategy produces a much higher number of sharing events than the other four strategies, and the *Ring 1/2* strategy often gives the worst speedup. In terms of CPU time, despite the fact that the IVM-based WS strategies are faster than their LL-based counterpart for both the management of the pool of subproblems and the total time, the overall difference in computation time is not very significant. The difference in performance would probably be more clear when using a less expensive bound than the one used in these experiments. In terms of memory, the experimentations show a clear advantage for IVM compared to LL. Another advantage of IVM-B&B is that its memory footprint does not vary much for problems of a given size, whereas the memory footprint of LL-B&B can vary a lot for larger problems. This memory usage pattern is a huge advantageous for many-core architectures, such as GPU and MIC accelerators, where performance depends a lot on memory usage. The next chapter revisits this parallel multi-core IVM-based B&B algorithm for many-core architectures.

CHAPTER 5

Many-core IVM-based B&B

Contents

5.1 Introduction **81**

5.2 Coprocessor-accelerated B&B: the general design **82**

5.3 GPU-based implementation of B&B **84**

5.3.1 Parallelization on GPU 84

5.3.2 Parallelization of B&B for GPU 86

5.4 MIC-based implementation of B&B **88**

5.4.1 Parallelization on Intel Xeon Phi 88

5.4.2 Parallelization of B&B for Intel Xeon Phi 91

5.5 Experimentation **92**

5.5.1 Hardware and software testbed and parameter setting 93

5.5.2 Experimental results 94

5.6 Conclusion **96**

5.1 Introduction

As previously stated, the Johnson's lower bound LB has been used in this work for solving the Flow-Shop permutation problem. The time complexity of Johnson's algorithm for two machines is $O(\text{n} \log \text{n})$, and therefore $O(m^2 \text{ n} \log \text{n})$ for m machines. The computation of LB is consequently time intensive especially for problem instances for which m is high. In order to experimentally evaluate its CPU time, the lower bound has been implemented in [Chakroun 2013c] and investigated using the Taillard's instances [Taillard 1993] with $m = 10, 20$. The results have shown that the time spent by the B&B evaluating the lower bounds of the examined subproblems is on average between 98 % and 99 % of its total execution time. This result demonstrates that the bounding operator needs massively parallel computing.

On the other hand, coprocessors or accelerators are increasingly becoming key building blocks of High Performance Computing platforms. In addition to their energy efficiency, they boost the performance of traditional processors through the combination of a larger number of processing cores, vector-SIMD processing and multi-threading. Actually, the most used accelerators (Top500 ranking of July 2015) are Nvidia GPUs and Intel MIC coprocessors. The former are composed of a large number of slim cores while the latter integrate a relatively smaller number of streamlined largish cores relying on SIMD processing. Today, coprocessors allow to achieve peak performance of the order of one TeraFlops. Nevertheless, it is often difficult for the programmers to extract a large portion of the theoretically available performance. Indeed, the specific features of these coprocessors raise several issues including the optimization of data transfer between the processor and its coprocessor, vectorization, etc. More details on the these hardware features and related challenging issues are given in the next sections.

The objective of this chapter is to revisit the parallel bounding model combined with the parallel tree exploration model of B&B algorithms to allow highly efficient solving of large instances of the Flow-Shop problem on GPU accelerators and Intel MIC Xeon Phi coprocessors. In Section 5.2, we first present the general design of the coprocessor-accelerated B&B. In Section 5.3 (respectively Section 5.4), we describe the implementation of the GPU-accelerated (respectively Phi-accelerated) approach. In Section 5.5, we report some experimental results comparing the two coprocessor-based many-core implementations and their multi-core implementation counterpart. Finally, some conclusions are drawn in Section 5.6.

5.2 Coprocessor-accelerated B&B: the general design

As mentioned above, the coprocessors are many-core devices dedicated to massively parallel computing. Therefore, to take maximum advantage of the computational power provided by these coprocessors these latter should be fed by a large number of computations. In our proposed parallel coprocessor-based approach, as illustrated in Figure 5.1, several B&B trees are explored in order to generate multiple pools maximizing the use of the coprocessor cores. Each pool is implemented using either the IVM data structure or the linked-list. During the exploration of each pool, except the selection and pruning operators which are performed on the processor the branching and bounding operators are executed on the coprocessor. As shown in Figure 5.1, on the processor side, at each iteration of the exploration process a set of tree nodes (whose size is a user-parameter) is selected. The selected set of nodes is offloaded to the coprocessor to

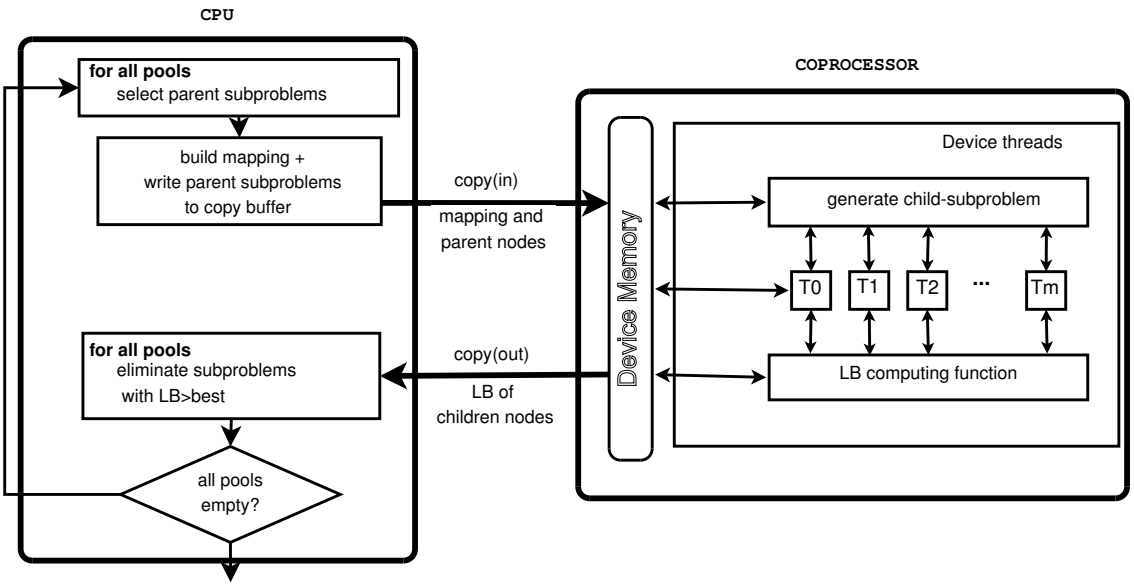


Figure 5.1: Coprocessor-based B&B

be processed.

On the coprocessor side, as illustrated in Algorithm 12, each parent node is processed by a thread. After performing its mapping, the thread branches the parent node if it is not a leaf. The resulting children are evaluated (bounded) and returned back to the CPU with their associated lower bounds. Every child having a lower bound greater than the cost of the best solution found so far is pruned on CPU. All the non-pruned children are inserted into the pools. The process is iterated until the exploration is completed and the optimal solution is found.

Algorithm 12 Kernel of the computation of the lower bounds on the coprocessor.

```
 1: procedure KERNEL EVALUATEONCOPROCESSOR
 2:     (in: parent-subpbs, mapping out: lower bounds of children-subpbs)
 3:     thdId←blockIdx.x*blockDim.x + threadIdx.x
 4:     child-subpb←generateChild(thdId, mapping, parent-subpbs)
 5:     if isLeaf(child-subpb) then
 6:         LB←evaluateSolution(child-subpb)
 7:     else
 8:         LB←computeBound(child-subpb)
 9:     end if
10:     poolOfBounds[thdId]←LB
11: end procedure
```

5.3 GPU-based implementation of B&B

In this section, we first present the parallelization model on GPU. To do that we recall the hardware view of GPU, its parallel programming model and its associated algorithmic challenging issues. Then, we show how these issues are dealt with in the implementation of the GPU-accelerated B&B.

5.3.1 Parallelization on GPU

For a long time, GPU computing has been used to speed up image and video processing. Since 2006, with the introduction by Nvidia of its Cuda software toolkit the use of GPUs has been extended to numerous other application domains including combinatorial optimization. The popularity of Cuda is due to its simplicity as it is an extension of the C language with data parallel features. The principle is easy: the programmer writes a code for one thread (kernel) and can instantiate it on a large number of threads to allow massive parallel computing on GPU. In addition, Cuda is portable between successive generations allowing transparent scalability of Cuda applications.

Before the Cuda parallel model is presented, let us recall the hardware architecture of a GPU device. As shown in Figure 5.2, a GPU is a coprocessor, coupled to a CPU through a PCI Express bus. In the Cuda vocabulary, the processor is called “host" and the GPU is called “device". The GPU is composed by a set of streaming multi-processors (processors) including each a pool of 32-bit or 64-bit SIMD processors (processing cores). For instance, a Kepler GPU device contains 13 processors of 192 Cuda cores for a total of 2496 Cuda cores. A GPU is also composed of several memories including global and local off-chip memories, and a shared memory, registers and a cache memory. These memories have different characteristics in terms of size and access latency. For instance, the global memory is big and has a long latency while registers are small and fast memories.

