Grascomp
École Doctorale Sciences Pour l'Ingénieur, Université de Lille Nord-de-France

UMONS
Université de Mons

Université de Lille 1 SCIENCES ET TECHNOLOGIES

THÈSE

préparée dans le cadre d'une cotutelle
pour obtenir le grade de

Docteur en Sciences de l'Ingénieur de l'Université de Mons
Docteur en Informatique et applications de l'Université de Lille

présentée et soutenue publiquement par

Jan Gmys

le 19 décembre 2017

Heterogeneous cluster computing for many-task exact optimization - Application to permutation problems

devant le jury composé de

	Patricia Stolf,	Maître de conférence	Université Toulouse Jean-Jaurès
	Imen Chakroun,	Researcher	IMEC
	Pierre Manneback,	Professeur	Université de Mons
	Frédéric Semet,	Professeur	École Centrale de Lille
	Andrzej Jaszkiewicz,	Professeur	Poznan University of Technology
Rapporteurs	Franck Cappello,	Senior Researcher	Argonne National Laboratory (USA)
	Farouk Yalaoui,	Professeur	Université de Technologies Troyes
Directeurs	Daniel Tuyttens,	Professeur	Université de Mons
	Nouredine Melab,	Professeur	Université de Lille
Co-directeur	Mohand Mezmaz,	Chargé de Recherche	Université de Mons

Abstract:

Branch-and-Bound (B&B) is a tree-based exploration method for exactly solving combinatorial optimization problems by implicit enumeration of the solution space. B&B dynamically generates large irregular search trees which need to be processed in parallel as only very small problem instances can be solved within a reasonable amount of time on a sequential computer. The latest *Top500* ranking of the world's largest supercomputers confirms the tendency towards heterogeneous systems including GPUs and multi-core/many-core processors as main building blocks. This thesis revisits the design and implementation of B&B for hybrid multi-core and multi-GPU platforms, from single-node systems to large-scale heterogeneous high performance computing clusters. Although B&B is a generic method, the design of parallel B&B is strongly influenced by both the characteristics of the tackled problem and the target architecture. The focus is put on permutation-based combinatorial problems, using the Flowshop Scheduling Problem (FSP), the Quadratic Assignment Problem (QAP) and the n-Queens puzzle problem as test-cases.

An innovative data structure dedicated to permutation problems, called Integer-Vector-Matrix (IVM) is used for the efficient storage and management of the pool of subproblems instead of conventional data structures (e. g. stacks, deques, priority queues). The principle of parallel IVM-based B&B is to have several independent B&B processes use their private IVM for the exploration of different parts of the search space, which are compactly encoded as intervals. Several IVM-based parallel B&B algorithms are presented in this thesis. For shared-memory multi-core processors, we propose a hybrid multi-core-GPU B&B that can use available GPU devices for the acceleration of the bounding operator, which is often the most time-intensive part of B&B. Targeting GPUs and single-node multi-GPU systems, we present the first B&B algorithm that runs entirely on the GPU, using different parallelization models for coarse- and fine-grained permutation problems. In order to exploit heterogeneous clusters with multiple distributed GPUs and multi-core CPUs, both approaches are combined using the master-worker paradigm.

In order to balance the workload IVM-based B&B processes exchange intervals in a work stealing approach. Indeed, considering the highly irregular nature of B&B, one of the main challenges is the dynamic distribution of subproblems (nodes of the B&B tree) among a large set of processors. The design of efficient work stealing strategies is therefore a central contribution of this thesis. Interval-based work stealing strategies are proposed for multi-core B&B, including its GPU-accelerated variant and for GPU-centric B&B, performing work stealing inside the GPU. Using hierarchical work stealing approaches, inter-GPU and inter-node load balancing approaches are proposed for

single-node multi-GPU systems and hybrid distributed clusters.

A second major challenge raised by the irregularity of B&B is the efficient use of single-instruction multiple-data processing at the instruction-level, as well as the minimization of detrimental effects from irregular memory access patterns. Targeting Intel multi-core and many-integrated core (MIC) processors, a vectorizable implementation of the FSP bounding operator is proposed. In order to improve control flow efficiency and reduce instruction replay overhead, alternative mapping schemes are presented for the GPU-centric B&B.

Extensive experimental evaluations are performed using the three test-cases, revealing input-dependent behaviors. Results demonstrate the scalability of the approach at different levels, with respect to the number of CPU cores, GPU cores, GPU devices and heterogeneous compute nodes. In particular the use of multi-GPU systems and large clusters allows to solve instances whose exact resolution is otherwise impractical. For a class of 20 jobs-on-20 machines FSP instances with sequential execution times between 15 minutes and 22 hours, the resolution time using four GPUs ranges from 1 second to 1 minute, i. e. an improvement of three orders of magnitude compared to a single CPU core. A large FSP instance, defined by 50 jobs and 20 machines, whose resolution requires 25 days of processing on more than 300 CPU cores is solved within 9 hours on a cluster containing 36 GPUs.

Keywords:

Branch-and-Bound, Combinatorial optimization, High-performance computing, GPU computing, Permutation problems

Résumé:

L'algorithme Branch-and-Bound (B&B, en Français, séparation et évaluation) est fréquemment utilisé pour la résolution exacte de problèmes d'optimisation combinatoire. Il s'agit d'une méthode de recherche arborescente qui procède par énumération implictite de l'ensemble de solutions. L'algorithme génère des arbres de recherche souvent très larges et fortement irréguliers qui doivent être explorés en parallèle, car seules des petites instances peuvent être résolues en un temps raisonnable sur une machine séquentielle. Le dernier classement mondial des supercalculateurs les plus puissants (*Top500*) confirme que ces derniers tendent à être de plus en plus hétérogènes, combinant processeurs multi-core, many-core et processeurs graphiques (GPUs). Dans cette thèse nous réexaminons la conception et l'implémentation d'algorithmes B&B sur de larges plateformes hétérogènes de calcul haute performance. En dépit de sa généricité, la conception d'algorithmes B&B parallèles est fortement influencée par les caractéristiques du problème résolu et de l'architecture ciblée. Nous nous concentrons sur des problèmes combinatoires définis sur l'ensemble des permutations. Le problème d'ordonnancement Flow-Shop (FSP), le problème d'affectation quadratique (QAP) et le problème des n-dames sont utilisés comme cas d'étude.

Une structure de données originale (appelée IVM ou Integer-Vector-Matrix) dédiée aux problèmes de permutation est utilisée pour le stockage et la gestion efficaces du pool de sous-problèmes (noeuds de l'arbre). Dans le B&B parallèle basé sur IVM, plusieurs processus B&B indépendants explorent en parallèle différentes parties de l'espace de recherche en utilisant leur structure IVM privée. Chaque partie de l'espace de recherche assigné à un processus B&B est encodée de manière compacte sous forme d'un intervalle de nombres factoriels.

Plusieurs algorithmes B&B basés sur IVM sont présentés dans cette thèse. Pour des processeurs multi-coeurs à mémoire partagée, nous proposons un algorithme hybride qui utilise les GPUs disponibles pour l'accélération de l'évaluation de sous-problèmes, celle-ci étant souvent la partie la plus coûteuse en temps. Ciblant les GPUs et les systèmes multi-GPUs, nous proposons le premier algorithme B&B executé entièrement sur l'accélérateur, utilisant deux modèles de parallélisation adaptés aux problèmes à granularité fine et grossière. Pour l'exploitation de clusters hétérogènes à mémoire distribuée, les approches multi-core et GPU sont combinées en les intégrant dans un modèle fermier-travailleur.

Afin d'équilibrer la charge de travail dynamiquement, les processus B&B basés sur IVM s'échangent des intervalles en utilisant une approche de vol de tâches. A cause de la nature fortement irrégulière et imprévisible de l'algorithme B&B, la répartition dynamique de sous-problèmes parmi l'ensemble des processeurs représente en effet un

des défis majeurs. Par conséquent, la conception de stratégies de vol de tâches occupe une place centrale dans cette thèse. Des strategies de vol de tâches basées sur les intervalles de nombres factoriels sont proposées pour le B&B multi-core (accéléré par GPU) et pour le B&B basé sur GPU, équilibrant la charge à l'intérieur du GPU. En utilisant une approche hiérarchique du vol de tâches, des mécanismes d'équilibrage de charge inter-GPUs et inter-noeuds sont proposés pour des systèmes multi-GPUs et pour des clusters hybrides à mémoire distribuée.

L'utilisation efficace du traitement "instruction unique, données multiples" (SIMD) est un second défi posé par l'irrégularité inhérente à l'algorithme B&B. En effet, la divergence de contrôle et les accès mémoire irréguliers dégradent fortement la performance du traitement SIMD, qui est un levier de performance important au niveau des instructions. Ciblant l'exploitation des registres SIMD des processeurs multi-core et MIC, une nouvelle implémentation vectorisable de la borne inférieure utilisée pour le FSP est présentée. Pour reduire les effets négatifs de la divergence de contrôle sur l'efficacité du traitement par GPU, des mappings alternatifs pour l'implémentation sur GPU des différents opérateurs B&B sont présentés.

Des évaluations expérimentales approfondies utilisant les trois cas d'étude ont été effectuées, révélant l'impact du problème résolu sur le comportement des algorithmes. Les résultats expérimentaux démontrent une bonne scalabilité des approches proposées à plusieurs niveaux: par rapport au nombre de coeurs CPU et GPU, par rapport au nombre de GPUs utilisés et par rapport au nombre de noeuds hétérogènes dans le B&B distribué.

En particulier, l'utilisation de systèmes multi-GPUs et de larges clusters de GPUs permet une résolution d'instances de problèmes qui serait impraticable séquentiellement. Pour une classe d'instances du FSP (20×20) avec un temps de résolution entre 15 minutes et 22 heures sur un seul coeur CPU, le temps de résolution sur 4 GPUs est compris entre 1 seconde et 1 minute, soit trois ordres de grandeur plus court. Une très grande instance du FSP définie par 50 jobs et 20 machines, dont le temps de résolution séquentiel est estimé à 22 ans, est résolue en 9 heures sur un cluster de calcul équipé de 36 GPUs.

Mots clés

Branch-and-Bound parallèle, Optimisation combinatoire, Caclul haute performance, Processeurs Graphiques, Problèmes de Permutation

Contents

Contents 1

Introduction 1

1 Parallel Branch-and-Bound algorithms 8
- 1.1 Introduction . . . 9
- 1.2 Solving permutation combinatorial optimization problems . . . 10
- 1.3 Branch-and-Bound algorithms . . . 11
  - 1.3.1 Terminology and general description . . . 12
  - 1.3.2 Models for parallel Branch-and-Bound . . . 14
  - 1.3.3 Challenges in parallel Branch-and-Bound . . . 17
- 1.4 Computing Environments . . . 20
- 1.5 Related work . . . 25
  - 1.5.1 B&B for multi-core CPUs . . . 25
  - 1.5.2 B&B for Graphics Processing Units . . . 27
  - 1.5.3 Hybrid and distributed parallel B&B . . . 29
- 1.6 Test-cases: Permutation-based COPs . . . 30
  - 1.6.1 Flowshop Scheduling Problem (FSP) . . . 30
  - 1.6.2 Quadratic Assignment Problem (QAP) . . . 32
  - 1.6.3 n-Queens Problem . . . 33
  - 1.6.4 B&B tree analysis of the test problems . . . 34

2 IVM-based B&B for multi-/many-core systems 39
- 2.1 Introduction . . . 41
- 2.2 IVM-based parallel Branch-and-Bound . . . 41
  - 2.2.1 IVM-based serial B&B . . . 41
  - 2.2.2 Position vector: factoradic numbers . . . 44
  - 2.2.3 Work units: intervals of factoradics . . . 46

2.2.4 Work unit communication . . . 48
2.3 Work stealing for IVM-based B&B on multi-core CPUs . . . 50
2.3.1 Work stealing using factoradic intervals . . . 51
2.3.2 Victim selection policies . . . 51
2.3.3 Granularity policies . . . 54
2.4 Accleration of bounding operator . . . 55
2.4.1 GPU acceleration . . . 55
2.4.2 Vectorization of the FSP bounding procedure . . . 57
2.5 Experiments . . . 59
2.5.1 Evaluation of data structures for B&B . . . 60
2.5.2 GPU-acceleration of the bounding operator . . . 62
2.5.3 Evaluation of Work Stealing Strategies . . . 63
2.5.4 Performance evaluation on Intel Xeon Phi . . . 67
2.5.5 MC-B&B: performance on different multi-core CPUs . . . 70
2.6 Conclusions . . . 71

3 GPU-centric Branch-and-Bound 74
3.1 Introduction . . . 76
3.2 Discussion of design choices . . . 77
3.3 GPU-B&B and GPU-backtracking . . . 81
3.3.1 GPU-B&B: 2-level parallelization . . . 81
3.3.2 Thread-data mapping and branch divergence reduction . . . 86
3.3.3 GPU-BT: 1-level parallelization . . . 90
3.4 Work stealing strategies for GPU-B&B . . . 92
3.4.1 Victim Selection policies . . . 92
3.4.2 Work stealing for multi-GPU-B&B . . . 96
3.5 Experiments . . . 97
3.5.1 Evaluation of Mapping approaches . . . 97
3.5.2 Evaluation of Work Stealing strategies . . . 101
3.5.3 Scalability analysis . . . 103
3.5.4 Multi-GPU-B&B performance evaluation . . . 107
3.5.5 Hybrid CPU-multi-GPU-B&B . . . 111
3.6 Conclusions . . . 114

4 Branch-and-Bound for hybrid HPC clusters 117
4.1 Introduction . . . 118
4.2 B&B for hybrid clusters . . . 118

4.2.1 B&B@Grid . . . 118
4.2.2 Design of hybrid distributed B&B . . . 120
4.2.3 Redundant exploration . . . 122
4.2.4 Implementation of worker process . . . 124
4.3 Experiments . . . 127
4.3.1 Experimental protocol . . . 127
4.3.2 Resolution of very large problem instances . . . 127
4.3.3 Scalability: Ouessant . . . 132
4.3.4 Hybrid CPU/GPU scalability . . . 135
4.3.5 Solving other 50×20 FSP instances . . . 137
4.4 Conclusion . . . 138

5 Conclusions and Perspectives **140**

List of Figures **158**

List of Tables **160**

A Appendix **I**
A.1 Tree sizes . . . I
A.2 Lower bounds: complexities . . . I
A.3 Hardware . . . II

Introduction

Permutation-based optimization or constraint satisfaction problems frequently arise in industrial and economic applications such as routing, scheduling and assignment. Solving such problems consists in finding one or several permutation(s) that minimize/maximize[1] a given cost function or, respectively, satisfy a given set of constraints.

In this thesis we address NP-hard combinatorial optimization problems. Heuristic approaches are able to find near-optimal solutions in a reasonable amount of time, but they fail in general to find optimal solutions. Conversely, exact methods allow to find the optimal solution(s) with a proof of optimality, but the required computation power can be very huge. One of the most used exact methods to solve COPs is the Branch-and-Bound (B&B) algorithm. B&B implicitly enumerates all possible solutions by dynamically constructing and exploring a tree. This is done using four operators: branching, bounding, selection and pruning. Using the branching operator, the algorithm recursively decomposes the problem into smaller subproblems. A bounding function is used to compute lower bounds on the optimal cost of these subproblems. Using these lower bounds, the pruning operator discards subproblems from the search that cannot lead to an improvement of the best solution found so far. The tree-traversal is guided by the selection operator which returns the next subproblem to be processed according to a predefined strategy (e.g. depth-first search). Due to the pruning of branches the explored tree is highly irregular and unpredictable in size and shape.

Although the lower bound-based pruning mechanism of B&B significantly reduces the number of explored nodes, B&B algorithms often generate very large trees. Using a single processing core only small or moderately-sized instances can be solved in a reasonable amount of time. For this reason, over the last decades, parallel computing has been revealed as an attractive way to deal with larger instances.

Over the past 20 years we have witnessed an impressive evolution of computing technologies. Computer architects, programmers and researchers are moving towards

1. Without loss of generality the minimization case is considered in this thesis.

heterogeneous (or hybrid/collaborative) computing which aims at matching the requirements of each application to the strengths of the different architectures present in a heterogeneous computing system [MV15]. The biannual list of the most powerful supercomputers in the world, Top500 [16], reveals that advances in computer system design, microprocessor architecture, memory subsystem and networking allow to deliver factor 2 performance increase every $18-24$ months. While a similar growth is predicted over the next 10 years, this is being achieved through a huge increase in hardware complexity [GR14]. The cost of moving data, in terms of energy and time, is often greater than the cost of computations, although memory subsystems have evolved to an impressive level of complexity [Dre07]. As energy-efficiency has become a key issue, the trend towards more complex and more heterogeneous HPC systems is likely to continue. The rise of low-power processors and system-on-chip (SoC) designs, driven by the mobile-device market, indicate that future HPC systems are likely to contain a large number of heterogeneous, more or less powerful components [GR14]. An increasing number of heterogeneous components (and lower operating voltages) may lead to higher failure rates, calling for fault-tolerant algorithms and highly efficient communication schemes, to cite just two of the tremendous challenges that lie ahead in the field of HPC [Cap09; CGG+14].

The B&B algorithm can be used to solve basically any type of COP. Despite the genericity of B&B, the nature of the problem being solved has a strong impact on fundamental design and implementation decisions. In particular, the choice of the most suitable parallelization model and of the most suitable target architecture is determined mainly by the computational characteristics of the node evaluation function. Knowing that billions of nodes need to be evaluated, if possible in parallel, the following questions are crucial. Is the operation of evaluating a node very time and/or memory-consuming? Memory or compute-bound? Parallelizable, vectorizable? The research advances that have been made over the last decade, exploring the use of GPUs for B&B algorithms illustrate how the problem-dependent answers to these questions impact design choices.

One parallelization approach consists in generating large pools of nodes on the host CPU and offloading them to the GPU, where the nodes are evaluated in parallel. This approach has been applied for instance to the Flowshop Scheduling Problem (FSP) [CMMB13; VDM13] and Knapsack problems [LE12]. To reduce the data volume transferred between host and device, the branching and pruning operators are also implemented on the device [Cha13]. Using multiple streams in order to overlap data transfers and careful tuning of the size of offloaded pools [Cha13] help further decrease the overhead incurred by CPU-GPU communications.