From software programming point of view, as illustrated in Figure 5.3 a GPU Cuda-based parallel program is composed of two parts: a “host" part and a “device" part. The host part is a serial or weakly parallel code because the number of CPU cores is small compared to the number of GPU cores. The device part is massively parallel because a GPU contains from hundreds to thousands of processing cores. During the execution of a parallel program the host part offloads streams of threads to the GPU device to be executed according to a two-level parallelism: at the higher level the processors (or SMX) execute the thread kernel according to the Single Program Multiple Data (or SPMD)

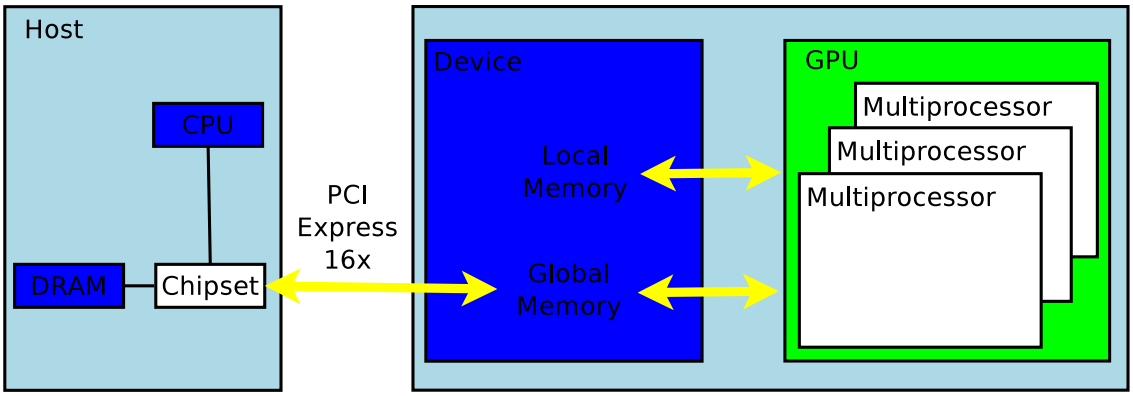


Figure 5.2: Hardware view: GPU = coprocessor of CPU.

model. At the lower level (intra-SMX), the threads are executed according to the Single Instruction Multiple Data (or SIMD) or Single Instruction Multiple Thread (or SIMT) model. Indeed, inside each processor the instruction flow composing a thread kernel is executed according to the SIMD model.

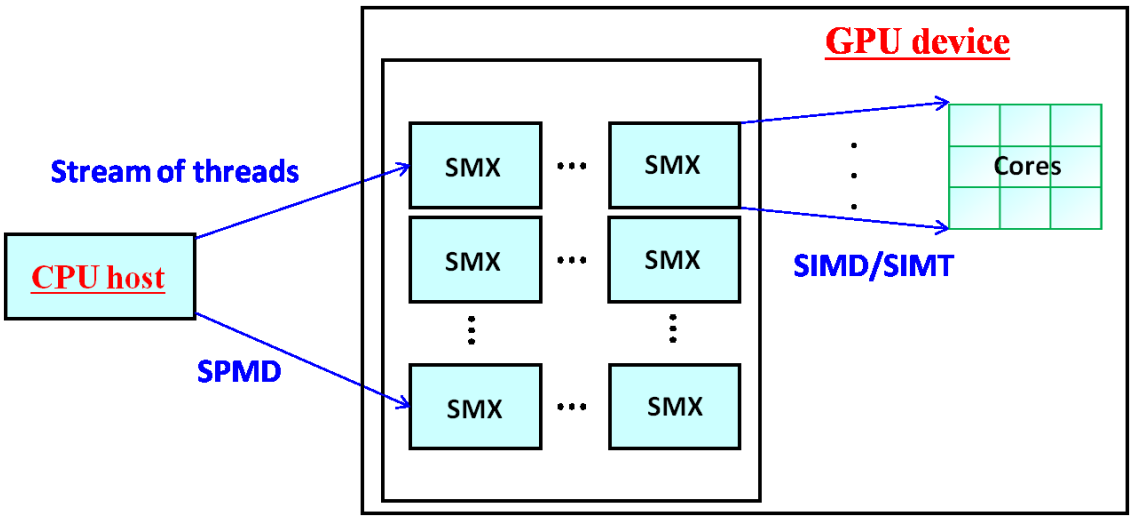


Figure 5.3: Software view: Parallel program = weakly parallel/serial host code + massively parallel device code.

The threads offloaded from CPU host to GPU device are organized by the programmer in a hierarchical way into grids of blocks of threads. Grids are arrays $1D$ or $2D$ of blocks and blocks are arrays $1D$, $2D$ or $3D$ of threads. The thread organization corresponds to the organization of application data which are often vectors, matrices or volumes. As shown in Figure 5.4, the blocks are assigned to the SMXs by the Cuda runtime. Inside each SMX

each block is split into warps i.e. pools of 32 threads. Warps are scheduling units i.e. the threads are executed by pools of 32. This allows to overlap the memory access latency by computation. Context switching is very fast as each thread has its own registers.

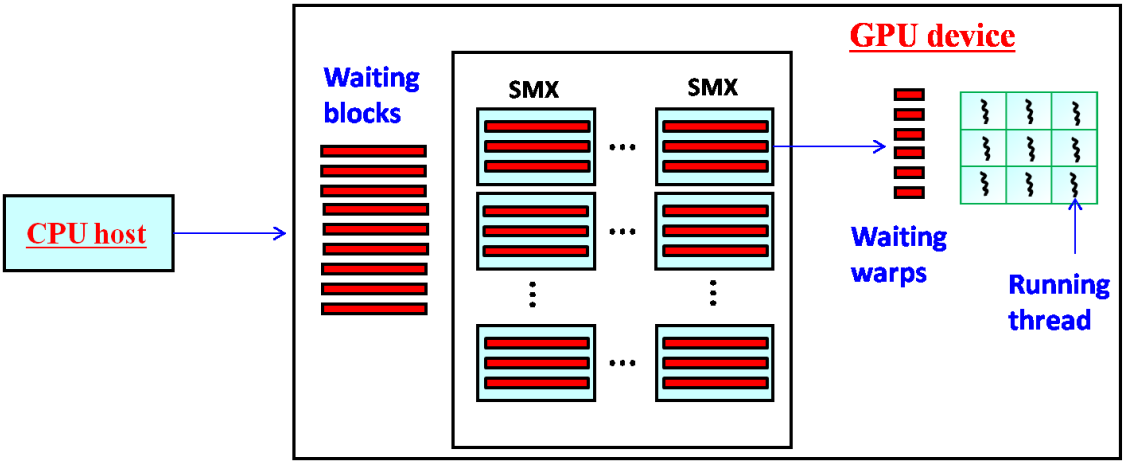


Figure 5.4: Software view: Parallel program = grid(s) of block(s) of threads executed as warps of 32 threads.

To sum up, from algorithmic and software programming point of view at least three issues should be addressed: (1) the optimization of the data transfer between CPU and GPU; (2) the optimization of the data placement on the hierarchy of memories of the GPU having different sizes and latencies; and (3) thread or branch divergence management especially for irregular applications.

5.3.2 Parallelization of B&B for GPU

The implementation on GPU of the coprocessor-based B&B according to the general design presented in Section 5.2 requires to address the challenging issues quoted above. First, to deal with the CPU-GPU data transfer optimization the branching operator, which generates tree nodes or subproblems, is moved to GPU. The execution of the branching operator on the GPU device allows one to avoid the transfer of the branched parent nodes from CPU to GPU which is costly. However, this raises other issues related to thread granularity and mapping. Indeed, if a parent node is processed entirely (branching and bounding) by a single thread there will be a load imbalance leading to thread/branch divergence. In fact, the parent nodes may have different numbers of children as they are located at different levels in the B&B tree. To deal with this problem the processing of each thread is limited to a single node, meaning that each thread generates and evaluates only one child of the parent node.

Second, to tackle the problem related to data placement optimization on GPU we have followed the recommendation proposed in [Melab 2014]. Indeed, as illustrated in Algorithm 13, the implementation of the lower bound algorithm includes 6 data structures: the matrix PTM of the processing times of the jobs, the matrix of lags LM, the Johnson's matrix JM, the matrix RM of the earliest starting times of jobs, the matrix QM of their lowest latency times and the matrix MM containing the couples of machines. The algorithm needs as input a subproblem defined as a permutation with some jobs already scheduled at its beginning and/or its end.

Algorithm 13 Computation of lower bound (un-vectorized)
input: subproblem = permutation, nLeft (#jobs fixed left), nRight (#jobs fixed right)
constant data (MM, JM, PTM, LM)
output: lower bound (LB) of subproblem

```
1:  procedure COMPUTE LB
2:      RM, QM, SM ← InitTabs(permutation, nLeft, nRight)
3:      LB ← 0
4:      for (k = 0 → J(J−1)/2) do
5:          tmp0, tmp1, ma0, ma1 ← InitFun(k, nLeft, MM, RM)
6:          for (j = 0 → J) do
7:              job←JM[k][j]
8:              if (SM[job]==0) then
9:                  tmp0 += PTM[ma0][job]
10:                 tmp1 = max(tmp1, tmp0 + LM[k][job]) + PTM[ma1][job]
11:             end if
12:         end for
13:         tmp1←EndFun(tmp0, tmp1, k, nRight, QM)
14:         LB = max(tmp1, LB)
15:     end for
16:     return LB
17: end procedure
```

The semantics of these data structures is not the focus of this thesis. For more details on these ones and on the lower bound please refer to the Ph.D thesis of I. Chakroun [Chakroun 2013c]. The focus is rather put here on the optimization of the placement of these data structures on the different memories of the GPU device. Due to the limited size of the shared memory, the matrices do not fit in all together, especially for large problem instances. Based on the complexity analysis of these data structures and an experimental study conducted in [Chakroun 2013c], it is suggested to put in the shared memory the Johnson's and/or processing time matrices (JM and PTM). The other data structures

are mapped either to the global memory or to the constant memory. Such data placement allows one to achieve accelerations of more than $\times 100$ compared to a single-CPU core serial execution of B&B. In our implementation, MM, JM, PTM and LM are put on the constant memory. A part of PTM is then moved to the shared memory. The other matrices are stored on the global memory.

Third, the parallelization of irregular applications such as B&B applied to permutation problems due to the thread or branch divergence issue [Chakroun 2013b]. In our implementation, the irregularity is due to two factors: as the tree nodes have different levels they require different amounts of work (number of children) ; on the other hand, the lower bound function, as it can be seen in Algorithm 13, includes several conditional instructions and loops. To deal with the first factor, each thread handles only one child as mentioned above. Therefore, all the threads perform the same amount of work. To tackle the second problem, we have reused the refactoring approach proposed in [Chakroun 2013b] even if the achieved performance improvement is not significant as the size of the factorized branches is small.

Finally, the mechanisms used on GPU are not experimented individually here because their efficiency has been demonstrated in [Chakroun 2013b, Melab 2014, Chakroun 2013c]. These three citations can be used for further details on the mechanisms. However, the performance of the whole GPU-accelerated B&B is evaluated and compared to the performance of the Xeon Phi-based B&B in Section 5.5.

5.4 MIC-based implementation of B&B

In this section, we first present the parallelization model on MIC architecture. To do that we recall the hardware view of Intel Xeon Phi, its parallel programming model and its associated algorithmic challenging issues. Then, we show how these issues are dealt with in the implementation of the Intel Xeon Phi-accelerated B&B.

5.4.1 Parallelization on Intel Xeon Phi

The market of accelerators has been dominated by Nvidia during several years. Since recently, they are faced to the competition of Intel with its Many Integrated Cores (MIC)-based Xeon Phi. This latter is a coprocessor coupled to the processor through a PCI Express bus. As illustrated in Figure 5.5, a typical platform consists of one to two Intel Xeon processor(s) (CPUs) and one to eight (two in this figure) Intel Xeon Phi coprocessors per

host. Multiple such platforms may be interconnected to form a cluster or supercomputer.

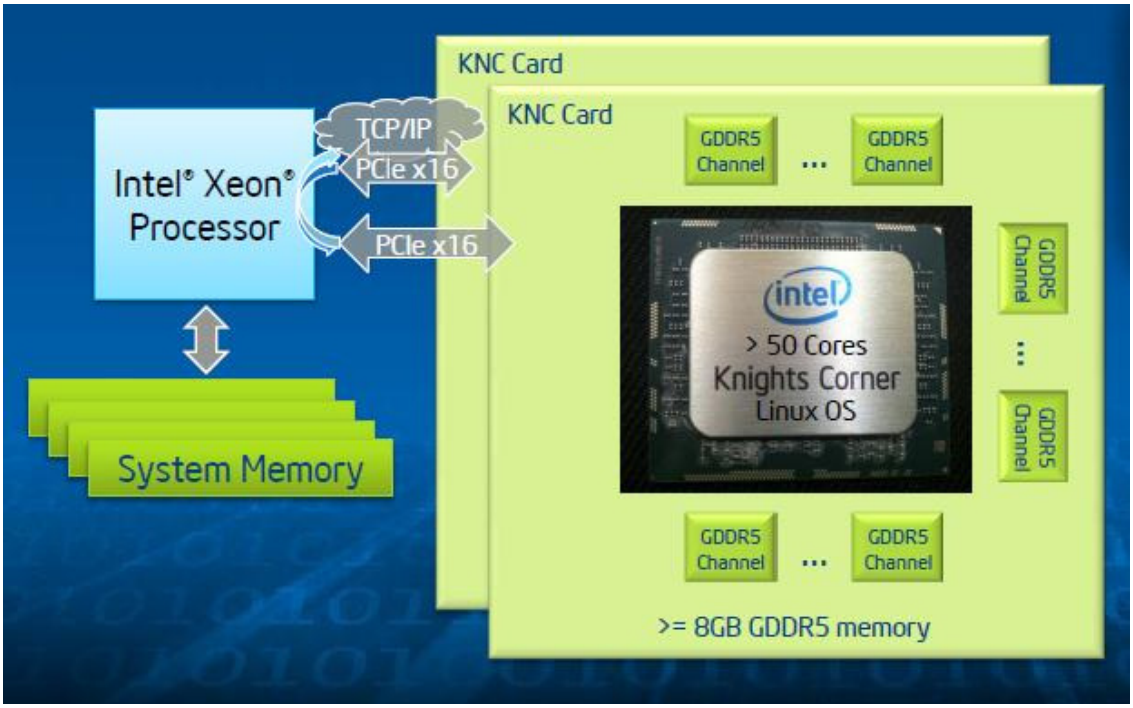


Figure 5.5: Hardware view: Intel Xeon Phi = coprocessor of CPU.

From the hardware point of view, the Xeon Phi board has one Knights Corner (KNC) processor, the first production chip based on the MIC architecture, and 8 GB of GDDR5 RAM. As illustrated in Figure 5.6, KNC integrates up to 61 CPU-cores interconnected by a high-speed bi-directional ring, and runs at over 1 GHz. It connects to its private external memory with a peak bandwidth of over 320 Gbps. The cores are based on the Intel Pentium architecture. Each core has 32 KB of L1 data and instruction cache, 512 KB of L2 data cache, and a 512-bit vector Floating Point Unit (FPU). This latter performs fused-multiply-add (FMA) operations. Therefore, the peak performance is about 32 (resp. 16) GFlops in single (resp. double) precision. Consequently, the KNC delivers a peak performance of about 2 (resp. 1) TFlops in single (resp. double) precision.

From a programming standpoint, the key is to treat the Intel Xeon Phi coprocessor as an x86-based SMP-on-a-chip with over 50 cores, with multiple threads per core and 512-bit SIMD instructions. From programming language point of view, Intel Xeon Phi is more accessible than Nvidia GPU because it can be programmed using standard programming environments such as OpenMP, MPI, Cilk Plus and Posix Threads. However, to achieve higher performance one should consider two fundamental features: scaling through locality and Simultaneous Multi-Threading (SMT) and vectorization. On the other hand, as an

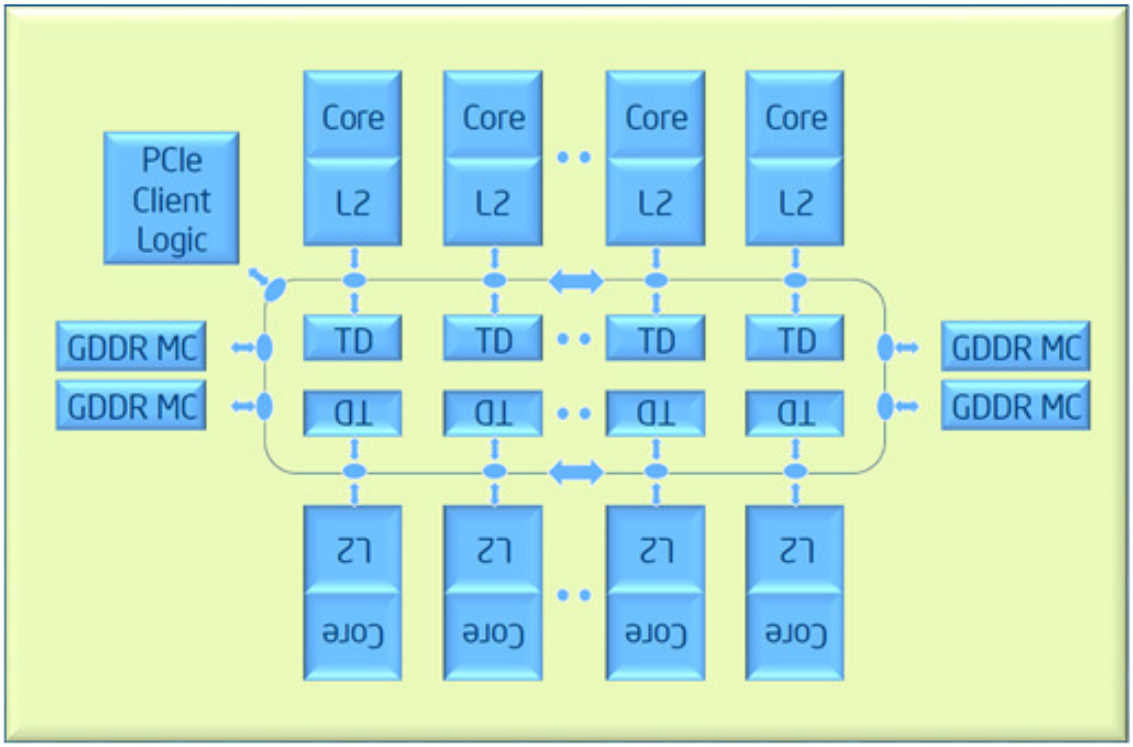


Figure 5.6: Hardware view: Knights Corner core.

Intel Xeon Phi coprocessor runs an operating system (Linux) and has its own IP address, there are two ways to involve it in a parallel program:

- A processor-centric “offload" mode where the program is viewed, like for GPUs, as running on processors and offloading work to coprocessors. Another issue which arises with the offload mode is, like for GPU, the optimization of the data transfer between the processor and the coprocessor.

- A “native" mode where the program runs natively on only coprocessors or on coprocessors and processors together. In the latter case, the two devices may communicate with each other by various methods. At least two other issues arise in this case: the optimization of the data/task partitioning and the communication between the processor and the coprocessor. The challenge for data/task partitioning is the load balancing between the “big cores" of the processor and the “little" cores of the coprocessor. Regarding the communication, the challenge is to overlap the communications by the computation, to manage data locality, etc.

5.4.2 Parallelization of B&B for Intel Xeon Phi

In this chapter, we investigate the offload and native modes for the parallelization of B&B algorithms on Intel Xeon Phi coprocessors. In the native-based parallelization approach, the random work stealing strategy proposed for multi-core processors presented in the previous chapter has been reused for Phi coprocessor independently (without processor). The objective is to study the performance of the two data structures IVM and LL and the scalability of the associated work stealing strategy. Indeed, the number of processing cores is higher in Xeon Phi coprocessor than in Xeon processor. This allows to study the behavior of the scalability beyond 16 threads. In the offload-based parallelization approach, the processor-coprocessor data transfer optimization approach is the same as for GPU. Indeed, the branching operator is performed on Phi, this allows one to reduce the cost of data transfer between the two devices. On the other hand, all the data structures which are not modified between offloading operations are offloaded once.

One of the major mechanisms allowing performance improvement on Intel Xeon Phi is vectorization. Different levels are provided ranging from compiler-based automatic easy-to-use vectorization to manual and programmer control vectorization. In our work, as quoted previously, the most consuming part of the B&B algorithm is computation of the lower bound function (Algorithm 13). In this chapter, we propose a vectorization method of the lower bound function focusing on its most compute-intensive portion and the main data-dependencies. This portion of code is the inner for-loop (line 6-12) which consumes about 70% of the bounding time. The body of this inner loop is executed $J^2 \times (J-1)/2$ times. Regarding data dependencies, the statement in line 10, including a dependency of current $tmp1$ on $tmp1$ from previous iteration) prevents vectorization (icc do not auto-vectorize it). In addition, except for line 14 the iterations of the outer loop are independent (private variables: tmp0, tmp1, ind0, ind1, current). However, only the inner loop of a loop nest may be vectorized [1].