For problems where the node-evaluation cost is relatively low this approach becomes less efficient, as data transfers and the sequential selection and insertion of nodes become bottlenecks. For such fine-grained applications an approach that focuses on performing the search itself in parallel on the GPU is preferred. It consists in exploring the B&B tree on the CPU until a predefined cutoff depth, storing all frontier nodes in a data structure. After this initial search, this set is sent to the GPU and each node in this set is used as a root for a concurrent exploration of the associated subtree. For instance, this approach has been successfully applied to the n-Queens [LLW+15a], the Traveling Salesman [CMNL11] and Multiproduct Batch Plant problems [BHG15]. However, such approaches fail to explore highly irregular trees efficiently, as they rely on a static workload repartition.

This thesis deals with the mapping of the B&B algorithm onto heterogeneous computing systems. The focus is put on solving large instances of three well-known permutation-problems, the Flowshop Scheduling Problem (FSP), the Quadratic Assignment Problem (QAP) and the n-Queens Problem. These three use cases present different characteristic features and we aim at matching the algorithm with the heterogeneous target architecture in a transparent way for any of the three test-problems. The computing environments considered are shared-memory multi-core CPUs, Many-Integrated Core (MIC) processors, Graphics Processing Units (GPU) and hybrid clusters integrating these three processing units.

As for other algorithms, data structures play a major role in the performance of a B&B algorithm [MR90]. In [Ler15], a data structure using an Integer, a Vector and a Matrix (called IVM) was introduced as an alternative for linked-list-based data structures which are conventionally used in B&B algorithms for permutation problems. In [Ler15], Leroy compares IVM with linked-list-based data structures from a theoretical point of view and experimental results reported in [MLMT14] show that IVM, compared to linked-lists (LL), allows to reduce the amount of time spent in managing the pool of subproblems as well as memory requirements for storing this pool. The B&B algorithms for multi-core, many integrated core (MIC), GPU and hybrid clusters integrating the former three, which are developed in this thesis are based on the IVM data structure.

The addressed issues and proposed contributions are summarized in the following:

- In [Ler15; MLMT14] IVM has only been applied to the Flowshop Scheduling Problem (FSP) and the experimental evaluation is limited to the cost of managing the pool of subproblems. Therefore, **our first contribution consists in validating the IVM data structure by revisiting the approach considering other permutation**

problems, namely the Quadratic Assignment Problem (QAP) and the n-Queens Problem. In order to establish the utility of IVM for permutation problems in general, it is important to investigate with problems that present different characteristic features, in particular the shape of the explored tree and the computational complexity of the node evaluation function. Indeed, the performance of B&B is strongly influenced by these two problem-dependent parameters.

- We present an IVM-based multi-core algorithm (MC-B&B) for these three problems, as well as work stealing strategies using intervals of factoradics as work units exchanged between threads. Based on MC-B&B, **we propose a hybrid GPU-accelerated multi-core B&B algorithm** (GMC-B&B). In addition to the parallel exploration of the B&B tree, each thread in GMC-B&B concurrently offloads subproblems to the GPU, where the corresponding lower bound values are computed in parallel. Work stealing strategies for dynamic load balancing are revisited for the accelerated algorithm. For comparison, equivalent linked-list based versions of both algorithms are implemented and experimentally compared to the IVM-based approach.

- Another challenging issue related to B&B for multi-core CPUs is vectorization. In addition to being composed of multiple cores, almost all modern CPUs provide vector instruction sets (e. g. AVX, ARM NEON, AVX2, AVX-512) operating on 128, 256 resp. 512 bit registers (allowing 4, 8 resp. 16 32-bit operations to be performed in one cycle). In order to make full use of the processing capabilities of multi-core CPUs, the computationally intensive parts of an algorithm should be vectorized. In some cases this can be automatically achieved through compiler support, while other cases require rethinking the algorithmic structure of the code. **For the FSP, the CPU implementation of the lower bounding procedure is revisited and a vectorization mechanism is proposed.**

- **We propose a (multi-)GPU-based B&B algorithm (GPU-B&B) that implements the entire algorithm as well as load balancing mechanisms on the GPU.** To the best of our knowledge it is the first B&B algorithm to perform the entire search process on the GPU. At the core of GPU-B&B is the IVM data structure, whose small and constant memory footprint is better suited to GPUs than conventional LL-based data structures. Using IVM, thousands of tree explorations are performed in parallel and the evaluation of nodes is in turn parallelized to add a second level of fine-grained parallelism. The implementation of B&B on GPU is challenging because its irregular nature contrasts with the regularity of the GPU architecture

and execution model. Indeed, irregular instruction flow and unstructured memory access patterns are highly detrimental to the performance of GPU processing, as they cause bandwidth penalties and instruction serialization. As a well-informed choice of the thread-data mapping can minimize these effects, different mapping schemes are proposed for the four B&B operators.

- The choice of the best parallelization model depends on the problem being solved. When the application is computationally dominated by a relatively costly bounding procedure (e. g. FSP, QAP) the two-level parallelization of GPU-B&B is well-suited, because the overhead incurred by handling the nested level is compensated by performance gains. However, for fine-grained problems with inexpensive node evaluation functions, like in heuristic backtracking algorithms [RK93], parallel node evaluation is inefficient. **For the efficient resolution of fine-grained permutation problems on GPUs we propose a GPU-based backtracking algorithm (GPU-BT)**. In contrast to the 2-level GPU-B&B, workers perform multiple iterations of B&B within the same kernel until the number of idle worker reaches a critical threshold and a load balancing phase is triggered.

- Dynamic load balancing is a crucial component of parallel tree search algorithms. As GPU-B&B completely bypasses the CPU and performs massively parallel searches on irregular trees, it is necessary to design and implement an efficient mechanism for load balancing inside the GPU. Adapting CPU-based approaches like the work stealing strategies used in MC-B&B to the GPU is not straightforward, as these approaches require locking and mutual exclusion mechanisms, unavailable or inefficient on GPUs. Therefore, **the work stealing mechanism for IVM-based B&B is rethought and adapted to the single-instruction multiple-data (SIMD) execution model of the GPU and multi-GPU systems**. We propose five work stealing strategies to tackle the problem of work load imbalance inside the GPU. These strategies use different topologies (e. g. ring and hypercube), victim selection policies and mechanisms to adapt themselves to the different phases of the parallel exploration process (ramp-up and shutdown). The proposed load balancing approach is used by both variants of the algorithm.

- In order to solve very large permutation COPs, like the 50-job FSP instance $Ta056$ [Tai93] which requires 22 CPU-years to be solved to optimality, all available resources of a hybrid GPU- and/or MIC-accelerated cluster must be used. For that purpose **we revisit the B&B@Grid approach [MMT07a] on heterogeneous clusters combining multi-core CPUs and GPUs.** The original B&B@Grid approach is a fault-tolerant

and highly scalable algorithm designed for computational grids. In this master-worker approach, workers in this algorithm are single-core processors. The design of the coordinator process which distributes work units across distributed compute nodes is revisited with the goal of including GPU devices, MIC and multi-core processors. We evaluate the proposed hybrid distributed B&B algorithm (HD-B&B) on a cluster with a total of 130 000 GPU cores located at the IDRIS[2] institute.

This thesis is organized in four chapters.

Chapter 1 introduces all the background and prerequisites necessary to the comprehension of the global document, namely the B&B algorithm, associated data structures and the FSP, QAP and n-Queens problems. The introduction of the algorithm and the test-cases is completed by a preliminary analysis of the sequential B&B algorithm. This introductory chapter also provides an overview of the different computing environments and the main features of the respective architectures. It contains a summary of the major parallelization strategies for B&B, a literature overview and synthesis of existing work dealing with parallel B&B classified by the target computational platform (GPU and MIC many-core processors, multi-core systems, computational grids and hybrid cluster systems).

Chapter 2 contains our contributions targeting multi-core systems, including the many-integrated core (MIC) architecture. First, it introduces the sequential IVM-based B&B algorithm, followed by the parallel IVM-based B&B for multi-core processors and associated work stealing strategies. Then, it describes the GPU-accelerated multi-core algorithm and the extension of the work stealing strategies to the hybrid MC-B&B. After presenting the approach that consists in accelerating the bounding operator on GPU, its acceleration through vectorization is addressed. The presence of obstacles for the automatic compiler-assisted vectorization of the most time-consuming parts is discussed. The proposed solutions include a vectorizeable re-implementation of the lower bound for the FSP. In the last section of this chapter, a detailed experimental study evaluates and compares the presented algorithms. We evaluate the performance of the (accelerated) multi-core algorithm, comparing its linked-list and IVM-based implementations for the three test problems. The efficiency of the vectorization approaches and the proposed work stealing strategies is also evaluated and discussed. Finally, a scalability analysis on different multi-core processors concludes this chapter.

2. *Institut du développement et des ressources en informatique scientifique*

Chapter 3 deals with the GPU-centric implementation of B&B. It starts by discussing and exposing fundamental design choices and challenges related to the constraints imposed by the SIMD execution model and the hierarchical memory organization of GPUs. The remainder of this chapter is organized in three parts: (1) a description of both variants of the GPU-centric B&B algorithm (2) a description of the GPU-based work stealing approach, including its extension to multi-GPU systems and (3) the experimental evaluation of the GPU-B&B algorithm. First, the implementation of the GPU-centric B&B algorithm using a two-level parallelization is presented, providing details on the implementation of each operator. A particular focus is put on the issue of branch divergence: the sources of branch divergence are identified and thread-to-data mappings that help solving this issue are presented. Then, the challenges faced by the two-level GPU-B&B when dealing with fine-grained permutation problems are identified and an alternative, single-level variant of the algorithm is presented. The following section addresses the crucial issue of load balancing inside the GPU. It contains a general description of the approach, followed by a presentation of five work stealing strategies for GPU-based work stealing and an hierarchical work stealing approach for inter-GPU load balancing in multi-GPU systems. In the final part of this chapter, the performance of the proposed GPU-B&B algorithm is evaluated. Experiments are performed with the three test-cases on a multi-GPU system composed of four GPUs.

Chapter 4 presents a hybrid distributed B&B algorithm based on the B&B@Grid approach. It starts by describing the original B&B@Grid approach, upon which the proposed HD-B&B algorithm is build. In order to enable the use of multi-core and many-core-based workers in B&B@Grid, a redefinition of work units is proposed and the implications of this modification are discussed. A detailed description of the master and worker processes is provided, focusing on the revisited communication scheme. Finally, HD-B&B is experimented on three different GPU-enhanced clusters with up to 36 GPUs.The experimental evaluation includes scalability and stability analysis and concludes by a perspective on solving very large, unsolved FSP instances.

Finally, in Chapter 5 some concluding remarks are drawn and some future extensions of the proposed approaches are presented.

Chapter 1

Parallel Branch-and-Bound algorithms

Contents

1.1 Introduction 9
1.2 Solving permutation combinatorial optimization problems 10
1.3 Branch-and-Bound algorithms 11
1.3.1 Terminology and general description 12
1.3.2 Models for parallel Branch-and-Bound 14
1.3.3 Challenges in parallel Branch-and-Bound 17
1.4 Computing Environments 20
1.5 Related work 25
1.5.1 B&B for multi-core CPUs 25
1.5.2 B&B for Graphics Processing Units 27
1.5.3 Hybrid and distributed parallel B&B 29
1.6 Test-cases: Permutation-based COPs 30
1.6.1 Flowshop Scheduling Problem (FSP) 30
1.6.2 Quadratic Assignment Problem (QAP) 32
1.6.3 n-Queens Problem 33
1.6.4 B&B tree analysis of the test problems 34

1.1 Introduction

Combinatorial optimization problems (COP) consist in finding an object within a finite (or countably infinite) set which is optimal according to a given criterion.

Formally, a COP can be defined as a couple (X, f), where X is the search space and $f : X \to R$ the objective function to be minimized[1]. Constraints that must be fulfilled by a feasible solution $x \in X$ can be incorporated in the definition of the search space X or the objective function f. The objective function f takes its values in a totally ordered set, usually the set of real numbers or integers. The value $f(x)$ measures the cost (e. g. quality, time, benefit) of solution $x \in X$. The goal is to find one or multiple solution(s) $x^\star \in X$ that is (are) feasible and for which $f(x^\star) \leq f(x), \forall x \in X$.

Many COPs can be modeled as optimization problems defined on sets of permutations. For instance, the solution of a permutation-based COP may represent:

- a scheduling of a set of jobs, such that the completion time of a manufacturing product is minimized (a scheduling problem),
- an assignment of facilities to locations such that the placement cost is minimized (an assignment problem),
- a planning of routes such that the total length of the routes is minimized (a traveling salesman problem).

In such problems candidate solutions are permutations of n integers $1, 2, \ldots, n$. The number n represents the number of jobs to schedule, the number of facilities to assign, etc. and is referred to as the *problem size*. As the set of permutations of size n is of cardinality $n!$, the search space associated with a permutation-based COP is usually very large, even for values of n that may seem moderate. While a permutation problem of size 5 could be easily solved by enumerating all 120 feasible solutions, this approach becomes clearly unfeasible for problems of size 50 or 100 which admit about 3×10^{64}, respectively 10^{158} candidate solutions.

This thesis focuses on permutation-based COPs. In particular, we address NP-hard hard permutation-based COPs. Three permutation-based problems are used as test-cases: the Permutation Flowshop Scheduling Problem (FSP), the Quadratic Assignment Problem (QAP) and the n-Queens problem. A detailed introduction of these problems is provided later, in Section 1.6 of this chapter. Formally, the n-Queens problem is not an optimization problem, because it consists in finding (all) feasible solutions and no notion

1. Without loss of generality we consider the minimization case: the maximization of f is equivalent to the minimization of $-f$.

of optimality is defined. However, one may model this constraint satisfaction problem as a COP by defining the objective function as a constraint-checking function which assigns 0 to feasible and 1 to infeasible solutions.

1.2 Solving permutation combinatorial optimization problems

Approaches to solving COPs can be classified into two main categories: exact and approximate methods [Tal09]. Approximate methods aim at finding near-optimal solutions in a reasonable amount of time, exploring parts of the solution space where good quality solutions are expected to be found. Among the most used approximate methods are metaheuristics. Roughly described, metaheuristics are general-purpose optimization methods that require limited problem-specific information. Metaheuristics are either single solution-based or population-based. While the former consider a single initial solution which is iteratively improved (e. g. Hill-Climbing, Simulated Annealing), the latter operate on a set of solutions which are collectively or independently improved (e. g. Evolutionary Algorithms, Ant Colonies). However, heuristics and metaheuristics provide no error quantification. Even if a heuristic method finds the/an optimal solution it does not provide a certificate of optimality. Knowledge of exact solutions to benchmark instances for COPs is therefore valuable for assessing the quality of a heuristic method for a class of problems. Also, some highly cost-critical applications may benefit from closing even small optimality gaps.

While high-quality solutions to COPs can often be found within a few seconds, exact solving may require a huge amount of time and computational resources. This is due to the enumerative nature of exact methods, which require, in the worst case, a number of iterations which grows exponentially with the problem size. The most naive exact method consists in completely enumerating the solution space. For obvious reasons this is only feasible for very small problem sizes. A more sophisticated and widely used exact method is the branch-and-bound (B&B) algorithm. B&B dynamically builds and explores a tree using four operators: branching, bounding, selection and elimination. The B&B approach recursively decomposes the problem into smaller subproblems for which lower bounds on their optimal solution are computed. Based on these lower bounds the elimination operator discards unpromising subproblems which have lower bounds greater than the best solution found so far. This is also known as *pruning* of tree branches or *fathoming* of parts of the search space. A detailed description of the B&B algorithm is provided in Section 1.3 of this chapter.

Many other exact resolution methods are B&B-like tree-search algorithms. Besides

simple B&B there are two main variants: branch-and-cut (B&C) and branch-and-price (B&P). There are other less known variants of B&B such as branch-and-peg [GGS04], branch-and-win [PC04], and branch-and-cut-and-solve [CZ06]. This list is certainly not exhaustive. It is also possible to consider a divide-and-conquer algorithm as a B&B algorithm, as it is enough to remove the pruning operator from B&B. Some authors consider B&C, B&P, and the other variants as different algorithms than B&B and use B&X to refer to algorithms like B&B, B&C, B&P, etc. In this document B&B refers to simple B&B or any of its variants. Backtracking is a fundamental paradigm frequently used to solve constraint satisfaction problems and can also be interpreted as a special case of a depth-first search (DFS) B&B algorithm. The difference is that backtracking does not use a bounding operator to detect unpromising nodes. However it may incorporate pruning mechanisms, for instance based on evaluating the feasibility of a subproblem, which can be seen as a binary bounding function.

Compared to complete enumeration the pruning of branches significantly reduces the size of the explored tree. However, for many COPs the execution time of B&B significantly increases with the input size and only small or moderately sized instances can be practically solved with sequential algorithms. For this reason, the use of parallel computers is an attractive way to deal with larger instances of COPs. One might argue that there is no point in applying B&B to NP-hard COPs, unless we have an exponential number of processors for parallel processing. However, the approach may actually perform well for a given problem and parallel computing may provide the necessary computing power to solve instances up to a certain size. Running times for B&B are very hard to predict because it requires an estimate of the tree-size. This means that the decision whether an exact algorithm for a given problem is useful, can only be founded on empirical data.

The combination of approximate and exact methods is a promising approach. For instance, metaheuristics can be used to accelerate the search process of B&B, and B&B may provide candidate solutions that improve the quality of the approximate method. As the focus of this thesis is put on the efficient design of B&B on parallel computers, we refer the reader to [Meh11; Tal09] for more information on this subject.

1.3 Branch-and-Bound algorithms

This section provides a comprehensive overview of the B&B algorithm. As we focus on permutation-based problems, it is assumed in the following that B&B is applied to solve a permutation problem.