In order to allow more vectorization than provided by the compiler, the order of the nested loops must be inverted as illustrated in the vectorized lower bound function (Algorithm 14).

[1] http://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf

Algorithm 14 Computation of lower bound (vectorized)
input: subproblem = permutation, nLeft (#jobs fixed left), nRight (#jobs fixed right)
constant data (MM, JM, PTM, LM)
output: lower bound (LB) of subproblem

```
1: procedure COMPUTE LB
2:     RM, QM, SM ← InitTabs(permutation, nLeft, nRight)
3:     LB ← 0
4:     for (k = 0 → J(J−1)/2) do
5:         Tmp0[k], Tmp1[k], Ma0[k], Ma1[k] ← InitFun(k, nLeft, MM, RM)
6:     end for
7:     for (j = 0 → J) do                                        ▷ permute loop-order
8:         #pragma ivdep
9:         for (k = 0 → J(J−1)/2) do                             ▷ inner loop vectorizable
10:            job←JM[j][k]                                      ▷ transpose JM
11:            if (SM[job]==0) then
12:                Tmp0[k] += PTM[Ma0[k]][job]
13:                Tmp1[k] ← max(Tmp1[k], Tmp0[k] + LM[k][job]) + PTM[Ma1[k]][job]
14:            end if
15:        end for
16:    end for
17:    for (k = 0 → J(J−1)/2) do
18:        Tmp1[k]←EndFun(Tmp0[k], Tmp1[k], k, nRight, QM)
19:    end for
20:    LB←max-reduce(Tmp1[])
21:    return LB
22: end procedure
```

For auto-vectorization by the compiler it is preferable to write small separate loops, rather than merging into a single loop. The outer loop is thus split into 3 separate serial loops and a max-reduce operation (line 20) in order to isolate the k-dependent instructions from the inner-loop. The cost to pay for this is to declare the scalars $tmp0$, $tmp1$, $ma0$, $ma1$ as arrays (resp. $Tmp0$, $Tmp1$, $Ma0$, $Ma1$) of size K. The same strategy on GPU severely breaks down performance due to the memory problem (these intermediate variables are no longer stored into registers). Even with the highest optimization level activated ($-O3$) the Intel compiler (icc) still needs the hint "#pragma ivdep to vectorize the inner loop (line 9) successfully. The two other for-loop are auto-vectorized.

5.5 Experimentation

In this section, we present an experimental study of the proposed many-core approaches using GPU and Intel Xeon Phi and compare them to their multi-core counterpart. We first present the hardware and software testbeds and some parameter setting used for our experiments. Then we report and discuss some experimental results.

5.5.1 Hardware and software testbed and parameter setting

For the multi-core implementation, we have used OpenMP Version 4.0. All the experiments are run on hardware described in Table 5.1. The compiler used for the CPU and MIC implementations is Intel icc version 15.0 for Intel devices. For the GPU-accelerated implementation, the NVIDIA CUDA Toolkit release 5.0.35 is used together with the gcc version 4.4.7. For all experiments the compilation level 3 (-O3) is used. On the other hand, the UNIX `time` command is used to measure the elapsed execution time for each Flow-Shop instance. Finally, the GFLOPS(DP) row is obtained using the following computations:

- MIC: $16(flops/Ghz) * 60(cores) * 1.053(GHz/core) = 1010.88GF/s$
- CPU: $8(flops/Ghz) * 8(cores) * 2.6(GHz/core) = 166GF/s$
- K20x: $14(SM) * 64(cores/SM) * 0.732(GHz/core) * 2(flops/Ghz)(FMA) = 1311GF/s$

	Intel Xeon E5-2670	NVIDIA Tesla K20x	Intel Xeon-Phi 5110P
#Physical cores	8	14	60
#Logical cores	16	2688	240
clock(Ghz)	2.6	0.732	1.053
GFLOPS(DP)	166	1311	1011
SIMD	256-bit	N/A	512-bit
Cache(MB)	20	1.5	30
Mem BW(GB/s)	51.2	250	320
Watt	115	235	225

Table 5.1: Hardware execution platform

The last raw indicates that the three hardware configurations (considering a two-socket multi-core processor) are equivalent in terms of energy consumption.

As quoted in Section 5.2, the coprocessors are many-core devices dedicated to massively parallel computing. Therefore, they need to be fed by a large number of computations. To do that, many B&B trees are explored in order to generate multiple (IVM-based) pools maximizing the use of the coprocessor cores. Before the experiments are performed, the number M of IVM-based pools to be created is calibrated through a series of experiments on the problem instance $Ta028$. The experimental results are reported in

Figure 5.7. Based on the figure, the number of IVMs is fixed to 1000 (resp. 1600) for the GPU-accelerated (resp. Xeon Phi-based) approach.

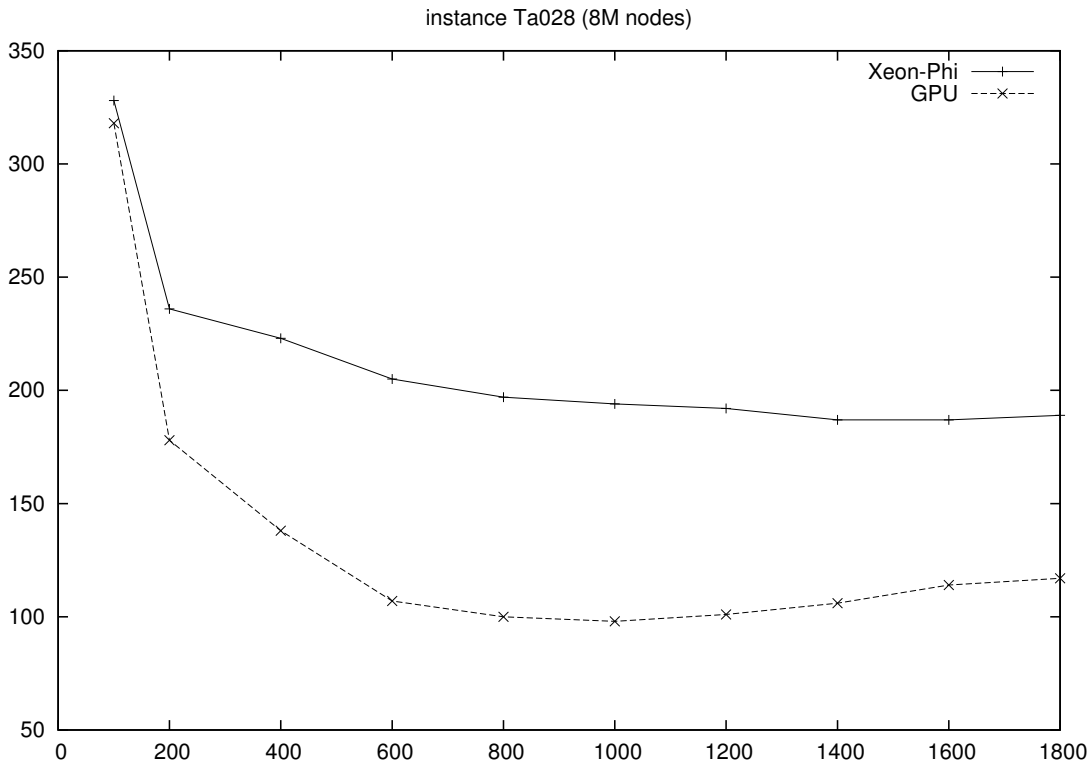


Figure 5.7: Elapsed time vs. Number of IVMs.

5.5.2 Experimental results

The performance analysis of the contributions of this chapter is done in two steps. The first one consists in comparing the performance of the IVM-based parallel multi-core approach and the two many-core approaches: the GPU-accelerated approach and the MIC-accelerated approach. In order to get a fair comparison between the two many-core approaches the offload mode is considered for the MIC-based approach. In the second step of the performance analysis, we have considered the native mode of the MIC-based approach. The objective is to compare the private IVM and private linked-list-based approaches in terms of efficiency (execution time) and scalability according to the number of Intel Xeon Phi threads.

Performance comparison between multi and many-core approaches

As quoted above, the objective is to compare the many-core GPU and Phi-accelerated approaches between them and to their multi-core counterpart. The multi-core approach is exactly the same than the coprocessor master-worker approach except that the bounding and branching operators are performed on the processing cores of the CPU. The implementation is vectorized using the same method as for the MIC-based approach. In our experiments, hyper-threading (2 threads per physical core) is used for the multi-core approach and simultaneous multi-threading (SMT with 2 threads per core) is considered for the MIC-based approach. The different approaches have been experimented using the 10 instances ($Ta021 - Ta030$) of the Taillard's problem 20 jobs on 20 machines. The best solution found so far is initialized to the optimal solution to guarantee that the amount of work (explored nodes) is the same for each of the experimented approaches. This allows to prevent from the well-known speed up anomalies investigated for instance in [Mans 1996]. The obtained experimental results are reported in table 5.2.

	Nodes	GPU	Xeon Phi		Multi-core	
Inst.	($\times 10^6$)	-	No-Vect	Vect	No-Vect	Vect
21	41.4	549	1486	917	2065	1362
22	22.1	293	819	468	1015	670
23	140.8	1698	5143	2956	6926	4678
24	40.1	434	1276	744	1667	1173
25	41.4	491	1678	964	2121	1338
26	71.4	774	2273	1334	2930	2019
27	57.1	610	1732	998	2357	1720
28	8.1	100	319	187	394	258
29	6.8	83	256	154	394	258
30	1.6	26	75	42	77	53
AVG	**43.1**	**506**	**1506**	**876**	**1987**	**1349**

Table 5.2: Exploration time (in seconds) for solving Flow-Shop instances *Ta021-Ta030* initialized at optimal solution

The first two columns of the table contain respectively the numbers of the 10 solved problem instances and their associated search space sizes in millions of nodes. The following columns designate the exploration time in seconds obtained using respectively the

GPU-accelerated approach, the vectorized and non-vectorized MIC-based approaches using 120 threads, and the vectorized multi-core approach using 16 threads. From the last row of the table, three major observations can be made. First, the GPU-accelerated approach outperforms the MIC-based approach even in its vectorized version. Second, vectorization allows one to speed up the MIC-based approach with a factor of two. Finally, the many-core approaches are faster than the multi-core approach. Indeed, the GPU-accelerated (resp. MIC-based) approach is 5 (resp. 3) times faster than its multi-core counterpart.