1.3.1 Terminology and general description

The B&B algorithm proceeds by implicit enumeration of all the solutions of the problem being solved. Exploration of the space of potential solutions (search space) is performed by dynamically building a tree where:

- The **root node** represents the initial problem to be solved (the search space X).
- **Internal nodes** represent subproblems of the initial problem (subspaces $S \subset X$).
- **Leaf nodes** are possible solutions.

A complementary and useful point of view is to consider B&B as a method for iteratively constructing solutions. Suppose B&B is applied to solve a permutation problem of size 4 which consists in finding an optimal scheduling of the jobs $\{1, 2, 3, 4\}$ [2]. If none of the jobs is fixed at a particular position, all 4! permutations can be attained from this state. This initial state corresponds to the root node and will be denoted $/1234/$. The jobs written before the first slash symbol ("/") are scheduled at the beginning, jobs behind the second ("/") symbol are scheduled at the end, and jobs in between are not yet scheduled. A possible way of constructing candidate solutions is to fix each of the unscheduled jobs successively at the first, second, third position, and so on. At the first level this yields four internal nodes $1/234/, 2/134/, 3/124/, 4/123/$. The number of scheduled jobs in an internal node is called its *depth* or its *level*. A leaf node is a node in which all jobs are scheduled, for example $234//1$, or simply 2341. Both points of view, recursive "partitioning of search space" on the one hand and iterative "construction of solution" on the other, provide equivalent descriptions of the B&B tree and vocabulary related to both is used interchangeably.

All nodes generated and not yet processed are kept in a data structure. In the beginning this data structure contains only the initial problem. The algorithm saves the best solution found so far (also called the *incumbent*) as well as the associated cost (also referred to as the *upper bound*). The latter is either initialized at ∞ or at the cost of a feasible solution known beforehand (for instance, found by an approximate method). This upper bound can be improved from an iteration to another. At each iteration of the algorithm:

- The **bounding operator** is used to compute a lower bound on the minimal cost of a subproblem. In order to include the possibility of applying the algorithm to constraint satisfaction problems, the bounding operator is frequently called

2. Throughout the remainder of this thesis we will refer to the elements of a permutation as *jobs*.

node evaluation function in this document. The node evaluation function solely depends on the problem being solved. There are essentially two possible modes of evaluation, called lazy and eager evaluation [CP99]. In the eager evaluation mode bounds are computed as soon as nodes are generated, i.e. bounding is called after the branching operator. In the lazy evaluation mode bounds are only computed if really necessary, i.e. after selection and before the branching operator. The algorithms presented in this thesis use the eager evaluation mode.

- The **pruning operator** uses the lower bound value of a subproblem to decide whether it is kept in the data structure for further exploration or discarded from the search. Infeasible subproblems and problems whose lower bound is greater than the best solution found so far can be eliminated.

- The **branching operator** divides a subproblem into several smaller, pairwise disjoint subproblems. In general, this is achieved by partitioning the search space into smaller subspaces on which the same optimization problem is defined. The internal nodes generated by branching a node A are called A's *children*. For permutation problems, a possible branching scheme consists in assigning all untried alternatives to a fixed position in the permutation. This is known as polytomic branching [Rou87], in contrast to dichotomic branching, which generates two children per node. The following polytomic branching scheme is used in this work. Two sets of children nodes are generated, by fixing jobs at the first free position at the beginning and at the end, respectively. Both sets are evaluated and the set which is likely to lead to a smaller B&B tree is retained (based on a heuristic estimate). Generated subproblems are added to the data structure which is used to store unexplored subproblems.

- The **selection operator** chooses the next subproblem to be expanded among all pending subproblems. This choice follows a predefined exploration strategy. The selection of a node could be based on its depth, which leads to a depth-first exploration strategy (DFS). In DFS the entire subtree rooted in the current node is fathomed before another node is processed. A best-first selection strategy could also be used. It is based on the presumed capacity of the selected node to yield good solutions. However, memory requirements for pure best-first search are often excessive. DFS prescribes no particular order of exploration among sibling nodes. Therefore, nodes on the same level can be processed in increasing order according to their lower bounds. Unless indicated differently the B&B algorithms presented are based on this best-bound DFS exploration strategy.

Whenever the algorithm reaches a valid solution, this latter is evaluated and compared to the best solution found so far. If an improvement of this latter is possible, it is updated. The algorithm stops when the selection operator fails to choose a node to expand because the pool of pending nodes is empty. At the end of this dynamic exploration process the best found solution is proven to be optimal.

The size of the explored B&B tree strongly depends on the quality of the bounding operator, i.e. the tightness of the lower bounds. Moreover, for a given bounding function the tree size also depends on the rate at which the upper bound decreases towards the optimal cost - in other words, on the search strategy defined by the selection operator.

If the algorithm is initialized at an optimal solution $x^\star$, then B&B explores exactly the nodes x for which $LB(x) < f(x^\star)$[3]. Independently from the search strategy, these nodes must be explored in order to prove the optimality of $x^\star$. The tree formed by these (critical) nodes is called the *critical tree*.

1.3.2 Models for parallel Branch-and-Bound

The parallelization of B&B is well-studied and different classifications have been proposed [Mel05; TdB92; GC94]. The taxonomy presented in this subsection is the one from [Mel05] which is based on the classification of Gendron and Crainic [GC94]. Four models are identified: (1) parallel tree exploration, (2) parallel evaluation of bounds, (3) parallel evaluation of a bound, and (4) multi-parametric.

Parallel tree exploration model

In the parallel tree exploration model several disjoint search subspaces (branches of the B&B tree) are explored in parallel. This means that all four operators, selection, branching, bounding and pruning, are applied in parallel to different subproblems. This can be done either synchronously or asynchronously. In synchronous mode the B&B algorithm has several phases in which the B&B processes perform their exploration independently. The B&B operators are applied in parallel to multiple data (e. g. work pools, nodes, lower bounds). Between these phases exploration processes are synchronized and can exchange information, such as the best solution found so far. In asynchronous mode the B&B processes communicate in an unpredictable manner.

Among the four models, the parallel tree exploration model is the most frequently used. One of the reasons is that the degree of parallelism in this model can be very

3. Or $LB(x) \leq f(x^\star)$, if the goal is to find *all* optimal solutions, not only *one*. For all problems considered in this thesis, except n-Queens, the goal is to find one optimal solution and/or prove its optimality.

high, especially when solving large instances. Each B&B process holds one or several work units which correspond to subspaces of the total search space. The number of explorers that can be kept busy simultaneously therefore depends on (1) the rate at which new subproblems are generated and (2) the efficient assignment of subproblems to B&B processes. In both modes, synchronous and asynchronous, load balancing is one of the main issues raised by this model. Indeed, as the B&B tree is highly irregular, some branches contain much more work than others. Among other issues are the placement and management of the set of pending subproblems. Especially in distributed contexts, the communication of the best solution found so far and termination detection also become challenging.

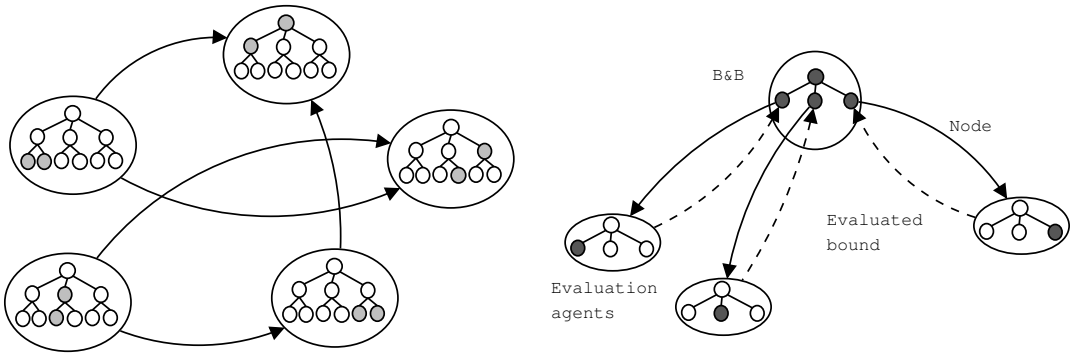
This parallelization model alters the order in which nodes are explored and therefore the size of the explored tree. A parallel tree exploration B&B may indeed find an optimal solution much faster than its sequential counterpart, or, on the contrary, expand much more nodes to find such a solution. In other words, in parallel tree-search B&B the number of expanded non-critical nodes does not only depend on the search strategy, but also on other factors, like work load distribution and communication schemes. Therefore, in synchronous and asynchronous parallel tree exploration B&B speedup anomalies may occur [DKT95].

Aside from the potentially very high degree of parallelism, the popularity of this parallelization approach is also due to its genericity. As this model does not affect the bounding operator, it can be applied to any COP without prior knowledge of its characteristics, at least in principle. Therefore, this model is used by most frameworks that aim at facilitating the implementation of B&B algorithms. This parallelization model corresponds to the ”type 2” (tree-based) parallelism in the classification of Gendron and Crainic [GC94]. It falls in the category of ”high-level” parallelization in the classification of Trienekens and de Bruin [TdB92].

Parallel evaluation of bounds model

Another source of parallelism is the parallel evaluation of bounds. After the decomposition of a node all its children may be evaluated in parallel. The degree of parallelism in this model depends on the branching scheme and varies according to the depth of a node in the tree. In order to reach a high degree of parallelism the selection and branching operators can be applied multiple times until a pool of children nodes is large enough to be efficiently evaluated in parallel [Cha13]. This model leads to more fine-grained parallelism than the parallel tree exploration model. It is well-suited in cases where the evaluation is costly. Moreover, if the node evaluation is regular (in the sense that

each node represents approximately the same amount of work), it is well-suited for Single Instruction-Multiple Data (SIMD) processing. This model can be nested inside the parallel tree exploration model. As the previous model, this model is generic in the sense that it requires no knowledge of the particular problem being solved. If nodes are evaluated directly after their generation, this model does not change the amount of work done or the shape of the B&B tree.



(a) Parallel tree exploration model (b) Parallel evaluation of bounds model

Figure 1.1: Illustration of main parallelization models used in this thesis.

Parallel evaluation of a bound/solution model

At an even lower level, the evaluation of a bound or solution itself may be parallelized in some cases. This depends on the possibility of parallelizing the node evaluation function that is used. As this model only modifies the bounding operator it can be nested inside both previous models to add a third level of parallelism. For problems with a very compute intensive bounding function the model may also be used alone. The parallel evaluation of a bound model has no impact on the search trajectory. Like the previous model, it is also known as ”node-based” [GC94] or ”low-level” [TdB92] parallelization.

Multi-parametric model

Of the four identified models the multi-parametric model is the less studied one. It is a coarse-grained parallelization that consists in launching multiple independent B&B processes that explore the same search space with different algorithm parameters. For instance, the different B&B algorithms may use different branching schemes or selection strategies. In [KK84] only one of the B&B algorithms uses the actual current upper bound (UB) while the others are ϵ-approximations using $UB - \epsilon$ ($\epsilon > 0$) as upper bound. In this

model redundant explorations are very likely to occur.

1.3.3 Challenges in parallel Branch-and-Bound

Irregularity. Most of the difficulties that arise when implementing a parallel B&B algorithm are direct consequences of the algorithm's irregularity. In each of the four presented parallelization models irregularity manifests itself in a different way. Because of the unpredictable pruning of branches, some subproblems require much more computation time than others, which leads to load imbalance. As subproblems are dynamically assigned to processing units at runtime, dynamic load balancing is a requirement for an efficient exploitation of the available computing resources. The parallel evaluation of nodes yields a more fine-grained and often more regular workload. However, the evaluation of nodes may require a variable amount of time, depending on the problem being solved and the depth of each node.

The search space management as well as the problem-dependent node evaluation subroutines are often characterized by highly irregular control flows. In a permutation problem the memory accesses during node evaluation depend on the partially constructed solution. Many accesses are therefore masked and have irregular and unpredictable strides. Diverging instruction flows and random memory access patterns are well-known to be major obstacles to SIMD processing. This may compromise the efficient usage of GPUs and/or vector processing units - two of the main factors that have increased the performance of HPC systems over the last decade.

Work pool management. We call *work pool* the data structure that is used to store generated and not yet evaluated subproblems. A subproblem is usually implemented as a structure containing all information necessary for its evaluation. The role of the work pool is essentially to allow the insertion/retrieval of subproblems. Moreover, the work pool may maintain subproblems in a certain order, facilitating the implementation of a search strategy. For instance, DFS corresponds to processing nodes in last-in first-out (LIFO) order and is therefore naturally implemented by a stack. Best-first search is usually implemented using priority queues. In general, priority queues offer good flexibility, because it is enough to change the sorting criterion to implement another search strategy.

In parallel B&B there are different strategies for implementing the work pool. One approach is to use a single centralized work pool, concurrently accessed by all workers to pick subproblems for branching/evaluation. In shared memory systems all workers concurrently access the same work pool to get subproblems. After branching the subproblem each worker (thread) inserts the evaluated and non-pruned children into the pool. In

single-pool approaches synchronization between workers to gain exclusive access to the pool is inevitable. These concurrent accesses may create a bottleneck and memory contention, limiting the scalability of the approach. In distributed memory configurations the single pool strategy is implemented using the master-worker paradigm. Again, the scalability of this approach is limited as the master process becomes a bottleneck.

Multiple-pool approaches aim at solving this issue by using several pools. There are variants of multiple pool B&B algorithms. The most popular are collegial, grouped and mixed [GC94]. In the collegial variant each worker has its private pool. The grouped approach uses one shared pool for a group of workers and the mixed variant combines both approaches using a hierarchy of pools. Multiple pool strategies alleviate the bottleneck problem that occurs in single-pool approaches but they raise the issue of balancing the workload between multiple pools. Also, the sharing of knowledge among workers, like the best known solution and termination detection, become non-trivial. Generally speaking, multiple pool approaches require more sophisticated communication models than single pool approaches.

Load balancing. While the single work-pool approach implicitly balances the work load among workers, multiple-pool approaches require explicit dynamic load balancing. Over the last decade work stealing [BL99] has been widely adopted as a standard way to distribute tasks among workers. In the context of B&B tasks are subproblems (nodes of the B&B tree). In a work stealing algorithm, each thread uses a double-ended queue (deque) for storing tasks. Locally, a thread uses the tail of its deque as a stack, popping tasks to execute and pushing newly generated tasks onto the stack. When a thread's deque is empty it becomes a *thief* and steals tasks from the head of another thread (called *victim*). Expressed in terms of a multiple-pool B&B algorithm, when a work pool is empty, one thread associated with the pool steals nodes from another work pool.

There are mainly two reasons for using deques in work stealing approaches. The first is that stealing tasks from the head of the deque may allow the victim thread to continuously work on the deque's tail without being slowed down by steal operations (work-first principle [FLR98]). A first non-blocking work stealing deque that prevents contention during concurrent operations was proposed by Arora, Blumofe, and Plaxton [ABP01]. Dinan *et al.* [DLS+09] propose a work-stealing deque partitioned into a local and a public portion using a periodically updated pointer. This data structure, called a split-queue, allows lockless accesses to both portions of the deque and requires locking only for updating the split-pointer. The scalability of this work stealing using split-queues has been demonstrated on up to 8 192 distributed processing cores. In [ACR13] it is

reported that concurrent deque operations require expensive memory fences, which has led to recent interest in implementations of work stealing with non-concurrent (private) data structures [ACR13; vDvdP14]. The second reason concerns the granularity of the work stealing mechanism. In B&B, like in many task parallel applications, nodes at the bottom of the task stack (i.e. older tasks) represent a larger amount of work as recently spawned tasks on top of the stack. Granularity (i.e. the number of nodes that are stolen) is one of the characteristic features of a work stealing strategy, together with the policy of selecting the work stealing victim. A recent survey of work stealing methods for scheduling parallel computations can be found in [YH17].

Data Structures. The two previous paragraphs illustrate the central role of the data structure used for the storage of the huge number of subproblems. As mentioned, work pools are usually implemented as stacks, deques or priority queues. Operations on the B&B tree, like node selection, insertion of branched nodes and work transfers between multiple pools are implemented as push and pop operations on dynamic sized data structures. We generically refer to this type of dynamic data structure as *linked-lists* (LL).

Using such data structures has many advantages. For instance, it is relatively easy to adapt B&B to different problems by changing the definition of a node. Similarly, the search strategy can be modified easily, for instance by using a different sorting criterion for a priority queue. However, departing from the general case of B&B, the particular structure of a class of problems can be exploited, making it possible to use other types of data structures.

For example, a DFS-B&B applied to a $0-1$ integer COP could use a single bit array of length n (problem size) and an integer d indicating the current depth of the search. The values of the bit array up to depth d represent the current partial solution and on each level the algorithm successively tries the alternatives $0/1$. Branching consists in incrementing the current depth d. Backtracking is performed by decrementing d and incrementing the bit array. The basic idea is that it is not necessary to explicitly store all frontier nodes, because they can be deduced from the current active node. It should be emphasized that this is possible because the maximal B&B tree (which would be explored if no pruning was used) is known in advance, and the exploration order (DFS) is deterministic.

For permutation problems the structure of the search space can also be exploited to design alternative data structures for DFS-B&B. One example are bitsets, which allow a very compact implementation of DFS [SRR08; Ric97]. For instance, DFS-B&B for permutation problems can be implemented using only one vector and two integers for

the search procedure. A vector is used to store the current partial solution and the first integer indicates the current depth of the search. The second integer is seen as a bitset that keeps track of already scheduled jobs. The bit k of this second integer is set if and only if job k is already scheduled. Searching the B&B tree in depth-first order is performed by incrementing the positions of the vector while using the integer to check whether a partial solution is valid. Such a bitset-based approach has a very small and constant memory footprint. Bitset-based sequential implementations of backtracking can be extremely fast, for instance for solving the n-Queens problem [Zon02; PE17; Som]. However, it is unclear how dynamic load balancing could be performed with such bitset-based data structures. Also, as only one node is generated at once, it does not seem possible to use the parallel evaluation of bounds model.