Performance analysis using the native MIC-based approach

The performance analysis using the native MIC-based approach is done in two steps. The first one consists in comparing the private IVM and private linked-list-based approaches in terms of scalability according to the number of Phi hardware threads. In this experimentation step we have considered only the smallest Flow-Shop problem instance with 20 jobs (Ta030). This instance generates 1.6 million of subproblems at execution. Figure 5.8 reports the evolution of the elapsed CPU time according to the number of Xeon Phi hardware threads for the two approaches. The figure shows that the IVM-based approach is faster and scales better than its linked-list counterpart. The IVM-based approach scales nicely up to 120 hardware threads and remains constant above. The linked-list approach scales well up to 60 hardware threads meaning that it does not support hyper-threading.

In the second step of the performance analysis, we have compared the CPU elapsed time obtained on the 10 problem instances for the two approaches. The experimental results are reported in Table 5.3. The table shows that in average the IVM-based approach is about 10 times faster than the linked-list approach.

The number of page faults is one of the main explanations of the good performance of IVM compared to the linked-list data structure. Indeed, the IVM version has a smaller memory footprint, which means fewer page faults than the linked-list version. As shown in Figure 5.9, for example when using 240 threads, the linked-list version generated $8,632$ page faults which is about 1.34 times more than the IVM-based version where $11,588$ page faults were generated.

5.6 Conclusion

According to the recent Top500 ranking (July 2015), it is confirmed that hybrid HPC platforms including coprocessors is the trend towards the exascale era. On the other hand, it appears that the market of hybrid HPC is dominated by Nvidia followed by Intel with its Xeon Phi. In this chapter, we have revisited the parallelization of B&B algorithms

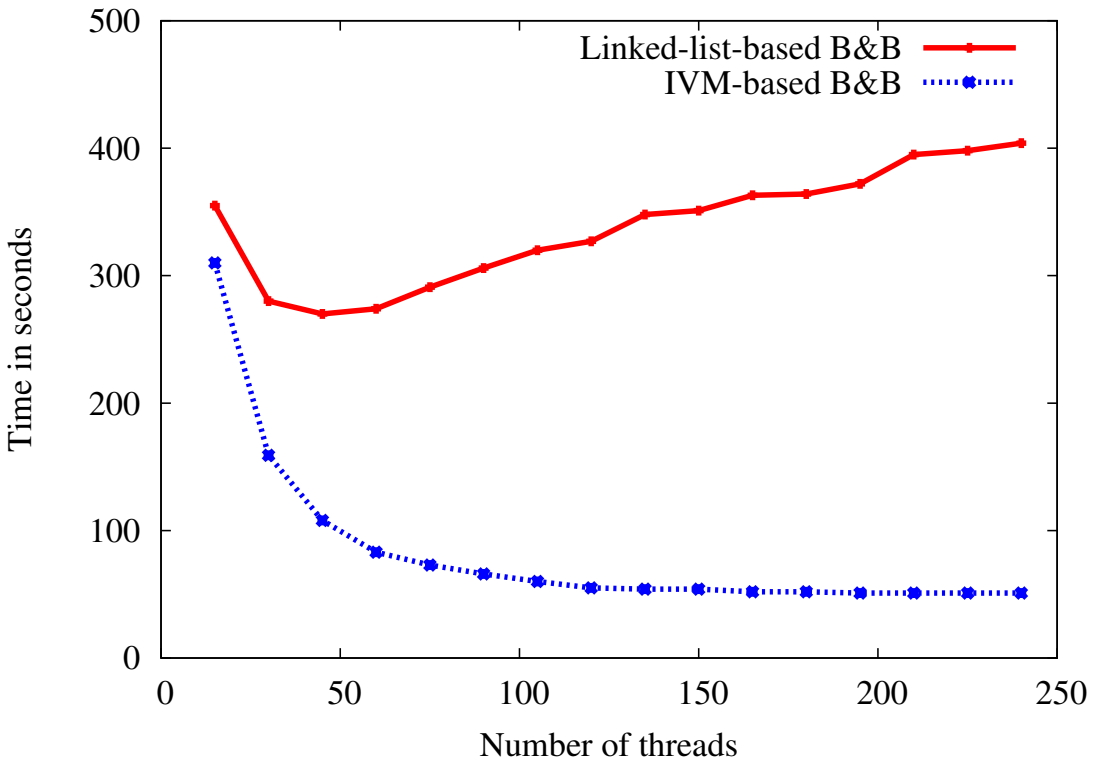


Figure 5.8: Comparison between the linked-list and IVM-based B&B in terms of elapsed CPU time when solving instance 30 using different number of hardware threads.

for many-core coprocessors, in particular Nvidia GPU and Intel Xeon Phi. From the design point of view, we have combined two hierarchical parallel models: the parallel tree exploration model and the parallel bounding. The bounding operator is performed on the coprocessor because, on the one hand, it is the most time-consuming part of the B&B algorithm. On the other hand, it is massively data parallel and thus well-suited for coprocessors. In addition, the branching operator, which generates tree nodes during the exploration process, is also performed on the coprocessor to minimize the cost of their offloading from the processor to the coprocessor.

Such coprocessor-based design of B&B algorithms gives rise to other issues: thread mapping, thread/branch divergence and data placement optimization on GPU, and vectorization on Intel Xeon Phi. From the GPU side, we have reused some recommendations proposed in [Chakroun 2013c]. For the MIC-based approach, we have proposed a vectorization method for the lower bound function. The different implementations have been experimented on the 10 instances of the 20 jobs-on-20 machines problem

# Instance	Linked-list (Minutes)	IVM (Minutes)	Ratio (LL/IVM)
ta021	171	19	9.0
ta022	88	10	8.8
ta023	608	64	9.5
ta024	154	16	9.6
ta025	162	20	8.1
ta026	273	28	9.7
ta027	235	22	10.7
ta028	33	4	8.2
ta029	28	3	9.3
ta030	7	1	7.0
Average	**171.9**	**18.7**	**9.4**

Table 5.3: Comparison between the linked-list and the IVM-based B&B in terms of elapsed CPU time when solving instances with 240 threads

using equivalent hardware configurations in terms of energy consumption. The reported results show that the GPU-accelerated approach outperforms the MIC offload-based one even in its vectorized version. Moreover, vectorization improves the efficiency of the MIC offload-based approach with a factor of two. Finally, the many-core approaches are faster (5 times for NVIDIA Tesla K20x and 3 times for Intel Xeon Phi 5110P) than their multi-core counterpart (using Intel Xeon E5-2670).

Finally, we have proposed a MIC-based using a native mode. The same multi-core approach proposed in the previous chapter has been deployed on the Intel Xeon Phi coprocessor using the work stealing strategy. The approach has been extensively experimented on an Intel Xeon Phi 5110P. The reported results show that the IVM-based work stealing approach is about 10 times faster than the linked-list traditionally used for parallel B&B.

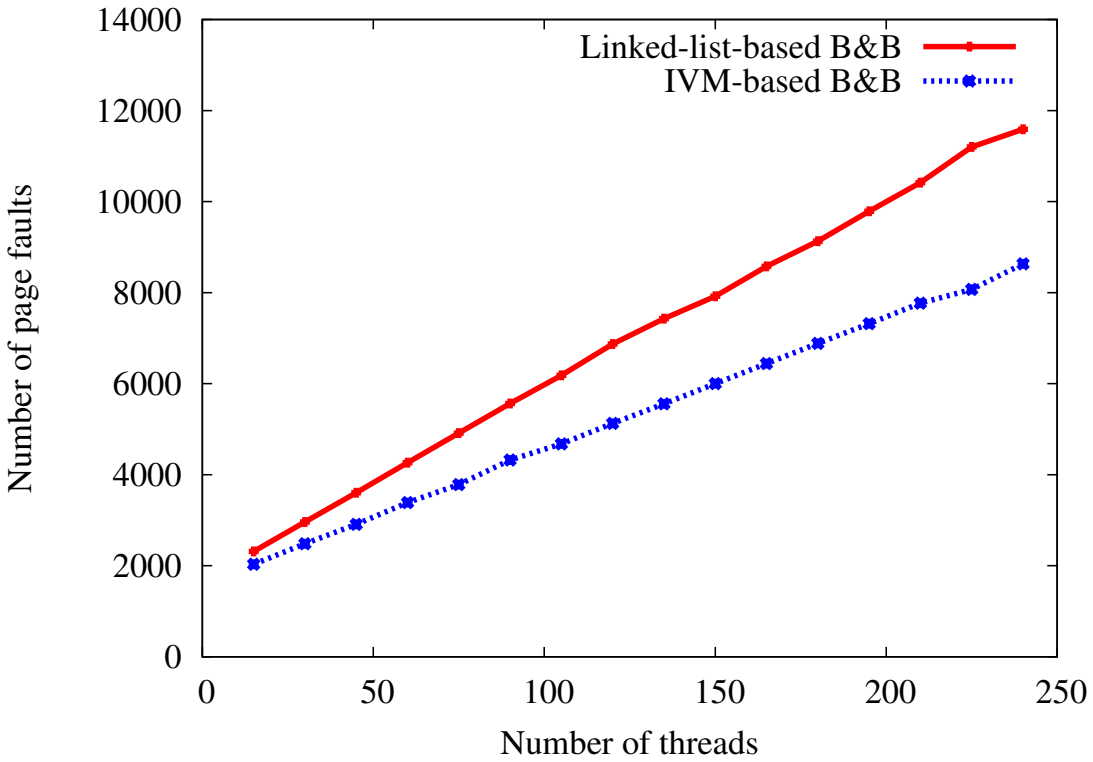


Figure 5.9: Comparison between the linked-list and the IVM-based B&B in terms of the number page faults when solving instance 30 using different number of hardware threads.

CHAPTER 6

Conclusions and perspectives

According to the Top500 international ranking [1] the future of High Performance Computing technologies is hybrid combining multi-core and many-core devices. Indeed, in the last ranking (June 2015) 97% of the systems use processors with 6 or more cores and 87.8% use 8 or more cores. In addition, a total of 90 systems on the list are using accelerator/co-processor technology. Fifty-two of these use Nvidia chips, 35 systems include Intel MIC technology (Xeon Phi), and 4 systems use a combination of Nvidia and Intel Xeon Phi accelerators/coprocessors. In this thesis, the focus is put on parallel tree-based exact combinatorial optimization. Indeed, we have revisited the design and implementation of B&B algorithms for multi-core processors and coprocessors. We have considered Nvidia GPU and Intel MIC Xeon Phi as accelerators/coprocessors in our study. Without loss of generality, FSP has been considered as a case study to validate our approaches.

Solving large permutation COPs using B&B algorithms results in the generation of a very large pool of subproblems. Defining an efficient data structure is highly required to store and manage efficiently that pool [Roucairol 1996, Shavit 2011]. In this thesis, we have first proposed an original data structure called IVM for permutation COPs and consequently redefined the operators of the B&B algorithm acting on it. At any moment of execution, IVM indicates the subproblems of the pool associated to the COP being solved that are being examined and those that remain to be visited. To evaluate the performance of IVM it has been experimented using a serial B&B applied to FSP and compared to the LL data structure in terms of memory consumption and execution time. The reported results show that IVM-based B&B outperforms LL-based B&B in terms of memory as well as computation time. Indeed, compared to LL, on average the IVM data structure requires 9 (resp. 35) times less memory space for the instances defined with 20 (resp. 50) jobs. Moreover, the IVM-based approach is 2.5 (resp. 3) times faster than its LL-based counterpart for instances with 20 (resp. 50) jobs.

[1] Top500 gives the tendency of the evolution of High Performance Computing technologies.

As a second contribution, we have revisited the Work Stealing (WS) mechanism on multi-core processors. Most of existing works related to multi-core B&B use concurrent data structures mainly a stack for depth-first exploration [Chakroun 2013c]. In addition, work units are defined as subsets of nodes making their communication costly. In such works, at each stealing operation each thread picks a subproblem from the shared pool and the generated subproblems are returned back to be inserted in the shared pool. In this thesis, work units are coded in a coalesced thus optimized way, using factoradic-based intervals. Furthermore, as recommended in [Acar 2013], private IVMs are used to store and explore locally subsets of subproblems. The basic principle of this approach is that each thread explores locally a set of subproblems using its private IVM. When the set is empty it steals a work unit from another thread according to the victim selection and granularity policies. Choosing the adequate WS strategy is a great challenging issue and has a great impact on performance. This is why we have investigated different WS strategies combining two granularity policies (half and T^{th}, T being the number of involved threads) and 4 victim selection policies: ring, random, largest and honest. The IVM-based approach has been experimented and compared to the private LL-based approach considering these different WS strategies. To do that, we have implemented this later which is itself a new contribution. The reported experimental results show that our approach is more efficient in terms of memory usage and management time. Indeed, according to the reported results except for largest steal-half WS strategy IVM gives better speed-up than LL. In particular, IVM allows a faster pool management than LL. Moreover, the results confirm that the random steal-half strategy, often used in the literature, outperforms the other WS strategies for large FSP instances. On the other hand, the experimental results confirm that IVM offers a predictable and lower memory footprint than LL.

Finally, we have revisited the parallelization of B&B algorithms for many-core coprocessors, in particular Nvidia GPU and Intel Xeon Phi. Two parallel models have been combined: the parallel tree exploration model and the parallel bounding. The bounding operator is performed on the coprocessor as it is costly and massively data parallel and thus well-suited for coprocessors. Furthermore, the branching operator is also performed on the coprocessor to minimize the cost of their offloading from the processor to the coprocessor. Such design raises some issues mainly thread mapping, thread/branch divergence and data placement optimization on GPU, and vectorization on Intel Xeon Phi. From the GPU side, we have reused some recommendations proposed in [Chakroun 2013c]. For the MIC-based approach, we have proposed a vectorization method for the lower bound function. The many-core implementations have been

experimented on the 10 instances of the 20 jobs-on-20 machines problem using equivalent multi-core and many-core hardware configurations in terms of energy consumption. The reported results show that the GPU-accelerated approach outperforms the MIC offload-based one even in its vectorized version. Moreover, vectorization improves the efficiency of the MIC offload-based approach with a factor of two. Finally, the many-core approaches are faster (5 times for NVIDIA Tesla K20x and 3 times for Intel Xeon Phi 5110P) than their multi-core counterpart (using Intel Xeon E5-2670).

Finally, we have proposed a MIC-based B&B using a native mode. The same multi-core approach proposed in the previous chapter has been deployed on the Intel Xeon Phi coprocessor using the work stealing strategy. The approach has been extensively experimented on an Intel Xeon Phi 5110P. The reported results show that the IVM-based work stealing approach is about 10 times faster than its LL-based counterpart.

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

- As a short-term future work, we will validate our approaches on other single permutation COPs such as TSP, Job-Shop and QAP, and COPs with more than one permutation like Q3AP [Mehdi 2011]. This work will be conducted within the framework of a collaboration with Tiago Carneiro Passoa from University of Ceará-Brazil. This later is invited for one year in the Dolphin team since September 2015. We will also revisit the GPU-accelerated approach so that the whole IVM-based B&B algorithm is executed on GPU. This is an ongoing work realized within the framework of the Ph.D thesis of Jan Gmys started in February 2015.

- As a medium-term future work, we will extend our contributions by combining the multi-core, GPU-accelerated and MIC-based approaches in a single hybrid one. Work partitioning among the three chips will be particularly a challenging issue to be addressed. Then, we will extend this hybrid approach with a cluster-level parallelism. The processor and coprocessor-level work stealing strategies will be investigated at the cluster level. In addition, the B&B@Grid approach proposed in [Mezmaz 2007b] will be reused. The CPU-level checkpointing mechanism proposed in B&B@Grid will be used to deal with failures even if the CPU is coupled with a coprocessor device. The mechanism should be revisited to take into account coprocessor-level failures. This will allow us to solve to optimality unsolved problem

instances [2] using large hybrid clusters. This a also a part of the Ph.D thesis of Jan Gmys.

- We believe that the conclusions drawn from the experiments are the same whatever is the shape of (how irregular is) the tree and thus for any tree-based application. Therefore, in the long term, we will try to generalize the approach to other tree-based exploration algorithms such as B&X (X=cut, price, ...).

[2] as we did it in 2007 using grid computing [Mezmaz 2007b].

Bibliography

[Acar 2013] Umut A. Acar, Arthur Chargueraud and Mike Rainey. *Scheduling Parallel Programs by Work Stealing with Private Deques*. In Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM.

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

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

[Angel 2002] Eric Angel and Vassilis Zissimopoulos. *On the Hardness of the Quadratic Assignment Problem with Metaheuristics*. Journal of Heuristics, vol. 8, no. 4, pages 399–414, July 2002.

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

[Barreto 2010] L. Barreto and M. Bauer. *Parallel Branch and Bound Algorithm-A comparison between serial, OpenMP and MPI implementations*. In Journal of Physics: Conference Series, volume 256, page 012018. IOP Publishing, 2010.

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

[Bierwirth 1996] Christian Bierwirth, DirkC. Mattfeld and Herbert Kopfer. *On permutation representations for scheduling problems*. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg and Hans-Paul Schwefel, editeurs, Parallel Problem Solving from Nature — PPSN IV, volume 1141 of *Lecture Notes in Computer Science*, pages 310–318. Springer Berlin Heidelberg, 1996.

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

[Brucker 2007] Peter Brucker and P Brucker. Scheduling algorithms, volume 3. Springer, 2007.

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

[Casado 2008] L.G. Casado, J.A. Martinez, I. Garcia and EMT Hendrix. *Branch-and-bound interval global optimization on shared memory multiprocessors*. Optimization Methods & Software, vol. 23, no. 5, pages 689–701, 2008.

[Chakroun 2013a] I. Chakroun, N. Melab, M. Mezmaz and D. Tuyttens. *Combining multi-core and GPU computing for solving combinatorial optimization problems*. Journal of Parallel Distributed Computing, vol. 73, no. 12, pages 1563–1577, 2013.

[Chakroun 2013b] I. Chakroun, M. Mezmaz, N. Melab and A. Bendjoudi. *Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm*. Concurrency and Computation: Practice and Experience, vol. 25, no. 8, pages 1121–1136, 2013.

[Chakroun 2013c] Imen Chakroun. *Parallel heterogeneous Branch and Bound algorithms for multi-core and multi-GPU environments*. PhD Thesis from Université Lille 1, 2013.

[Climer 2006] S. Climer and W. Zhang. *Cut-and-Solve: an Iterative Search Strategy for Combinatorial Optimization Problems, Artificial Intelligence*. vol. 170, pages 714–738, 2006.

[Cung 1994] V.D. Cung, S. Dowaji, B. Le Cun, T. Mautor and C. Roucairol. *Parallel and Distributed Branch-and-Bound/A* Algorithms*. Rapport technique 94/31, Laboratoire PRISM, Université de Versailles, 1994.

[Dowaji 1995] S. Dowaji and C. Roucairol. *Load balancing strategy and priority of tasks in distributed environments*. In In IEEE Proc. of Intl. Phoenix Conf. on Computers and Communications, pages 15–22. IEEE, 1995.

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

[Garey 1979] Michael R Garey and David S Johnson. *Computers and intractability: a guide to the theory of NP-completeness. 1979*. San Francisco, LA: Freeman, 1979.