The so-called Integer-Vector-Matrix (IVM) data structure [MLMT14] is an innovative data structure dedicated to solving permutation-based COPs. In terms of flexibility and compactness, IVM can be seen as a compromise between bitsets and linked-lists. IVM is more flexible than bitsets, but less compact. It is more compact than LL-based data structures, but offers less flexibility. IVM uses an integer to indicate the current depth of the search, a vector to indicate the path of the current node, and a matrix to store the unscheduled jobs at each level. Like bitset-based B&B, IVM-based B&B can be performed with constant memory requirements. IVM also allows to define splittable work units - intervals - which can be exchanged between workers to implement dynamic load balancing. In [Ler15] the advantages of using IVM in multi-core B&B algorithms, compared to conventional linked-list, are shown theoretically and experimentally. The reported results show that, for 20×20 FSP instances, IVM-based B&B consumes on average 60 times less memory, requires about 9 times less CPU time for pool management, performs less context switches and produces less page faults than its LL-based counterpart. However, IVM does not offer the same flexibility as LL-based data structures: as mentioned, IVM is dedicated to permutation-problems and depth-first search.

All algorithms presented in this thesis are based on the IVM data structure. A detailed description of IVM and the IVM-based B&B algorithm are provided in Section 2.2.

1.4 Computing Environments

High Performance Computing (HPC) technologies are evolving at high pace and architectures of computing systems become increasingly complex. The programmer has to understand the hierarchical organization of these machines in order to take advantage of the full computing power they are able to provide. A detailed technical description of

the hardware used in this thesis, or a discussion of current and future developments in HPC goes beyond the scope of this chapter. However, the design of algorithms presented in this thesis is, to a large extend, guided by the architecture of the targeted computing platform. Therefore, this section attempts to provide some context, giving a brief outline of current trends and challenges in HPC and a succinct description of the hardware used in this thesis. There is abundant literature discussing the impressive evolution of computing systems over recent years, for example [GR14; EBS+11; Dre07; KDK+11; Mär14; OHL+08], to cite only a few. Appendix A.3 lists the main technical specifications of computing devices used in this document.

Energy efficiency of computing systems has become very important. When CPUs had a single processing core, performance increase was mainly achieved by increasing the clock frequency and through improved instruction pipelining. As increasing frequencies led to unsustainable power consumptions, over the past ten years performance was improved by using multiple cores running at slightly lower frequencies. This is made possible by the shrinking manufacturing process (2004: 90 nm, 2017: 14 nm) and the subsequent growth of transistor count. In the latest edition (June 2017) of the biannual TOP500 [16] ranking of the worlds fastest supercomputers no system has less than 4 CPU cores per socket and more than half of the systems in the TOP500 list have at least 12 cores per socket or more. The x200 generation of Intel's Xeon Phi processors (code-named Knight's Landing) are composed of more than 60 cores operating at a base frequency of 1.4 GHz each.

In order to increase core-level performance, in recent years the trend has been to use wider SIMD vector instructions. Most modern multi-core processors have instruction set extensions, allowing to operate simultaneously on multiple data objects residing in the same registers. Therefore it becomes increasingly important to exploit data parallelism (SIMD) besides instruction-level parallelism (ILP) and thread-level parallelism (TLP). For instance, Intel's Xeon Phi processors provide 512-bit wide vector registers, allowing up to 16 single-precision operations to be performed in one clock cycle. Launched in 2008, the AVX (up to 256-bit) vector extensions for x86 processors can substantially improve per-core performance, provided the compute-intensive portions of the executed code allow SIMD processing.

Including all these levels of parallel processing capability, the theoretical single-precision peak performance of a multi-core processor is given by:

$$\#\text{cores} \times \text{clock rate} \times \#\text{instructions/cycle}$$

This shows that, without efficient exploitation of thread-level parallelism and, if possible, data parallelism one can only achieve a small fraction of the theoretical peak performance.

A major issue is that the speed and efficiency of the memory subsystem is not improving proportionally to the advances in processors. The cost of moving data is usually greater than the cost of performing operations on it [GR14]. Besides bandwidth, an important aspect of the memory subsystem is latency, the minimum amount of time required to fetch a single piece of data from main memory. For both, CPUs and GPUs, latencies for accessing main memory are counted in hundreds of clock cycles. Main memory latency can be hidden by prefetching cache lines into different levels of the cache hierarchy, bringing data closer to the processor. However, this requires predictable memory accesses and applications with scattered, irregular memory access patterns are often latency-bound. In order to deal with this issue, many CPUs use a large portion of the chip for complex management, like speculation, resolution of data dependencies and out-of-order execution of instructions. A study has shown that the energy consumption of these "overhead" operations is as high as 36% of the total dynamic energy consumption, even more than the energy consumption of data movements ($\sim$ 25%) [KGKH13].

GPUs, in contrast, use a large number of smaller, in-order cores which execute groups of threads in lockstep (SIMD). Compared to CPUs, a much larger part of the chip area is dedicated to ALUs, and L1 and L2 caches are much smaller. Generally speaking, GPUs deal with the memory latency issue by combining fast context-switching and massive multi-threading. The instruction scheduler issues warps [4] from a pool of resident warps, prioritizing threads which have their input data ready. This means that a large number of threads are launched - for instance, the maximum number of resident threads in Nvidia's Pascal GPUs exceeds 100 000 threads. However, streaming multiprocessor (SM) occupancy (ratio of active warps to maximum number of warps per SM) is not the only criterion for achieving good performance on GPUs.

The reality is much more complex than the brief outline provided here. Understanding how GPUs hide latencies and modeling GPU performance are active fields of research ([Vol16; Li16] are two recent dissertations on these topics, containing much more detailed descriptions of GPU architecture than we could provide).

GPU computing. 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

4. group of (32) threads, in CUDA terminology. Until now all CUDA capable device have a warp size of 32.

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 1.2, a GPU is a coprocessor, coupled to a CPU through a PCI Express bus[5]. 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 (SM) including each a pool of 32-bit or 64-bit SIMD processors (processing cores).

For instance, a Pascal P100 GPU device contains 56 SMs of 64 Cuda cores for a total of 3 584 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.

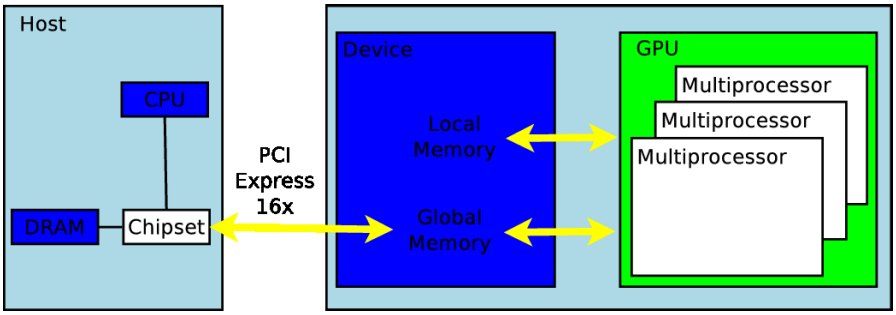


Figure 1.2: Hardware view: GPU = coprocessor of CPU.

From software programming point of view, as illustrated in Figure 1.3 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 issues kernels to one or more streams, for execution on the GPU device. Each stream is a first-in first-out (FIFO) queue

5. For x86 CPUs the connectivity is only through PCI Express, for now. For Power8 CPUs, full NVLink connectivity may replace PCI Express.

of kernels and other Cuda calls, and multiple streams can be used concurrently. Kernels are executed according to a two-level parallelism: at the higher level the processors (SMs) execute the thread kernel according to the Single Program Multiple Data (SPMD) model. At the lower level (intra-SM), the threads are executed according to the Single Instruction Multiple Data (SIMD) or Single Instruction Multiple Thread (SIMT) model. Indeed, inside each processor the instruction flow composing a thread kernel is executed according to the SIMD model.

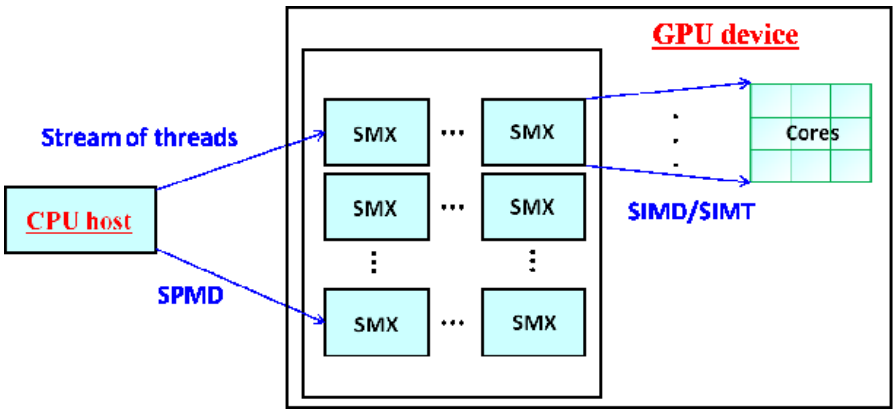


Figure 1.3: Software view: Parallel program = weakly parallel/serial host code + massively parallel device code.

The programmer configures the kernel launch by specifying several parameters: the hierarchical organization of threads into grids of blocks of threads, the amount of shared memory allocated to each block of threads and the stream to which the kernel is issued.

Grids are $1D$ or $2D$ arrays of blocks and blocks are $1D$, $2D$ or $3D$ arrays of threads. The thread organization corresponds to the organization of application data which are often vectors, matrices or volumes. As shown in Figure 1.4, the blocks are assigned to the SMs by the Cuda runtime. Inside each SM each block is split into warps, i. e. pools of 32 threads. Warps are scheduling units, i. e. the threads are executed *in lockstep* 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.

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.

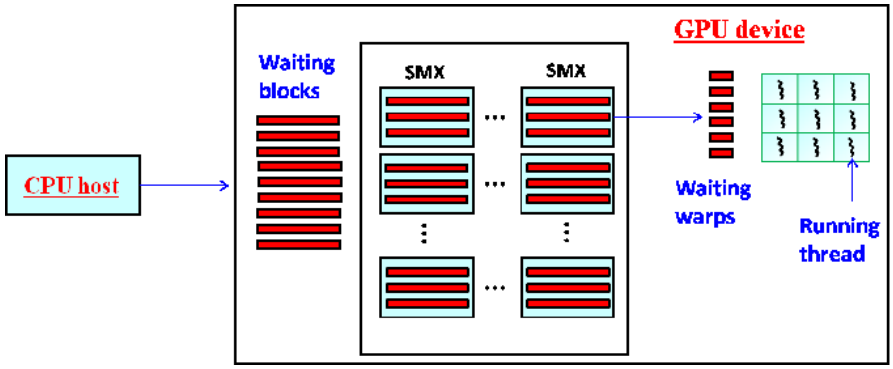


Figure 1.4: Software view: Parallel program = grid(s) of block(s) of threads executed as warps of 32 threads.

1.5 Related work

The design of parallel B&B algorithms is strongly influenced by the target architecture and the characteristics of the problem being solved [BHP05]. Therefore, and in spite of the simple, generic formulation of B&B, a large number of parallel algorithms have been proposed for different problems and architectures. Gendron and Crainic [GC94] provide a complete, but over twenty year old survey of parallel B&B.

1.5.1 B&B for multi-core CPUs

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

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

began its existence as the "core" layer of the parallel mixed integer programming (MIP) solver PICO.

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

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

Evtushenko, Posypkin, and Sigal [EPS09] present a software platform called BNB-Solver, allowing the use of serial, shared memory and distributed memory B&B. The proposed approach uses a global work pool and local work pools for each thread. Each thread stores generated subproblems in its local pool during N B&B iterations. After N iterations a part of the local nodes are transferred to the global pool. When the local pool of a thread is empty, the thread attempts to take nodes from the global pool and blocks if the global pool is empty. The algorithm terminates when the global pool is empty and all threads are blocked. The authors compare the performance of BNB-Solver with the ALPS and PEBBL frameworks and obtain results similar to Herrera *et al.* [HSH+17], in the sense that, for a knapsack-problem (with a reported sequential node processing rate in the order of 1 Mn/s) BNB-Solver outperforms both frameworks.

Casado *et al.* [CMGH08] propose two schemes for parallelizing B&B algorithms for global optimization on shared memory multi-core systems, Global- and Local-PAMIGO (Parallel advanced multidimensional interval analysis global optimization). Both algorithms are parallelized using POSIX threads. In Global-PAMIGO, threads share a global work pool and therefore a synchronization mechanism is used for mutually ex-

clusive accesses to the pool. For Local-PAMIGO, where thread has its own pool of subproblems, a dynamic load balancing mechanism is implemented. A thread stops when its local pool of subproblems is empty. When the number of running threads is less than the number of available cores, and a thread has more than one subproblem in its local pool it creates a new thread and transfers a portion of its pool to the new thread. Local-PAMIGO ends when there exists no more running threads, and Global-PAMIGO ends when the global pool is empty. The authors report profiling results for PAMIGO which show that memory management represents a large percentage of the computational burden. As a very large number of subproblems are created in a relatively short amount of time, the kernel needs to satisfy memory allocation and deallocation requests from all threads, creating memory contention.

The vast majority of parallel B&B algorithms in the literature store subproblems in one or several pool(s) implemented as linked-lists (e. g. priority queues, stacks, deques). As mentioned, a multi-core B&B based on the IVM data structure was proposed in [MLMT14; LMMT07]. Because of the direct relevance for this thesis, sequential and multi-core IVM-based B&B are presented in the beginning of Chapter 2.

1.5.2 B&B for Graphics Processing Units

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

The GPU-B&B and backtracking algorithms for **fine-grained** problems proposed

in [CMNL11; CNNdC12; FRvLP10; LLW+15b; RS10; ZSW11] perform massively parallel searches on the GPU, based on the parallel tree exploration model. The evaluation of a node for the n-Queens problem in [FRvLP10; LLW+15b; ZSW11] requires only a few registers of memory and only a couple of bit-operations. The lower bound for the Asymmetric Traveling Salesman Problem (ATSP) used in [CMNL11; CNNdC12] is incrementally obtained by adding the cost of the last visited edge to the current cost and therefore has a complexity of $O(1)$. It requires an access to the distance matrix which can be stored in constant or texture memory. The size of the problems being solved is < 20 for both the ATSP and the n-Queens problems. These algorithms for fine-grained problems share a common approach: the search is split in two parts, an initial CPU-search and a final GPU-search. The upper tree of depth d_{cutoff} is processed in sequential or weakly parallel manner on CPU, generating a set of active nodes at depth d_{cutoff}. This active set is sent to the GPU, where the lower part of the tree is processed in parallel. Each node of the active set is used as root node for an independent search, which is mapped either to a thread or a warp. This approach requires very careful tuning of the cutoff depth, which strongly influences granularity and load balance.

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

For many COPs the cost of the bounding operator is very high, compared to the rest of the algorithm. For instance, the most used lower bounding function for the FSP consumes $97 - 99\%$ of the sequential execution time [MCB14]. However, the cost of evaluating one node is sufficiently small to be efficiently performed by a single GPU-thread. We therefore refer to this type of problem as **medium-grained**. For these problems, existing GPU-accelerated B&B algorithms in the literature use the GPU to evaluate large pools of subproblems in parallel [CMMB13; VDM13; LE12]. They use conventional stacks or queues to store and manage the B&B tree on the host, offloading the parallel evaluation of bounds to the device. Indeed, for these problems substantial speedups can be achieved despite sequentially performing pool management on the host. Substantial efforts have been made to port larger portions of the algorithm to the GPU and to reduce overheads incurred by data transfers between CPU and GPU. For instance, branching nodes on the

device allows to copy only parent nodes to the GPU. Similarly, pruning evaluated nodes on the device reduces the sequential portion and requires only the transfer of non-pruned children nodes back to the host. Further performance improvements can be obtained by overlapping data transfers with GPU computations, as for example in [VDM13]. For fine-grained problems this approach is likely to perform poorly.

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

1.5.3 Hybrid and distributed parallel B&B

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

Some of the largest known exact resolutions of COPs have been performed using the Master-Worker paradigm in combination with grid computing technologies (e. g.

nug30 [ABGL02]). The B&B@Grid platform [MMT07a] uses an interval-encoding for work units which significantly reduces the size of messages communicated in distributed B&B. Designed for volatile computing environments, B&B@Grid is fault-tolerant thanks to its checkpointing mechanism. As B&B@Grid constitutes the foundation of the hybrid distributed B&B presented in this thesis, the approach is described in more detail in in Chapter 4. In [BMT12a] an adaptive multi-layer hierarchical master-worker approach is applied to the B&B algorithm, using FSP as a test-case. The proposed approach evolves as new resources join the computation, and integrates three types of processes, a super-master, masters and workers. Results obtained at the scale of up to 2 000 CPUs show that the multi-layered hierarchical approach clearly outperforms single-layered and classical Master-Worker approach in terms of efficiency for instances smaller than *Ta056*, as it minimizes bottlenecks at the level of the master and reduces idle time of the workers. In [BMT14] the authors extend their approach, proposing a fault-tolerance mechanism.

1.6 Test-cases: Permutation-based COPs

As mentioned, the nature of the permutation problem to be solved has a strong impact on the performance of the B&B algorithm. Using different test-cases allows to better understand the algorithm's behavior and reveals its strengths and weaknesses depending on the tackled problem. In this thesis, three permutation-based problems are considered: Flowshop Scheduling Problem (FSP), Quadratic Assignment Problem (QAP), n-Queens. The used node evaluation functions have different computational complexities and memory requirements.

1.6.1 Flowshop Scheduling Problem (FSP)

The FSP is the main test-case considered in this thesis. It is defined by a set of n jobs and m machines, arranged in a certain order. As illustrated in Figure 1.5, jobs are processed according to the chain production principle, meaning that a job cannot be processed on a machine j before it has finished processing on all upstream machines $0, 1, \cdots, j-1$. The n jobs have to be processed in the same order on each machine, and the processing of a job cannot be interrupted: solutions are therefore naturally encoded by permutations.