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

[Goldengorin 2004] B. Goldengorin, D. Ghosh and G. Sierksma. *Branch and Peg Algorithms for the Simple Plant Location Problem*. Computers & Operations Research, vol. 31, pages 241–255, 2004.

[Golomb 1966] S. Golomb. *Run-length encodings (Corresp.)*. Information Theory, IEEE Transactions on, vol. 12, no. 3, pages 399–401, Jul 1966.

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

[Janakiram 1988] V.K. Janakiram, D.P. Agrawal and R. Mehrotra. *A Randomized Parallel Branch-and-Bound Algorithm*. In in Proc. of Int. Conf. on Parallel Processing, pages 69–75, Aug. 1988.

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

[King 1980] J. R. King and A. S. Spachis. *Heuristics for flow-shop scheduling.* International Journal of Production Research, vol. 18, no. 3, page 345–357, 1980.

[Knuth 1998] Donald E Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addison-Wesley*. Reading, Massachusetts, 1998.

[Koopmans 1957] Tjalling C Koopmans and Martin Beckmann. *Assignment problems and the location of economic activities*. Econometrica: journal of the Econometric Society, pages 53–76, 1957.

[Kumar 1984] V. Kumar and L. Kanal. *Parallel Branch-and-Bound Formulations For And/Or Tree Search*. IEEE Trans. Pattern. Anal. and Machine Intell., vol. PAMI–6, pages 768–778, 1984.

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

[Lalami 2012] M.E. Lalami. *Contribution à la résolution de problèms d'optimisation combinatoire: méthodes séquentielles et parallèles.* Université de Toulouse III - Paul Sabatier., 2012.

[Lawler 1963] Eugene L. Lawler. *The Quadratic Assignment Problem*. Management Science, vol. 9, no. 4, pages pp. 586–599, 1963.

[Le Cun 1995] Bertrand Le Cun and Catherine Roucairol. *Concurrent data structures for tree search algorithms*. In Parallel Algorithms for Irregular Problems: State of the Art, pages 135–155. Springer, 1995.

[Lehmer 1960] D.H. Lehmer. *Teaching combinatorial tricks to a computer*. Proc. Sympos. Appl. Math. Combinatorial Analysis, vol. 10, pages 179–193, 1960.

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

[Loiola 2007] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter Hahn and Tania Querido. *A survey for the quadratic assignment problem*. European Journal of Operational Research, vol. 176, no. 2, pages 657 – 690, 2007.

[Luong 2011] T.V. Luong. *Métaheuristiques parallèles sur GPU.* Université des Sciences et Technologie de Lille, 2011.

[Mans 1996] B. Mans and C. Roucairol. *Performances of Parallel Branch and Bound Algorithms with Best-first Search*. Discrete Applied Mathematics, vol. 66, no. 1, pages 57–74, 1996.

[Mans 2006] B. Mans and C. Roucairol. *Concurrency in priority queues for branch and bound algorithms*. In Inria Research Report inria-00075248. HAL, 2006.

[Mautor 1994a] T. Mautor and C. Roucairol. *A New Exact Algorithm for the Solution of Quadratic Assignment Problems*. Discrete Applied Mathematics, vol. 55, no. 3, pages 281–293, 1994.

[Mautor 1994b] Thierry Mautor and Catherine Roucairol. *Difficulties of exact methods for solving the quadratic assignment problem*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 16, pages 263–274, 1994.

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

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

[Melab 2005] N. Melab. *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul*. LIFL, USTL, Novembre 2005. Thèse HDR.

[Melab 2012] N. Melab, I. Chakroun, M. Mezmaz and D. Tuyttens. *A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem*. In 2012 IEEE Intl. Conf. on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012, pages 10–17, 2012.

[Melab 2014] 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, vol. 26, no. 16, pages 2667–2683, 2014.

[Mezmaz 2007a] M. Mezmaz, N. Melab and E-G. Talbi. *An efficient load balancing strategy for grid-based branch and bound algorithm*. Parallel Computing, vol. 33, no. 4-5, pages 302–313, 2007.

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

[Miller 1993] D.L. Miller and J.F. Pekny. *The Role of Performance Metrics for Parallel Mathematical Programming Algorithms*. ORSA J. Computing, vol. 5, no. 1, pages 26–28, 1993.

[Pastor 2004] R. Pastor and A. Corominas. *Branch and Win: OR Tree Search Algorithms for Solving Combinatorial Optimisation Problems*. Top, vol. 1, pages 169–192, 2004.

[Ponsich 2010] Antonin Ponsich and CarlosA. Coello Coello. *Testing the Permutation Space Based Geometric Differential Evolution on the Job-Shop Scheduling Problem*. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej and Günter Rudolph, editeurs, Parallel Problem Solving from Nature, PPSN XI, volume 6239 of *Lecture Notes in Computer Science*, pages 250–259. Springer Berlin Heidelberg, 2010.

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

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

[Sahni 1976] Sartaj Sahni and Teofilo Gonzalez. *P-Complete Approximation Problems*. J. ACM, vol. 23, no. 3, pages 555–565, July 1976.

[Shavit 2011] Nir Shavit. *Data Structures in the Multicore Age*. Commun. ACM, vol. 54, no. 3, pages 76–84, March 2011.

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

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

[Vu 2014] Trong-Tuan Vu. *Heterogeneity and locality-aware work stealing for large scale Branch-and-Bound irregular algorithms*. PhD Thesis from Université Lille 1, 2014.

International Publications

Conference Papers

- M. Mezmaz, R. Leroy, N. Melab and D. Tuyttens. **A Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System.** In Proc. of IEEE IPDPS, pages 1203-1212, Phoenix, Arizona, USA, 2014.
- R. Leroy, M. Mezmaz, N. Melab and D. Tuyttens. **Work Stealing Strategies For Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System.** In Proc. of ACM SIGPLAN PPoPP/PMAM, Orlando, Florida, USA, 2014.
- N. Melab, R. Leroy, M. Mezmaz and D. Tuyttens. **Parallel Branch-and-Bound using private IVM-based work stealing on Xeon Phi MIC coprocessor.** In IEEE Proc. of HPCS, pages 394-399, Amsterdam, The Netherlands, 2015.
- R. Leroy, M. Mezmaz, N. Melab, and D. Tuyttens. **Using Factorial Number System to Solve Permutation Optimization Problems.** In 28th Annual Conference of the Belgian Operational Research Society, Mons, Belgium, January 2014

Journal paper under Minor Revision

- R. Leroy, J. Gmys, 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. Submitted in March 2015.

To be submitted

- N. Melab, J. Gmys, R. Leroy, M. Mezmaz and D. Tuyttens. **Many-core Branch-and-Bound for GPU accelerators and MIC coprocessors.**

APPENDIX A

Other experimental results

	IVM	LL		
Instance	**Data Structure Size (Bytes)**	**Maximum Pool Size**	**Data Structure Size (Bytes)**	**Data Structure Ratio (LL/IVM)**
Ta041		741	37050	26.00
Ta042		774	38700	27.16
Ta043	1426	714	35700	25.05
Ta045		483	24150	16.95
Ta050		652	32600	22.88
Average	1425	673	33640	23.61

Table A.1: Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of memory.

	IVM	LL		
Instance	**Data Structure Size (Bytes)**	**Maximum Pool Size**	**Data Structure Size (Bytes)**	**Data Structure Ratio (LL/IVM)**
Ta113	126751	14018	7009000	55.30
Ta115		8326	4163000	32.84
Average	126751	11172	5586000	44.07

Table A.2: Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of memory.

	IVM	LL	
Instance	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Ratio (LL/IVM)**
Ta021	129.59	439.90	3.39
Ta022	67.72	229.59	3.39
Ta023	448.11	1508.90	3.37
Ta024	119.37	398.92	3.34
Ta025	124.70	429.60	3.45
Ta026	210.19	693.52	3.30
Ta027	176.62	604.74	3.42
Ta028	25.23	88.12	3.49
Ta029	21.04	71.88	3.42
Ta030	5.15	17.45	3.39
Average	132.77	448.26	3.38

Table A.3: Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of CPU Time.

	IVM	LL	
Instance	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Ratio (LL/IVM)**
Ta041	0.16	0.71	4.47
Ta042	79.42	351.96	4.43
Ta043	5.59	24.36	4.35
Ta045	0.06	0.30	4.85
Ta050	13.30	58.83	4.42
Average	19.71	87.23	4.51

Table A.4: Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of CPU Time.

	IVM	LL	
Instance	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Ratio (LL/IVM)**
Ta071	378.65	1457.96	3.85
Ta073	1053.24	2956.72	2.81
Average	715.94	2207.34	3.33

Table A.5: Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to infinity in terms of CPU Time.

	IVM	LL	
Instance	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Time (Sec.)**	**Pool Mgmt Ratio (LL/IVM)**
Ta113	31.12	308.72	9.92
Ta115	0.96	7.79	8.13
Average	16.04	158.26	9.02

Table A.6: Comparison of serial IVM-B&B and LL-B&B when the solution is initialized to optimum in terms of CPU Time.

APPENDIX B

Inversion table

Permutation to factorial number using inversion table

It is also possible to encode a permutation as a factorial number using the inversion table method. Table B.1 explains with an example the used procedure for obtaining the code $T = (T_0 T_1 ... T_8)_!$ from a permutation $\pi = 527038614$. In the figures of this table, position, code and element are represented with the same colors than Table 2.3.

The procedure for obtaining the code T_0 is illustrated using the figure of the first cell the table. This code T_0 is equal to the number of inversions where 0 (0, not T_0) appears at the right side of a pair. Since the number of inversions of this type is equal to 3, then T_0 is equal to 3. In the same way, T_1 is equal to 6 since there are 6 inversions where 1 appears at the right side of pair. Therefore, T_i is equal to the number of inversions where i appears at right side of a pair (see Equation (B.1)).

$$T_i = |Inversions(\pi, i)| \tag{B.1}$$

The resulting number is a factorial number which has a size n. Indeed, T_i is always smaller than $8 - i$ since the number of inversions where i can appear at the right side of a pair is always smaller than $8 - i$. For example, T_0 is always smaller than $8 - 0$ (i.e. 8) since the number of inversions where 0 can appears at the right side of a pair can not exceed $8 - 0$. This the case of a permutation where π_8 is equal to 0. T_8 is always less than $8 - 8$ (i.e. 0) since the number of inversions where 8 appears at the right side of a pair is always equal to 0 (i.e. $8 - 8$). Indeed, no digit can be higher than 8 in a permutation of 9 elements. So there is no pair where 8 appears on the right side.