A $m \times n$ processing time matrix contains the time required for a machine to finish the processing of a job. The goal is to find a permutation schedule that minimizes the total processing time called *makespan*. In [GJS76], it is shown that the minimization of *makespan* is NP-hard from 3 machines upwards. The lower bound proposed by Lageweg, Lenstra, and Kan [LLK78] is used in our bounding operator. This bound is known for its

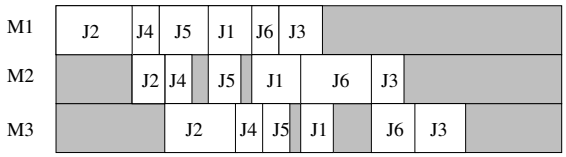


Figure 1.5: Example of a solution of a flowshop problem instance defined by $n = 6$ jobs and $m = 3$ machines.

good results and has complexity of $O(m^2 n \log n)$. This lower bound is mainly based on Johnson's theorem [Joh54] which provides polynomial time procedure for finding an optimal solution for solving the 2-machine FSP.

The most used benchmark instances used in the literature are the ones defined by Taillard [Tai93]. These instances are divided into 12 groups: 20 × 5 (i.e. group of instances defined by 20 jobs and 5 machines), 20 × 10, 20 × 20, 50 × 5, 50 × 10, 50 × 20, 100 × 5, 100 × 10, 100 × 20, 200 × 10, 200 × 20, and 500 × 20. In each group, 10 different instances are generated. For each instance, the duration of each job on each machine is randomly generated by [Tai93]. The instances of the 6 groups where the number of machines is equal to 5 or 10 (i.e. 20 × 5, 20 × 10, 50 × 5, 50 × 10, 100 × 5, 100 × 10, and 200 × 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 sequential 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 (i.e. 50 × 20, 100 × 20, 200 × 20, and 500 × 20) are very hard to solve. The only instance defined with 50 jobs and 20 machines exactly solved up to this day is *Ta056*. It's resolution using B&B@Grid [MMT07b] lasted 25 days, exploiting on average of 328 processors at 97% efficiency, i. e. its sequential computation time is estimated at 22 years. In many of our experiments we use the group of instances where the number of machines and the number of jobs are equal to 20. The resolution of the instances on a sequential computer is feasible but consumes enough computing time to justify the use of parallel processing. For these instances Table 1.1 shows the size of the critical tree, the time spent by a sequential B&B for exploring these trees (on E5-2630v3) and the corresponding node processing rate (in decomposed nodes/sec). As throughout the entire document, tree sizes and processing rates refer to the number of *decomposed* nodes, meaning that eliminated nodes are not counted. The units n/s, kn/s and Mn/s are used to designate 1, respectively 10^3 and 10^6 decomposed nodes per second. To ensure that exactly the critical tree is explored the B&B algorithm is initialized at the (known) optimal solution. Therefore the B&B proves the optimality of this solution. Solving all 10

of Taillard's 20 × 20 instances requires 63 hours of sequential processing time, one third of which is used to solve the largest of these instances, *Ta023*.

Table 1.1: Size of the critical B&B tree of Taillard's instances *Ta021-Ta030* (#decomposed nodes), corresponding sequential exploration times (in seconds, on Xeon E5-2630v3 CPU) and node processing rates.

Instance	**21**	**22**	**23**	**24**	**25**
#Nodes (in 10^6)	41.4	22.1	140.8	40.1	41.4
$\mathbf{T}_{sequential}$	6h38m	3h10m	21h31m	5h22m	6h51m
n/s	1 734	1 927	1 822	2 069	1 680
Instance	**26**	**27**	**28**	**29**	**30**
#Nodes (in 10^6)	71.4	57.1	8.1	6.8	1.6
$\mathbf{T}_{sequential}$	9h22m	7h38m	1h14m	0h59m	0h15h
n/s	2 117	2 074	1 822	1 892	1 888

1.6.2 Quadratic Assignment Problem (QAP)

The QAP was introduced by Koopmans and Beckmann [KB57] in 1957 as a mathematical model for the allocation of indivisible economic activities. It consists in assigning n facilities to n locations, given a $n \times n$ distance matrix (d_{ij}) and the flow between facilities in a flow matrix (f_{ij}). The objective is to find an optimal assignment that minimizes the total cost, which is given by the sum of distances multiplied by flows.

The QAP can be formulated as an optimization problem on the set of permutations of n integers $1, 2, \ldots, n$. In this formulation the positions in the permutation represent the n locations. In this formulation the QAP writes:

$$\min_{\pi \in S_n} \sum_i \sum_j f_{ij} d_{\pi(i)\pi(j)}$$

Figure 1.6 illustrates a solution of a QAP of size $n = 3$. A review for this problem including different mathematical formulations, applications and a complete state-of-the-art of exact and heuristic methods applied to the QAP can be found in [LdAB+07].

We use the well-known Gilmore-Lawler lower bound (GLB) for the QAP. This bound is used very frequently, although the quality of GLB deteriorates rapidly as the problem size increases [LPRR94]. The GLB is obtained by the resolution of based on the resolution of a linear sum assignment problem (LSAP) which can be solved using the so-called Hungarian Algorithm. The computational complexity of GLB is $O(n^3)$. We use different instances of size $n = 16$ to $n = 20$ from the QAPLIB library [BKR97] as test-cases.

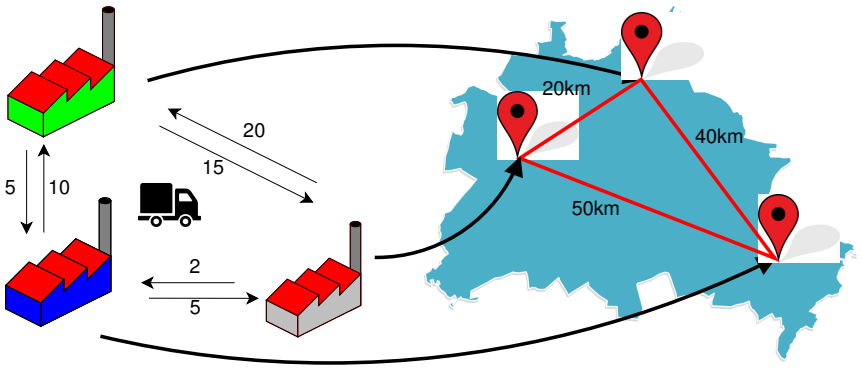


Figure 1.6: Illustration of the QAP for for $n = 3$.

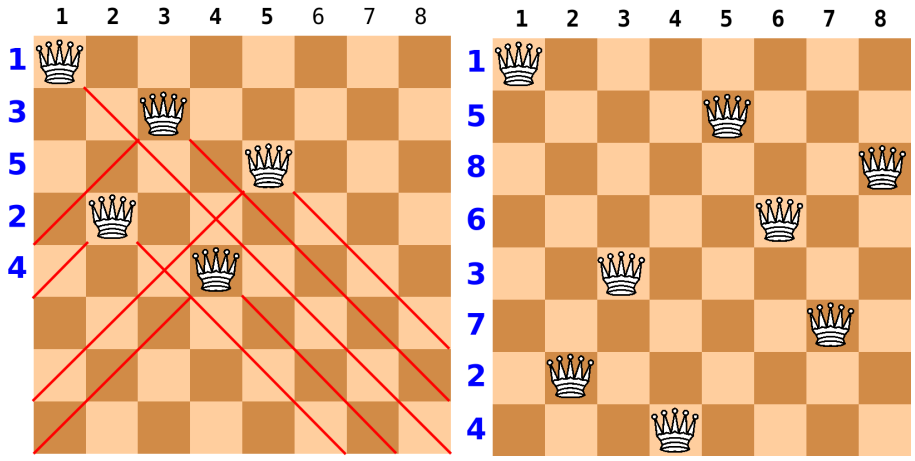
1.6.3 n-Queens Problem

The n-Queens problem is frequently used as a test-case for constraint programming algorithms. It consists in placing n non-attacking queens on a $n \times n$ chessboard. At least two variants of the problem exist: finding one valid configuration and enumerating *all* valid solutions. We use the version of n-Queens that consists in finding *all* valid solutions. N-Queens is easily modeled as a permutation problem: position i of a permutation of size n designates the column in which a queen is placed in row i. The encoding of a solution as a permutation of size n ensures that exactly n queens are placed on the board and that the "exactly-one" constraints on rows and columns are satisfied. In other words each n-element permutation represents a valid solution of a n-Rooks problem. Therefore, to evaluate the feasibility of a (partial) solution, it is enough to check for diagonal conflicts among the already placed queens. The board configuration (a permutation) is the only data structure needed for evaluating a node. For a partial solution of depth d, conflicts on the left and right diagonals with the d previous pieces need to be checked, i.e. $2d$ equality checks are performed. Thus, the evaluation of a node has complexity $O(n)$. In Figure 1.7 the n-Queens problem, modeled as a permutation problem, is illustrated. On the left-hand side (Figure 1.7a) a partial solution is shown. While there is no diagonal conflict in this subproblem, it is impossible to place a non-attacking queen in the next unoccupied row (row 6). Upon detecting this, the algorithm will backtrack to the last untried valid alternative, i. e. place the queen in row 5 in column 8. Figure 1.7b on the right-hand side shows a valid board configuration for $n = 8$.

It is easy to adapt B&B to solve this constraint satisfaction problem. Instead of searching for optimal solutions the goal is to find all valid solutions. Instead of a lower bound on the optimal cost of a subproblem it is enough to use a node evaluation function

that assigns the value 0 to feasible (partial) solutions and 1 to infeasible (partial) solutions. Initializing the algorithm at the upper bound 0 and pruning only in the case of strict inequality, the number of leaf nodes visited by B&B equals the number of valid board configurations.

For $n \leq 14$ the n-Queens problem can be solved within a fraction of a second by a sequential algorithm. The size of the explored tree grows exponentially, so we consider the n-Queens problem for $n = 15$–19. Permutations are only constructed from the beginning: because of symmetries the tree size cannot be reduced by constructing solutions from both ends. Several symmetries could be used to reduce the size of the explored tree. However, we only make use of one axial symmetry (i.e., restricting the queen in the first row to columns 1, 2 , ..., $\left\lfloor \frac{(n+1)}{2} \right\rfloor$), which divides the size of the search space by two.



(a) Feasible subproblem with no feasible children.

(b) Valid solution.

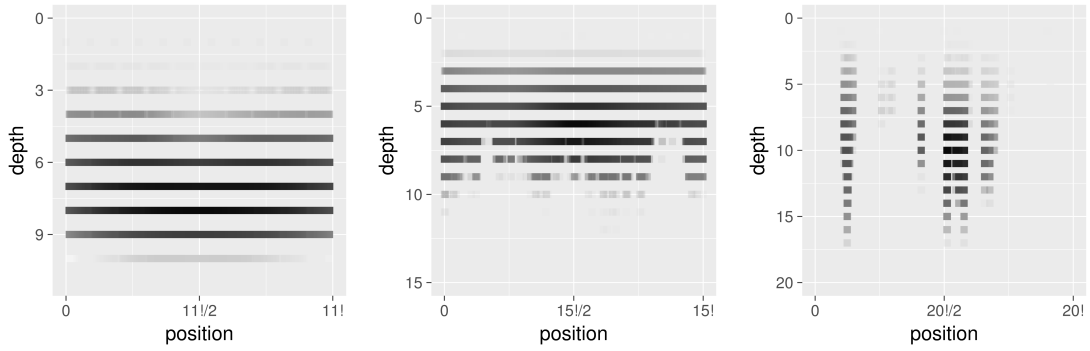
Figure 1.7: Illustration of the n-Queens problem for $n = 8$. On the left side of the board the permutation representing the board configuration is shown.

1.6.4 B&B tree analysis of the test problems

Preliminary experiments have been performed with a sequential B&B in order to illustrate the unpredictable, irregular nature of the workload and highlight differences between the three test-problems. Figure 1.8 illustrates the critical trees corresponding to the 11-Queens, *nug*15 and *Ta*020 instances. The trees have approximately the same number of nodes, despite very different problem sizes (11, 15 and 20 for 11-Queens, *nug*15 and *Ta*020 respectively). Each point in Figure 1.8 represents a decomposed node.

The x-Axis shows its relative position, among all possible subproblems arranged in increasing lexicographical order, and the y-Axis shows its depth. Each point is plotted with transparency, so the darker a node appears in the figure, the higher the density of critical nodes in its neighborhood.

The figure illustrates the efficiency of the pruning mechanism for the three problems. For the 11-Queens instance about $\frac{3}{1\,000}$ of all possible subproblems are visited, for *nug*15 only $\frac{1}{10^7}$ and for *Ta*20 about $\frac{1}{10^{15}}$. Of course, one can not extrapolate these numbers to all instances of the respective problem, but the behavior illustrates the strong problem-dependent variability of the workload generated by B&B. One can see in Figure 1.8a that for 11-Queens a relatively regular tree is developed. Indeed, some of the symmetries of the chessboard, which we chose not to exploit, can be observed in the structure of the tree. As reflected by the good results for parallel backtracking applied *n*-Queens [LLW+15a], a static repartition of active nodes at a certain depth may result in relatively low load balance.



(a) 11-Queens (151 810 nodes)

(b) QAP: *nug15* (174 201 nodes)

(c) FSP: *Ta020* (157 761 nodes)

Figure 1.8: Illustration of the critical trees associated with three problem instances of small size.

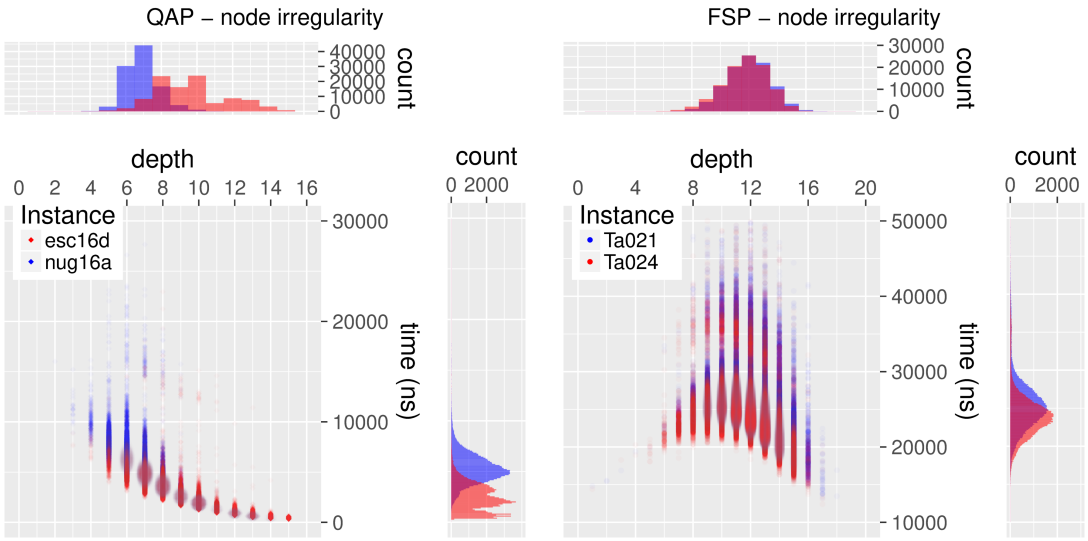
From Figures 1.8c and 1.8b it is clear that a static work repartitioning approach is likely to perform poorly for FSP and QAP problem instances. The FSP lower bounding mechanism allows to efficiently prune branches in the upper part of the tree, while a few nodes develop deep and large subtrees, concentrating most of the workload. For the selected QAP instance, the GLB lower bound is too weak for subproblems close to the root. Therefore, the upper part of the B&B tree is large, as the pruning operator only becomes effective at approximately mid-level of the tree. As mentioned, one must be very careful when generalizing to other instances of the same problem. However, the trees developed for other (small-sized) instances of the same problems are very similar.

As mentioned before, the duration of a node evaluation is a critical parameter in B&B. One cannot assume, in general, that each node evaluation requires an equal amount of time, even within the same tree. Using two QAP and two FSP instances, Figure 1.9 shows for 100 000 randomly sampled subproblems the time spent for computing a lower bound on their optimal cost. The two pairs of instances are selected because of the difference in node processing speed (elapsed time/processed nodes) that can be measured for sequential execution, despite equal problem sizes. For QAP instances of the *esc* class the measured node processing rates are 4 to 10 times higher than for *nug* instances of the same size. As shown in Table 1.1 the sequential resolution of *Ta024*, resp. *Ta021*, is performed at an average processing speed of 2069 n/s, resp. 1734 n/s.

In Figure 1.9 the time required for evaluating a node is plotted in function of its depth and marginal histograms show the distribution of nodes according to their depth and evaluation cost. For the two QAP instances, on the left-hand side (Figure 1.9a), one can observe that the cost for node evaluation decreases as nodes are closer to the leaves. This is due to the fact that the lower bounding procedure includes the resolution of a linear sum assignment problem whose size is equal to the number of unassigned facilities. As one can see in the histogram on the top, the majority of nodes in the critical tree of *nug16a* is found at depth 7, while for *esc16d* it is depth 10. Consequently, the average cost for evaluating a node is lower for *esc16d*. Moreover, one can observe that even the evaluation of nodes at the same depth requires more time for *nug16a*. The most likely explanation is that for the *nugent* instance, and the associated input flow and distance matrices the LSAPs are more difficult to solve but produce better lower bounds.

For the two FSP instances, on the right-hand side (Figure 1.9b), the distribution of nodes according to their depth is almost identical. However, as illustrated in the histogram on the right, the average time required for evaluating a node of *Ta024* is lower, which can explain the higher node processing rate achieved for this instance. For all four instances one can observe that the evaluation time for a node of a given depth is also subject to considerable fluctuation. In this example, the node evaluation cost for the FSP problems is on average more than five times higher than for QAP - the exact values depend on instance and problem size.

Figure 1.10 shows the results for the n-Queens problem, using instance size $n = 14, 17, 20$. The node distribution according to depth and evaluation cost is very similar for the three instance sizes. This indicates that the structure of the explored B&B tree is not substantially changing as its size grows exponentially according to n. One can notice that the average time spent for evaluating one node is about two orders of magnitude lower than for the two QAP instances. The node evaluation function returns as soon as



(a) QAP: *esc16d* and *nug16a*

(b) FSP: *Ta021* and *Ta024*

Figure 1.9: Illustration of node evaluation cost for two QAP instances of size $n = 16$ and two FSP instances of size $n = 20, m = 20$. B&B is initialized at optimal solution. Results are shown for 100 000 randomly sampled node evaluations.

the first diagonal conflict is detected, therefore the cost for this function is also variable.

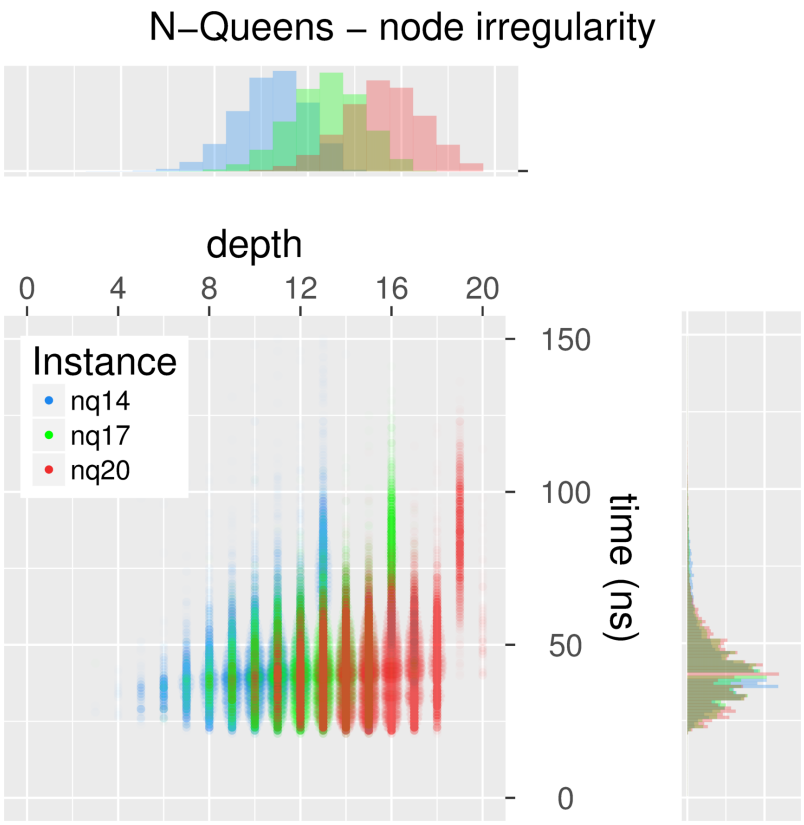


Figure 1.10: Illustration of node evaluation cost for n-Queens instances ($n = 14, 17, 20$). Results are shown for 100 000 randomly sampled node evaluations.

Chapter 2

IVM-based B&B for multi-/many-core systems

Contents

2.1 Introduction . . . 41

2.2 IVM-based parallel Branch-and-Bound . . . 41

2.2.1 IVM-based serial B&B . . . 41

2.2.2 Position vector: factoradic numbers . . . 44

2.2.3 Work units: intervals of factoradics . . . 46

2.2.4 Work unit communication . . . 48

2.3 Work stealing for IVM-based B&B on multi-core CPUs . . . 50

2.3.1 Work stealing using factoradic intervals . . . 51

2.3.2 Victim selection policies . . . 51

2.3.3 Granularity policies . . . 54

2.4 Accleration of bounding operator . . . 55

2.4.1 GPU acceleration . . . 55

2.4.2 Vectorization of the FSP bounding procedure . . . 57

2.5 Experiments . . . 59

2.5.1 Evaluation of data structures for B&B . . . 60

2.5.2 GPU-acceleration of the bounding operator . . . 62

2.5.3 Evaluation of Work Stealing Strategies . . . 63

2.5.4 Performance evaluation on Intel Xeon Phi . . . 67

2.5.5 MC-B&B: performance on different multi-core CPUs 70
2.6 Conclusions 71

Related Publications

- Gmys Jan, Leroy Rudy, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel, "Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms" in *Concurrency & Computation : Practice & Experience*, 28, 18, 4463-4484 (2016), https://doi.org/10.1002/cpe.3771
- Melab Nouredine, Gmys Jan, Mezmaz Mohand, Tuyttens Daniel, "Multi-core versus many-core computing for many-task Branch-and-Bound applied to big optimization problems" in *Future Generation Computer Systems* (2017), https://doi.org/10.1016/j.future.2016.12.039

2.1 Introduction

The objective of this chapter is to present the design and implementation of the IVM-based B&B algorithm for multi-core and many-core systems (MC-B&B). A GPU-accelerated approach, GMC-B&B, is presented, as well as a revisited vectorized implementation of the FSP bounding procedure.

Section 2.2 provides a detailed description of the Integer-Vector-Matrix (IVM) data structure and the serial IVM-based B&B algorithm. In Section 2.3 the parallel MC-B&B, based on the parallel tree exploration model, is introduced. This section also introduces factoradic work units and presents interval-based work stealing strategies. Besides improving parallel efficiency, the greatest potential for increasing the performance of B&B lies in the acceleration of the bounding operation.

In Section 2.4 two approaches for accelerating node evaluation are proposed. On multi-core systems equipped with GPUs the bounding operator can be accelerated by offloading this operator to the GPU. This approach, including a work stealing mechanism for the GPU-accelerated MC-B&B (GMC-B&B), is presented in Subsection 2.4.1. Potential for acceleration also lies in the vector processing units (VPU) of modern multi-core CPUs. In order to make use of AVX and the 512-bit vector extensions of Intel's Many Integrated Core (MIC) architecture, the implementation of the bounding operator for FSP is revisited in Subsection 2.4.2.

One primary goal of this chapter is the validation of the IVM data structure for other permutation problems than FSP, the only problem to which IVM has been applied until now. The presented algorithm includes CPU- and GPU-versions of node evaluation function for FSP, QAP and n-Queens. In order to assess the utility of using IVM in multi-core B&B algorithms, experimental results from extensive testing with all three test-cases are reported in Section 2.5. Finally, in Section 2.6 conclusions from this chapter are drawn.

2.2 IVM-based parallel Branch-and-Bound

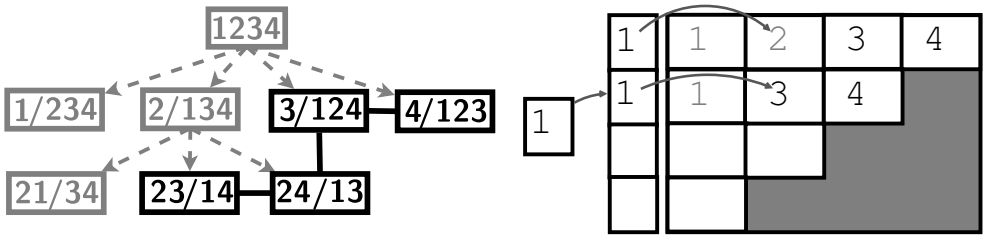
2.2.1 IVM-based serial B&B

The working of the IVM data structure is best explained with an example. Figure 2.1 illustrates a pool of subproblems that could be obtained when solving a permutation problem of size $n = 4$ with a DFS-B&B algorithm. On the left-hand side, Figure 2.1a shows a tree-based representation of this pool. The parent-child relationship between subproblems is designated by dashed gray arrows. As introduced in Subsection 1.3.1,

jobs before the "/" symbol are scheduled while the following jobs remain to be scheduled. Horizontal and vertical solid lines represent the linked-list data structure storing unexplored (solid black) nodes. On the right-hand side (Figure 2.1b) the corresponding IVM indicates the next subproblem to be solved.

In the example, the root node /1234 is decomposed into four subproblems by performing all possible assignments of unscheduled jobs to the first position. The example assumes that the first subproblem 1/234 has either been pruned or the branch has been fully explored. Therefore the second node, 2/134, was selected and decomposed, generating subproblems 21/34, 23/14 and 24/13. Again, the example assumes that the first of these subproblems has been completely processed. Therefore, the data structure contains at this point four unexplored nodes in the following order : 23/14, 24/13, 3/124, 4/123 This order corresponds to DFS with lexicographical ordering for nodes of the same depth.

It is unknown in advance which nodes are visited, but the relative order is completely determined by the ordered DFS search. Using this fact, IVM indicates the next subproblem to process without explicitly storing all generated pending nodes. The integer I of IVM gives the level (number of scheduled jobs -1) of this subproblem. The values of the so-called *position vector* V point to the jobs selected at each level. In other words, at each level (up to the current one) the vector V contains the index of the selected node among its sibling nodes. The matrix M contains the jobs that remain to be scheduled at each level: all n jobs (for a problem with n jobs) in the first row, $n-1$ jobs in the second row, and so on. In the example, the first row contains jobs 1, 2, 3, 4 and the second row contains jobs 1, 3 and 4 because job 2 is scheduled at the first level. The next subproblem to be solved can be decoded by reading from level 0 to I the jobs in the matrix M indicated by the vector V. Thus, the IVM shown in Figure 2.1b indicates 23/14 as the next node to process.



(a) Tree representation

(b) Integer-Vector-Matrix

Figure 2.1: Example of a permutation problem of size 4.

To use the IVM data structure in the B&B algorithm, some of the B&B operators are

revisited as follows:

- The **branching** operator consists in copying the jobs - except the selected one, pointed by V - from the current row into the next one. An illustration of the IVM-based branching operator is shown in Figure 2.3.

  In order to allow the scheduling of jobs at both extremities of the partial permutations an additional vector with binary values is used (not shown in Figure 2.3). This vector (called *direction vector*) indicates whether a selected job is placed in the beginning or the end of the partial schedule. For instance, if the IVM shown in the example of Figure 2.1 is completed with the direction vector 1000, then the current subproblem is 3/14/2. Moreover it should be noted that the jobs in each row can be stored in any order. For instance, the jobs in each row could be ordered according to the lower bounds of the corresponding subproblems. For the sake of simplicity, increasing lexicographic order is used in the previous example.

- **Pruned** nodes should be ignored by the selection operator. In order to flag a cell as "pruned" its value is multiplied by -1. With this convention the branching procedure actually consists in copying the absolute values to the next row, i.e. copying both $-j$ and j as j.

- To **select** a subproblem the values of I and V are set accordingly. Depth-first search is implemented as follows: vector V is incremented at position I until a non-pruned cell is found or the end of the row is reached. If the end of the row is reached (i. e. $V[I] = n-I$), then the algorithm "backtracks" to the previous level by decrementing I and again incrementing V. Figure 2.2 shows an illustration of the IVM-based DFS selection operator and Algorithm 1 provides the pseudo-code of a procedure implementing this operator.

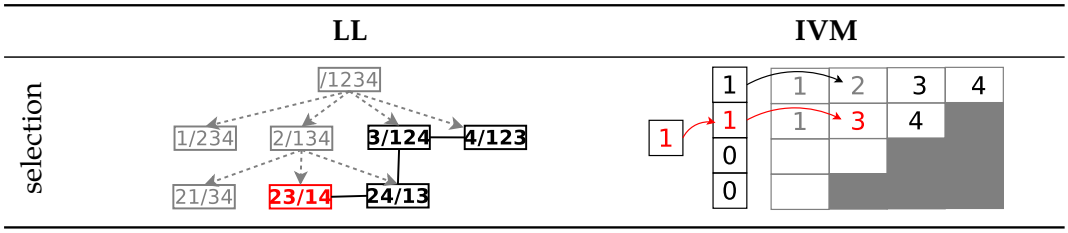


Figure 2.2: Illustration of IVM-based selection operator.

Each cell of IVM represents one node of the B&B tree. Therefore, the compact form of IVM reduces the worst-case memory requirements for the storage of the work pool by a factor n.

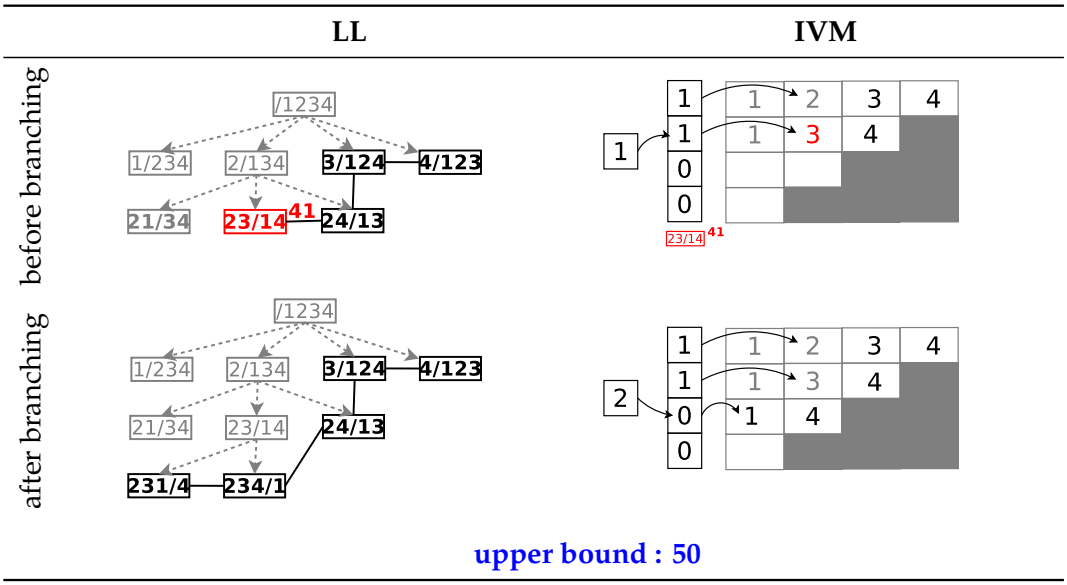


Figure 2.3: Illustration of IVM-based branching operator.

It should be emphasized that IVM can only be used for depth-first type search strategies. Indeed, the design of IVM assumes that any subproblem is completely explored before another subproblem on the same level is processed, which corresponds to the definition of depth-first search. However, IVM imposes no order restriction for jobs on the same level.

The sequential IVM-based B&B algorithm terminates when an attempt is made to move right after the last cell of the first row, in other words, when $V[0] = n$. Equivalently, this termination condition can be expressed by comparing the value of the position vector to a maximal allowed value, represented by a vector of length n, referred to as the *end-vector*. In Algorithm 1 this termination condition is included in Line 3.

2.2.2 Position vector: factoradic numbers

When running a B&B algorithm, the values of the vector V are continuously updated. As described in the selection procedure (Algorithm 1), when the end of row is reached the algorithm backtracks to the previous level. Therefore, at level $I = 0, 1, \ldots, n-1$ the value of V is bounded by $V[I] < n - I$. In the example of Figure 2.1b, the vector is equal to 0000 when the algorithm points to the first solution of the B&B tree, and equal to 3210 when it points to the last solution. Between these values, the vector successively takes the values 0010, 0100, 0110, 0200, 0210, ... , 3210. For each of these values, the algorithm points to a different solution. There are 24 possible values because there are 24 solutions (i.e. 4!).

Algorithm 1 Serial select-and-branch

```
 1: kernel SELECT-AND-BRANCH
 2:     state←EMPTY
 3:     while (positionVector ≤ endVector) do          ▷ termination condition
 4:         if (row-end) then                          ▷ (V[I] > I)?
 5:             cell-upward                            ▷ I − −; V[I] + +
 6:         else if (cell-eliminate) then              ▷ M[I, V[I]] < 0?
 7:             cell-rightward                         ▷ V[I] + +
 8:         else
 9:             state←EXPLORING
10:             break
11:         end if
12:     end while
13:     if (state=EXPLORING) then
14:         generate-next-line                         ▷ branch
15:     end if
16: end kernel
```

These 24 position-vector values correspond to the numbering of the 24 solutions using a special numbering system, called factorial number system [Knu97]. The factorial number system, also called factoradic, is a mixed radix numeral system adapted to numbering permutations. Applied to the numbering of permutations the French term *numération factorielle* is used in 1888 [Lai88].

In the factorial number system, the weight of the i^{th} position is equal to $i!$, contrary to the decimal number system, where the weight of the i^{th} position is equal to 10^i. While in the decimal number system, the highest digit allowed for each position is 9, in the factorial number system, the highest digit allowed for the i^{th} position is equal to i. Therefore, the digit of the first position, reading from right to left, is always 0.

The factorial number system satisfies the conditions of what Cantor called a simple number system (*einfaches Zahlensystem*) in [Can69], i.e. a number system in which each positive integer has a unique representation. A sufficient condition for such a system is the following [Can69]: for all $i = 0, 1, 2, ...$, the weight of the i^{th} position w_i must divide the next weight w_{i+1} without remainder and the highest digit allowed at the i^{th} position λ_i is equal to $\frac{w_{i+1}}{w_i} - 1$.

In particular, the set of n-digit factorial numbers can be put in bijection with the subset $[0, n![:= 0, 1, ..., n! - 1$. A n-digit factoradic number $a^{(!)}$ is transformed to its decimal form $a^{(10)}$ as follows:

$$a^{(10)} = \sum_{i=0}^{n-1} a_i^{(!)} \times i!$$

The conversion of a decimal number to its factorial form is obtained by performing successive euclidean divisions.

2.2.3 Work units: intervals of factoradics

As mentioned, in the parallel tree exploration model several independent B&B processes explore different parts of the B&B tree in parallel. Because of the highly irregular and unpredictable shape of the tree the latter tree should be dynamically distributed among B&B processes at runtime. To achieve this we have revisited the work stealing paradigm for IVM-based B&B. The challenge here is twofold:

- Defining work units and the way they are communicated (this subsection and Subsection 2.2.4)
- Defining victim selection and granularity policies to manage the stealing operations performed on these work units. (Section 2.3)

Based on the properties of the position-vector V we say that a serial B&B algorithm applied to a permutation problem of size n **explores the interval** $[0, n![$. In the parallel version of B&B, it is possible to have two B&B processes T_1 and T_2 that explore intervals $[0, x[$ and $[x, n![$ respectively. As the repartition of work (nodes to decompose) in these intervals (work units) is unpredictable and irregular, it is impossible to determine a number x *a priori* such that both processes finish the exploration at the same time. If T_1 finishes exploring its interval before T_2 (meaning that $V_1 = x$), then T_1 attempts to steal a portion of T_2's remaining interval. In order to achieve load balance, T_1 and T_2 can exchange intervals until the entire interval $[0, n![$ is explored. This approach can be generalized to an arbitrary number of B&B processes: In the IVM-based *work stealing* approach, when a B&B process has an empty interval it becomes a *thief* and attempts to steal a portion of the interval from another B&B process - called the *victim*.