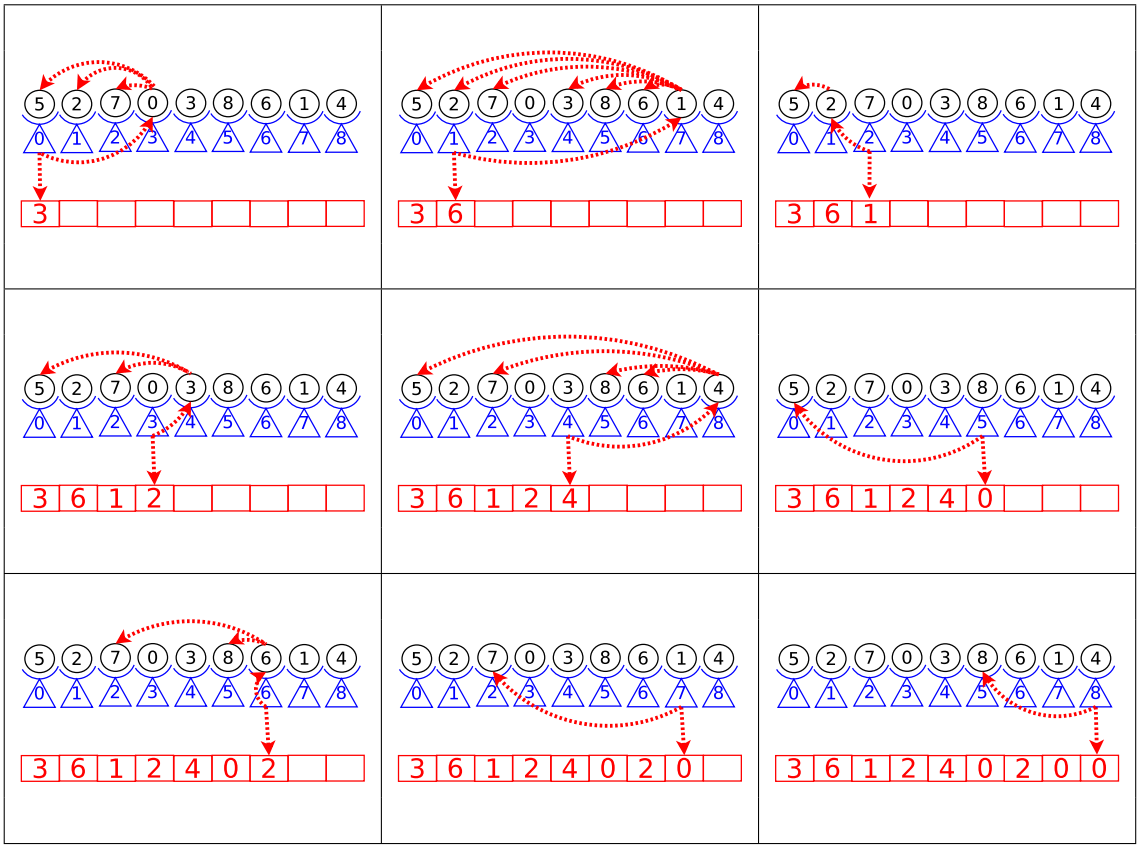


Table B.1: Coding of a permutation using the inversion table code.

Algorithm 15 allows to find the code for an inversion table. The algorithm receives as input a permutation π, and returns as output a code T of this permutation obtained with the inversion table. Like Algorithm 15, the role of the first step is to initialize to 0 all the components T_i of the vector T. This algorithm is described with two loops which test all possible inversions. For each found inversion (π_i, π_j), the value of T_k is incremented where k is equal to π_j. The algorithm has a complexity of $O(n\ log(n))$.

Algorithm 15 PERMUTATION-TO-TABLE-INVERSION(π)

```
Vector-initialize-zero(T);
for i ← 0 to (N-1) do
    for j ← (i+1) to (N-1) do
        if π_i > π_j then
            k ← π_j
            T_k ← T_k + 1;
        end if
    end for
end for
return T
```

Factorial number to permutation using inversion table

From a code $T = (361240200)_!$ obtained with the inversion table, the figures of Table B.2 explain, using an example, the method used to decode the permutation π of this code T. Unlike a Lehmer code that assigns elements at positions, the code of an inversion table allows to rather assign positions to elements. In other words, decoding T consists, not to distribute elements on positions, but rather to distribute positions on elements. At the beginning, all the positions, represented by blue triangles, are not assigned, and all the elements, represented by black circles, are also free.

From the first cell of the table, decoding T_0 gives the position associated to the element 0. As T_0 is equal to 3, then the $(3+1)^{th}$ free position must be associated to the element 0. As all the positions are not associated at the beginning, then the $(3+1)^{th}$ unaffected position is the position 3. Therefore, the position 3 is associated to the element 0. From the second cell of the table, decoding T_1 gives the position associated to the element 1. As T_1 is equal to 6, then the $(6+1)^{th}$ free position must be associated to the element 1. Since the position 3 is already occupied, the $(6+1)^{th}$ unaffected position is the position 7. Therefore, the position 7 is associated to the element 1.

Decoding T_i gives always the position associated to the element i which is the $(T_i+1)^{th}$ free position. At the end of decoding operation of $T = (361240200)_!$, the obtained permutation is π such as π = {(3,0), (7,1), (1,2), (4,3), (8,4), (0,5), (6,6), (2,7), (5,8)}. Using the representation of Equation (2.18), π is then equal to 527038614. The obtained permutation $\pi = 527038614$, using the inversion table code $T = (361240200)_!$, is therefore the same than the obtained permutation with the Lehmer code $L = (525013200)_!$, even if L and T are different factorial numbers.

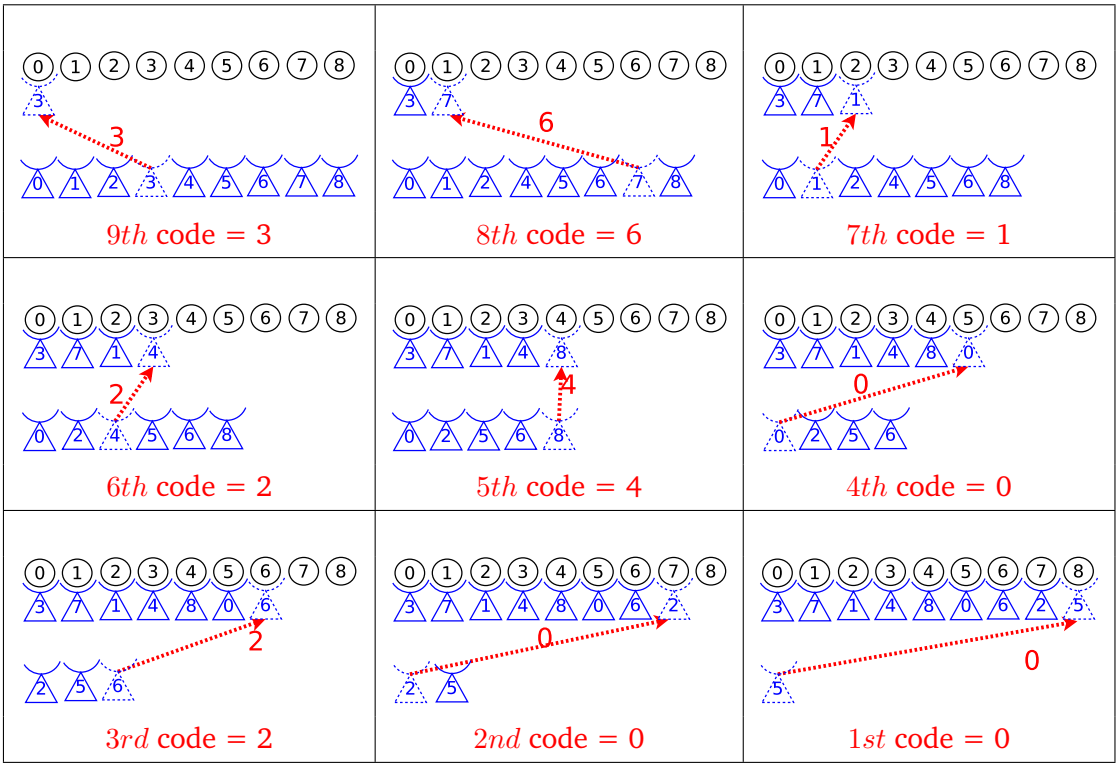


Table B.2: Decoding a permutation represented by a code obtained with the inversion table.

Algorithm 16 summarizes the decoding method of a code obtained with the inversion table. The algorithm receives as input a factorial number corresponding to a code obtained with the inversion table, and returns as output a permutation which corresponds to this code. The algorithm begins by initializing a position vector called $Places$ containing all the sorted positions. This initialization is done using the function VECTOR-INITIALIZE explained in Subsection 2.4.1. At each iteration $ITEM$, the algorithm computes the position $PLACE$ of element $ITEM$. Once $ITEM$ and $PLACE$ are known at the end of an iteration, the algorithm assigns $ITEM$ to the element π_{PLACE}.

To compute the position $PLACE$ associated to an iteration, the algorithm reads the place located at the position T_{ITEM} of the vector of the positions $Places$. The reading operation is done using the function VECTOR-SELECT. After reading the position $PLACE$, the function VECTOR-SELECT also shifts with one position to the left all the positions located at the right of $PLACE$. This algorithm has therefore a complexity equal to $O(n\ log(n))$.

Algorithm 16 TABLE-INVERSION-TO-PERMUTATION(T)

```
Vector-initialize(Places)
for ITEM ← 0 to (N-1) do
    i ← T_ITEM
    PLACE ← Vector-select(Places,i)
    π_PLACE ← ITEM
end for
return π
```