The principle of IVM-based parallel B&B can be summarized as follows:

- The resolution of a problem of size n corresponds to the exploration of the interval $[0, n![$, representing the search space.
- Multiple independent B&B processes explore the interval $[0, n![$ in parallel. The interval $[0, n![$ is partitioned into smaller, mutually disjoint intervals which are assigned to independent B&B processes; Each of these B&B processes uses its private IVM data structure to explore an interval $[A, B[\subset [0, n![$.
- Work units exchanged between IVM-based B&B processes are intervals. These intervals can be equivalently expressed in factoradic or in decimal form. In factoradic form, the extremities of an interval $[A, B[$ represent the starting position vector

(i. e., IVM is initialized at $V = A$) and the end vector (i. e. the work unit is *empty* when $V = B$).

- There is a one-to-one correspondence between a B&B process and the IVM structure it uses to explore exactly one interval $[A, B[$. When describing interaction between B&B processes we will sometimes use the term "IVM" to designate, by extension, the B&B process which uses it. For instance, we say that "IVM k steals a portion of IVM j's interval" confounding IVM-based B&B processes and the private data structures used by these processes.

Conventionally, in a LL-based B&B work units exchanged between threads (or processes) are sets of nodes. In such approaches, only nodes that were generated by the branching operator are exchanged. Using interval-based encoding for work units, the load balancing mechanism divides intervals units and distributes them among B&B processes. While intervals correspond to sets of nodes, and *vice-versa*, an important difference between both types of work units should be noted: In contrast to LL-based approaches, in the IVM-based approach work units are not generated by the B&B exploration processes but only by the load balancing mechanism.

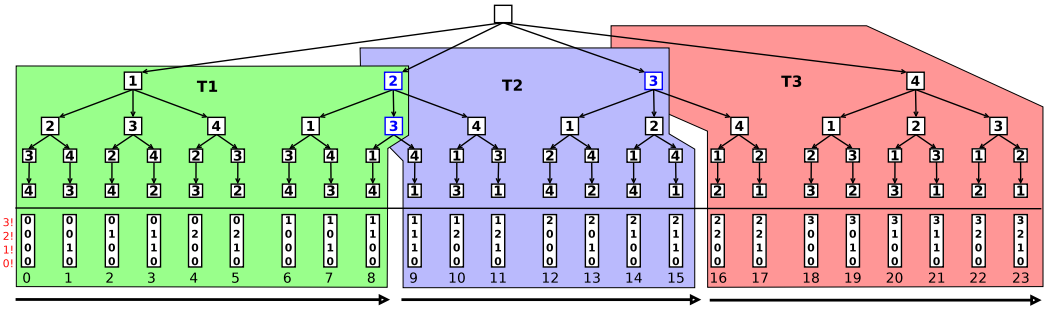


Figure 2.4: Illustration of parallel tree exploration using three B&B processes and interval-based work units.

Figure 2.4 illustrates the repartition of the full search tree for a problem of size $n = 4$ among three B&B processes. The interval $[0, 4![$ is split into three parts. Each of these parts is explored independently by one B&B process using its private IVM structure.

As mentioned, when a B&B process has finished the exploration of its interval it attempts to steal a portion of another interval.

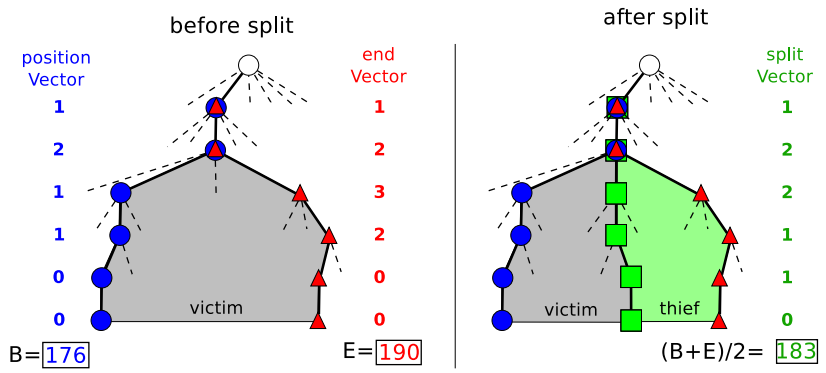
The communication of work units between B&B processes requires the two following procedures:

- a procedure for splitting an interval $[A, B[$ into two parts, $[A, C[$ and $[C, B[$,

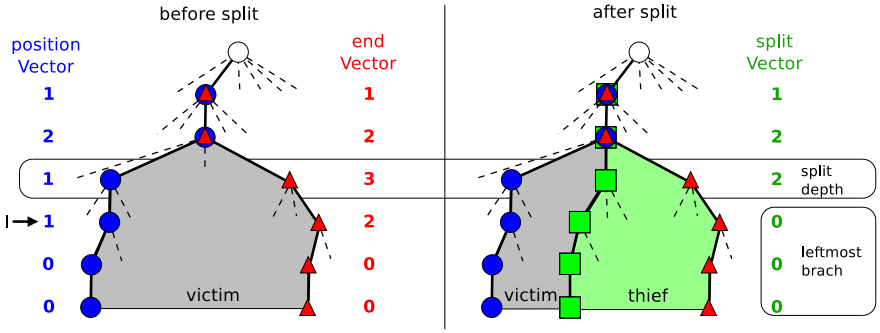
- and a procedure to initialize an IVM structure from a position vector $V = C$.

2.2.4 Work unit communication

In this subsection two alternative procedures for the communication of work units between IVM-based B&B processes are described. An illustration of both procedures is shown in Figure 2.5.



(a) Arithmetic interval splitting



(b) Subtree-based interval splitting

Figure 2.5: Illustration of work unit splitting.

Work unit splitting at an arbitrary position: An integer interval $[B, E[$ can be split at any integer C that is convex linear combination of B and E, i. e. $C = \lfloor (1-\alpha)A + \alpha B \rfloor$, $0 < \alpha < 1$. The computation of a splitting point C can be performed either by using decimal arithmetic operations or by implementing elementary arithmetic operations for factoradic numbers. The granularity of this procedure is controlled by the value of α. In the example shown in Figure 2.5a, the interval of the victim is split in two parts of equal size, setting $\alpha = 0.5$.

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

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

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

Subtree-based work unit splitting. In [Ler15] a method for exchanging work units between IVM-based B&B processes without initialization is proposed. This procedure is based on splitting the interval of the victim IVM directly in its factoradic form, without converting it to decimals. In addition to the new position and end vectors, an initialized matrix is transferred from the victim IVM to the thief IVM. Figure 2.5b illustrates this procedure. The transfer of a work unit from IVM S (Sender) to IVM R (Receiver) can be performed as follows.

1. Let $[B_S, E_S[$ be the interval to split. Let l be the smallest index such that $B_S[l] \neq E_S[l]$.

2. For $i = 0, 1, \ldots, l-1$, copy row i of IVM from S to R (position, end, direction vectors and matrix).

3. Split tree at level l by choosing C such that $B_S[l] < C \leq E_S[l]$. For the receiving IVM R, set $B_R[l] = C$ and $E_R[l] = E_S[l]$. For the sending IVM S, $E_S[l] = C - 1$.

4. For $i = l + 1, \ldots, n - 1$, set $E_S[i] = n - i - 1$ and $B_R[i] = 0$.

5. The receiving IVM R is now initialized and exploration can start at level l. Therefore I_R is set to $I_R = l$.

This initialization method proposed by Leroy [Ler15] requires no additional computation of bounds. Therefore it reduces the overhead induced by work stealing operations, compared to the previously introduced initialization procedure. Another important advantage of this method is that it avoids redundant computations. Indeed, if intervals are split at arbitrary points, some subproblems may be decomposed redundantly, as illustrated by the overlapping tree portions in Figure 2.4. In this example, thread T1 explores interval $[0000, 1100[$ and thread T2 explores $[1110, 2110[$, which may cause redundant computation of bounds along the frontier $11XX$.

However, there are also disadvantages. Especially in a distributed memory setting the size of data transfers should be kept low. Indeed, this is the primary motivation for using interval-encoded work units [MMT07a]. Also, in the subtree-based interval splitting, the granularity is controlled in a coarser way because *full* subtrees are communicated. This method is similar to stack-splitting strategies where nodes (and implicitly the subtrees rooted in these nodes) are transferred from non-empty to empty pools. Dividing intervals by the first method allows a finer control of granularity.

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

2.3 Work stealing for IVM-based B&B on multi-core CPUs

There are mainly two ways to increase the efficiency of a given parallel B&B algorithm. The first is to improve the distribution of work among processing units (achieve good load balance) and the second is to improve the usage of each individual processing unit. Load balancing aims at maximizing the benefits of parallelizing the exploration of the B&B tree by reducing idle time. Ideally, all processing units are kept busy without additional overhead, solving a given instance P times faster when using P identical processing units instead of one.

The second factor is in fact a sequential optimization of the B&B algorithm. As shown in Section 1.6.4, for typical instances of the three considered test cases, the bounding operator is called millions of times. Moreover, for the FSP and QAP problems it is by far the most time consuming part of the algorithm, consuming up to 99% of sequential execution time in the case of FSP.

In substance, both these objectives can be pursued independently, even though faster node evaluation can make load balancing more difficult, as it becomes harder to hide com-

munication overhead. In the following, we first present our load balancing approach for IVM-based multi-core and many-core B&B. Then, in Section 2.4, we present approaches aiming at the acceleration of the bounding operator, using GPUs as accelerators and leveraging vector processing capabilities of multi-core CPUs.

2.3.1 Work stealing using factoradic intervals

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

2.3.2 Victim selection policies

In this Subsection, four victim selection policies are described. As pseudo-code for these policies is shown in Algorithm 2. Two of them, the *random* and *ring* policies, have a low computational complexity and require no access to shared data structures or knowledge about the global workload repartition. The two other selection policies, namely the *largest* and the *honest* policies, use simple heuristics which aim at selecting a stealing victim holding a large or difficult piece of work. These policies use the available information about the workers' activity and require some additional computation as well as protected accesses to shared data structures.

- **Ring victim selection policy**: In this deterministic policy, threads are connected to each other with an unidirectional ring. A thread R always steals 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 threads. In this policy, the work stealing attempt of a thread R is a blocking event when thread R' has no work. In this case, the work stealing request will be satisfied when the thread R' will receive work. This policy is also used, for instance, in [KRR88]. If the thread numbering is matched with the underlying architecture the deterministic nature of this policy can be used to reduce communication costs. As shown in function *choose-ring* (Algorithm 2, Line 12), the cost of the victim selection function is very low. No locking or access to shared

Algorithm 2 Pseudocode of the victim selection policies for MC-B&B.

```
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(R)
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
```

data structures is required for this selection strategy: the only information an idle worker needs to select a victim is its own thread/process-ID. The main issue of this strategy is that work units may not propagate fast enough through the ring.

- **Random victim selection policy**: The *random* selection policy is provably efficient [BL99] and the most frequently used in the literature. In this policy, a victim thread R' is randomly selected when a thread R initiates a work stealing operation. Unlike the ring policy, this work stealing operation is not a blocking event. In other words, the thread R continues to choose other threads randomly until it finds a thread with a non-empty interval or linked-list. The state variable of the randomly selected victim, indicating whether work is available, should be accessed atomically.

- **Largest victim selection policy**: In a B&B algorithm, it is often impossible to determine the "hardness" of a work item. This policy is based on a simple heuristic to choose the thread with the most difficult work to finish. Indeed, the largest policy assumes that probably the larger the size of a work is, the more difficult this work will be. Therefore, this policy computes the amount of work of each thread, chooses the thread with the biggest size, and returns the rank R' of this thread. In the linked-list-based approach, the size of a linked-list is equal to the number of nodes it contains, and in the interval-based approach, the size of an interval $[A, B[$ is equal to $B - A$. As shown in function *choose-largest* (Algorithm 2, Line 27), this policy has a higher computational complexity than the three other victim selection policies. In particular a thief requires locked accesses the *length* or *size* variable of each busy worker. Moreover, each thread periodically (and atomically) updates this quantity. Although this polling may compromise the scalability of this strategy, good results for this policy are reported in [ASW+14].

- **Honest victim selection policy**: This strategy is based on another heuristic to determine the thread with the most difficult work to finish. The heuristic assumes that if a thread R_1 has stolen work less recently than a thread R_2, then the thread R_1 has probably a work which is more difficult than the work of the thread R_2. Therefore, the thread R steals the work from the thread victim R' which is the least recent thief. As shown in function *choose-honest* (Algorithm 2, Line 27), this policy has a higher computational complexity than the ring and random policies but a smaller computational complexity than the largest policy. In the largest victim selection policy, it is important to compute the amount of work of each pool and computing the size of any pool is a blocking operation for the thread which

owns this pool. In the honest victim selection policy, one operation of removing is performed on the *rank-threads* list, and this operation is a non-blocking. The operations on the global *rank-threads* list must be protected by locks.

2.3.3 Granularity policies

When a thread R' is contacted by a thread R, the thread R must determine the amount and which part of work to steal from its victim thread R'.

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

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

2.4 Accleration of bounding operator

2.4.1 GPU acceleration

One way to accelerate a multi-core B&B algorithm is to offload the bounding operation to one or several many-core accelerators, like GPUs. This approach is promising under the condition that the node evaluation fits the accelerators execution mode. As the offloading requires data preparation and transfers, the bounding operation should be substantially accelerated in order to compensate for the incurred overhead. The accelerator can be used to parallelize the lower bound function itself or to evaluate several generated subproblems in parallel. We choose the parallel evaluation of bounds model which is more generic and can, in principle, be applied to any node evaluation function. This introduction of a second level of parallelism concerns only the bounding operator, which is performed in three phases.

The first phase consists in copying the generated subproblems to the device. Secondly, a kernel function is launched which performs the evaluation of the subproblems on the device. The host thread needs to wait for the completion of this kernel since it produces lower bounds which are needed for the pruning operation. In order to avoid breaking the asynchronous execution of the parallel tree exploration, the CPU threads should be able to perform these operations concurrently. The number of CPU cores is usually higher than the number of GPUs available, so GPU devices are shared by multiple threads. Fortunately, modern GPUs support multi-threaded kernel offloading. For instance, nVidia's Hyper-Q technology[1] allows the concurrent launching of up to 32 kernels from different CPU-threads (depending on multiprocessor availability).

In order to accelerate the bounding operation effectively, the number of offloaded subproblems must be sufficiently high. However, the quantity of subproblems generated by one IVM may not yield an acceptable GPU-occupancy, unless the computation of a bound is also parallelized. To solve this issue, the number of B&B processes handled per thread may be increased, partially violating the paradigm of the parallel tree exploration model. In that case, instead of handling a single B&B process, each thread applies the branching, selection and pruning operators sequentially to M pools. This approach is schematically represented in Figure 2.6. The parameter M needs to be adjusted in function of the average number of subproblems generated per pool, the hardware constraints of the accelerating device and the relative cost of the bounding operator with respect to the rest of the algorithm. A good value for parameter M should be determined experimentally.

1. Introduced with CUDA compute capability 3.5; in this thesis we consider NVIDIA's CUDA programming toolkit, but of the concepts and conclusions should remain valid for OpenCL.

It also becomes necessary to equilibrate the work load among the M pools handled by the same thread. An intra-thread work redistribution phase must therefore be added to the algorithm and the work stealing strategies need to be adapted to this. With respect to the work stealing strategies defined in the previous subsection the pools belonging to the same thread behave like a single "super-pool", meaning that a thread attempts to steal work from another thread only if all its local work is exhausted. During a work stealing operation pool number k of the thief thread steals from pool number k of the victim thread according to the granularity policy. Using this convention the work stealing for $M > 1$ is consistent with the work stealing for $M = 1$.

In order to ensure work load balancing among pools belonging to the same thread a work repartitioning phase is introduced. In this phase work stealing is mimicked inside a thread, sequentially. For this intra-thread load balancing the *largest* policy with steal-half granularity is used. This strategy seems the most appropriate in this context: the *honest* policy makes no sense in a sequential context, the *ring* policy may favor pools with numbers within a certain range. As the pool sizes are locally available it also seems unnatural to reduce the cost of victim selection by choosing randomly.

The adapted work stealing for the hybrid CPU-GPU B&B is summed up by the following rules:

- **Rule 1** For intra-thread work stealing, only the largest policy with steal half granularity is used.
- **Rule 2** A work stealing operation between a thread R and a selected victim thread R' is defined as follows: the k^{th} pool of thread R tries to steal work from the k^{th} pool of thread R' according to the defined granularity policy.
- **Rule 3** A hierarchical work stealing approach is applied: Intra-thread has priority over inter-thread load balancing. Therefore a thread R attempts to steal a thread R' only if all of its M pools are empty.
- **Rule 4** The steal Tth granularity policy applies only to the ring topology.

With these additional rules the work stealing strategies for MC-B&B, described in Subsections 2.3.2-2.3.3 also apply to the hybrid GPU-accelerated algorithm. For instance, using work stealing strategy *random*-1/2 in the hybrid CPU-GPU-algorithm means: when all pools inside a thread R are empty, thread R attempts to steal from a randomly chosen thread R' (Rule 3). If R' has non-empty pools $k_i^{R'}$, half of the work in those pools is transferred to pools k_i^R of thread R (Rule 2). Moreover each thread balances the work load among its M pools internally using *largest*-1/2 strategy.

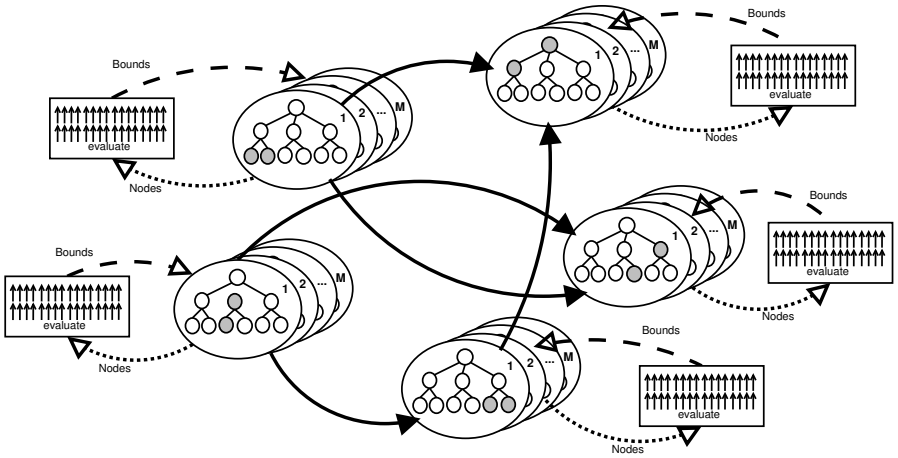


Figure 2.6: Illustration of the GPU-accelerated multi-core B&B (GMC-B&B).

2.4.2 Vectorization of the FSP bounding procedure

Another opportunity for accelerating multi-core B&B algorithms are the vector processing extensions available in most modern CPUs. There are different approaches for achieving vectorization, depending on the structure of the targeted portion, ie the node evaluation function. If the appropriate flags are passed to the compiler it will attempt to auto-vectorize the code. This may succeed if the node evaluation function is more or less trivially vectorizable. At a lower level, the programmer may insert pragmas, giving hints to the compiler and at an even lower level explicit vector instructions may be used to rewrite the node evaluation function.

We focus on the lower bounding function of the FSP. In particular, we target the most compute-intensive portion of the lower bound function and its main data-dependencies. The pseudo-code for the lower bound function, is given in Algorithm 3.

This compute-intensive portion of code is the inner for-loop (Algorithm 3, Lines 7-13) which consumes about 70% of the bounding time. The body of this inner loop is executed $\frac{n^2 \times (n-1)}{2}$ times, n being the number of jobs. Regarding data dependencies, the statement in Line 11, including a dependency of current *tmp*1 on *tmp*1 from previous iterations prevents vectorization (the Intel compiler *icc* does not auto-vectorize it and the vectorization report signals this dependency). In addition, except for Line 15 the iterations of the outer loop are independent (private variables: *tmp0, tmp1, ma0, ma1*). However, only the inner loop may be vectorized [2].

In order to use the vector processing abilities for this portion of the code the order of

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

Algorithm 3 Computation of lower bound (un-vectorized)
input: subproblem = {permutation, nbFixed (#jobs fixed)}, constant data (MM, JM, PTM, LM)
output: lower bound (LB) of subproblem

```
1:  n := #jobs
2:  function COMPUTE LB
3:      RM, QM, SM ← InitTabs(permutation, nbFixed)
4:      LB ← 0
5:      for (k = 0 → N(N−1)/2) do
6:          tmp0, tmp1, ma0, ma1 ← InitFun(k, nbFixed, MM, RM)
7:          for (j = 0 → n) do                                  ▷ ∼ 70% of time
8:              job←JM[k][j]
9:              if (SM[job]==0) then
10:                 tmp0 += PTM[ma0][job]
11:                 tmp1 = max(tmp1, tmp0 + LM[k][job]) + PTM[ma1][job]
12:             end if
13:         end for
14:         tmp1←EndFun(tmp0, tmp1, k, nbFixed, QM)
15:         LB = max(tmp1, LB)
16:     end for
17:     return LB
18: end function
```

the nested loops must be inverted. The vectorized lower bound function is illustrated in Algorithm 4.

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 21) in order to isolate the k-dependent instructions from the inner-loop. The cost to pay for this is to declare the scalars $tmp0$, $tmp1$, $ma0$ and $ma1$ as arrays (resp. $Tmp0$, $Tmp1$, $Ma0$ and $Ma1$) of size $\frac{N(N-1)}{2}$. In an environment with small last-level caches and/or few registers this strategy can therefore be highly detrimental to performance. We have indeed implemented this vectorized version of the bounding function on the GPU and observed severe performance degradation (these intermediate variables are no longer stored into registers).

In order to improve performance and assist the compiler in vectorizing the loops all arrays are aligned at 64 byte boundaries. For static arrays this is achieved by using `__attribute__((aligned(64)))`. For dynamically allocations the `_mm_malloc` function is used and the statement `__assume_aligned(arr, 64)` informs the compiler immediately before the concerned loop that the starting address of array `arr` is a multiple of 64 bytes. Even with the highest optimization level activated ($-O3$) the Intel compiler (*icc*) still needs the hint "#pragma ivdep" to vectorize the inner loop (Line 10) successfully. The reason for this is the conditional "if"-statement which causes diverging execution flows between SIMD lanes. The two other for-loops are auto-vectorized.

Algorithm 4 Computation of lower bound (vectorized)
input: subproblem = {permutation, nbFixed (#jobs fixed)}, constant data (MM, JM, PTM, LM)
output: lower bound (LB) of subproblem

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

2.5 Experiments

In this section we report the experimental results for the algorithms presented in this chapter. All experiments are run on a computer composed of two 8-core Haswell E5-2630v3 processors, and four GeForce GTX 980 GPUs. Unless specified otherwise, the compiler `gcc 5.4` is used with optimization level $-O2$ and version 7.5 of the CUDA toolkit is used.

For all runs the initial upper bound is set to the optimal cost, in order to evaluate the performance of the algorithm in the absence of speedup anomalies. Indeed such an initialization ensures that B&B explores exactly the critical tree in order to prove the optimality of the initial upper bound.

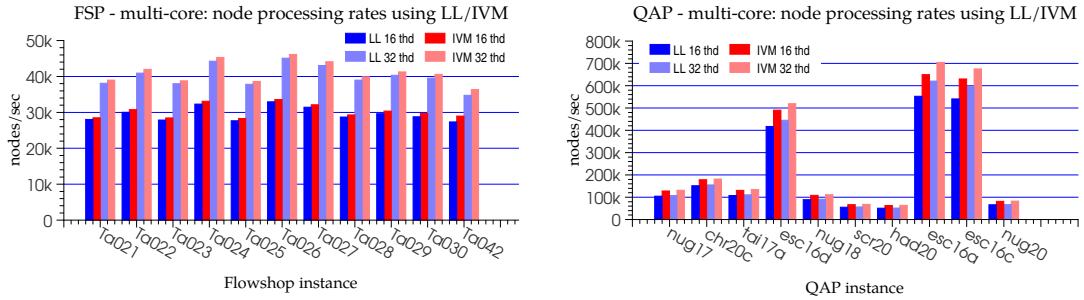
The execution times for different problem instances vary strongly because of varying tree sizes and node evaluation costs. In order to present experimental results for instances of different size in a comparable manner, most results are reported in terms of achieved average node processing speed, defined as the rate of *decomposed nodes per second*. We recall that the units n/s, kn/s and Mn/s are used to designate respectively 1, 10^3 and 10^6

decomposed nodes per second.

2.5.1 Evaluation of data structures for B&B

Figure 2.7 shows the number of nodes decomposed per second for the multi-core LL- and IVM-based B&B algorithm without bounding acceleration. The node processing rate is computed by dividing the number of nodes decomposed by the elapsed walltime for completing the exploration of the tree. Using this metric, rather than raw computation time, allows a better comparison of performance achieved for instances of different size. The reader interested in the actual resolution time can divide the number of explored nodes (given in Appendix A.1) by this rate.

For example, the critical tree of *Ta*028 is composed of 8 088 505 nodes. Using 32 threads, the IVM-based MC-B&B decomposes ≈40 kn/s, so the elapsed walltime for exploring this tree is ≈ 200 seconds (instead of 4 440 seconds or 1.8 kn/s for its sequential counterpart).



(a) Flowshop (b) QAP

Figure 2.7: Node processing rates for IVM- and LL-based multi-core B&B solving FSP and QAP instances using 2 Intel Xeon E5-2630v3 CPUs.

Comparing the achieved node processing rates for different FSP instances one can notice that similar rates are achieved for instances *Ta021-Ta030*. The node processing rates achieved for FSP instances of size 20 × 20 vary between 27–33 kn/s for 16 threads and 37–45 kn/s for 32 threads (using hyperthreading capabilities). Variations between instances are due to different tree shapes and variable node evaluation costs, as shown by the analysis of the sequential algorithm in Subsection 1.6.4. Comparing LL- and IVM-based implementations for FSP one can notice that the IVM-based B&B outperforms its LL-based counterpart only by an insignificant margin.

In contrast to FSP, for the QAP the node processing rate varies strongly depending on the instance being solved. As for the FSP, these variations are not due to the size of

the explored tree (in Figure 2.7b the QAP instances on the x-axis are sorted in increasing order according to the size of the explored critical tree). Instead, it appears the instance class has a strong influence on the processing speed.

For instance, solving the *nug* instances *nug17-20*, node processing rates around 100 kn/s are measured, while the *esc* instances are solved with processing rates up to 7 times higher. This is due to the nature of the GLB bound which is used by the bounding operator for the QAP. The computationally intensive part of this bounding procedure consists in solving a Linear Assignment Problem, using an implementation of the Kuhn-Munkres algorithm with $O(n^3)$ worst-case complexity. Depending on the coefficient matrix, and thus on the input flow and distance matrices, this step requires a variable amount of computation.

These variations allow one to make the following observation. For instances where the node evaluation function has a lower cost, the relative speedup of the IVM-based B&B over its LL-based counterpart is higher. This can be explained by the fact that the relative importance of the pool management is higher. As IVM allows a more efficient handing of the pool of subproblems, the effect on the total execution time becomes more noticeable.

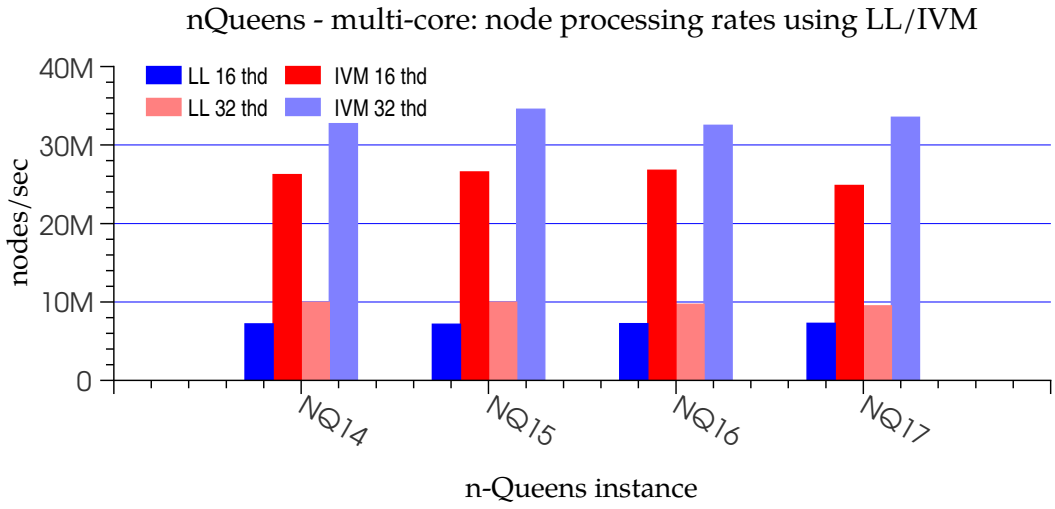


Figure 2.8: Node processing rates for IVM- and LL-based multi-core B&B solving n-Queens instances using 2 Intel Xeon E5-2630v3.

The node processing rates for n-Queens problems with n = 14-17 are shown in Figure 2.8. For this problem, the IVM-based algorithm spends less than 1/3 of the total execution time for the evaluation of subproblems. Therefore, the management of the work pool, including selection, branching and pruning of subproblem, is critical. As one

can see in Figure 2.8 the IVM-based algorithm outperforms its LL-based counterpart by at least a factor 3×.

2.5.2 GPU-acceleration of the bounding operator

The GPU-accelerated multi-core algorithm presented in Subsection 2.4 requires tuning of the number of B&B processes handled per CPU thread (M), i.e. the size of the offloaded pool. An ad-hoc value can be determined as follows. For 32 CPU-threads and 4 available GPUs (GTX980) the maximum number of resident GPU-threads in the system is

$$4 \text{ GPU} \times 16 \text{ SM/GPU} \times 2048 \text{ threads/SM} = 131\,072 \text{ threads}.$$

Supposing that n nodes are evaluated per average node decomposition [3], for a 20-job instance, setting $M \geq \frac{131\,072}{20\times32} \approx 200$ should be a good configuration for M. Figure 2.9 shows the elapsed walltime for solving instance *Ta028* using the LL and IVM-based algorithms with different values for M. According to this figure, for all following experiments using GMC-B&B, the number of IVM (resp. LL) per thread is set to $M = 600$.

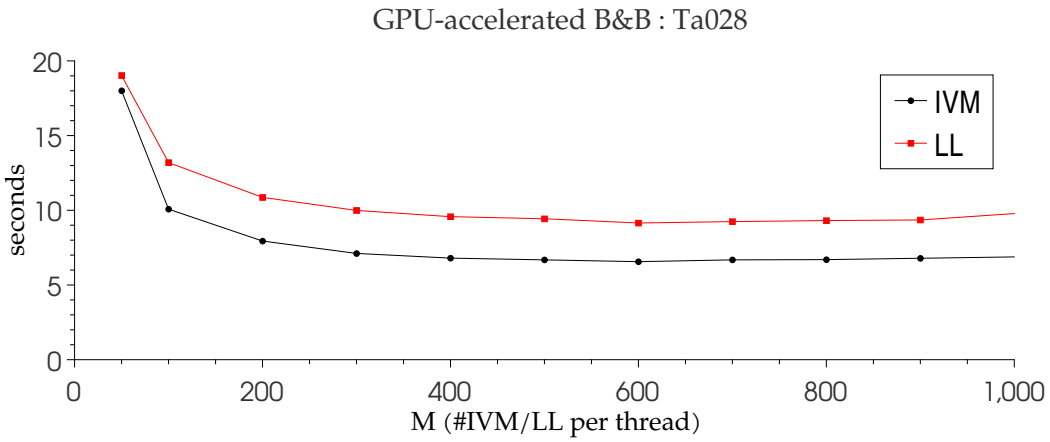


Figure 2.9: Calibration of parameter M (B&B processes handled by each thread. FSP instance: *Ta*028, 32 threads, 4 GPUs, *random*-1/2 (IVM) and *random*-1 (LL) work stealing strategies.

In Figure 2.10 the performance of the IVM-based GMC-B&B is evaluated and compared to its LL-based counterpart. On the x-Axis, Figure 2.10 shows the node processing rates achieved by the IVM-based GMC-B&B for FSP instances *Ta021-Ta030* and several QAP instances. On the y-Axis it shows the ratio between the execution time of the

3. The analysis of the sequential algorithm in Subsection 1.6.4 shows that this is a realistic estimate for FSP, if the generation of two candidate children sets `begin` and `end` is taken into account.

IVM-based algorithm and its LL-based counterpart (T_{LL}/T_{IVM}, which is equivalent to the ratio between node processing rates). In order to improve the readability of Figure 2.10 it contains an inlet, zooming on the lower left part of the Figure.

One can make several observations. First, one can notice a correlation between the node processing rate and the ratio T_{LL}/T_{IVM}. As previously shown by comparing the non-accelerated MC-B&B for FSP, QAP and n-Queens, this ratio increases with the node processing rate. Again, this is due to the fact that the algorithm spends less time in the bounding operation and more in managing the pool of subproblems.

For the FSP, node processing rates between 0.9 and 1.5 Mn/s are attained. This is 22-37× higher than the average attained by the 32-threaded MC-B&B. For the FSP, the IVM-based algorithm is on average 1.37× faster than its LL-based counterpart.

One can also notice, that the acceleration for instance *Ta030* is less efficient than for the other FSP instances. While the multi-core processing speed for *Ta030* is equivalent to that of other instances (see Figure 2.7a), it is about 1.5× lower for the GPU-accelerated algorithm. This is due to the relatively small size of the explored B&B-tree (1.6×10^6 nodes). The resolution of *Ta030* lasts about 15 minutes using a sequential algorithm and 41 seconds using 32 threads on 16 CPU-cores for a speedup of almost 22×. Using the GPU-accelerated multi-core B&B, *Ta030* is solved in 1.8 seconds, i. e. about 500× faster than the sequential resolution. For such a short execution time, the ramp-up and shut-down phases can not be neglected. In fact, the algorithm terminates before the load balancing mechanism can efficiently distribute the search space among all $32 \times 600 = 19\,200$ IVMs.

Furthermore, one can observe that the node processing rates for most QAP instances (except the *esc* class) are lower than the ones for the FSP instances. We recall that for the non-accelerated MC-B&B it is the contrary (cf. Figure 2.7). This indicates that the GPU-acceleration of the QAP bounding operator is much less efficient than for the FSP. Indeed, compared to the 32-threaded multi-core B&B the acceleration factors for the QAP range from 6 to 10× (instead of 22 to 37× for FSP).

For the n-Queens problem it does not make sense to offload the computation of bounds, which consumes only 1/3 of the total execution time (for IVM). Besides the fact that one could not expect more than 1.5× speedup, the cost of copying subproblems to the device is most likely larger than the very low evaluation cost.

2.5.3 Evaluation of Work Stealing Strategies

MC-B&B: Table 2.1 compares the five work stealing strategies, *honest-½*, *random-½*, *largest-½*, *ring-½* and *ring-1/T* for the IVM-based MC-B&B (without GPU-acceleration). FSP instance *Ta030*, QAP instance *nug16a* and the 16-Queens problem are used as test-

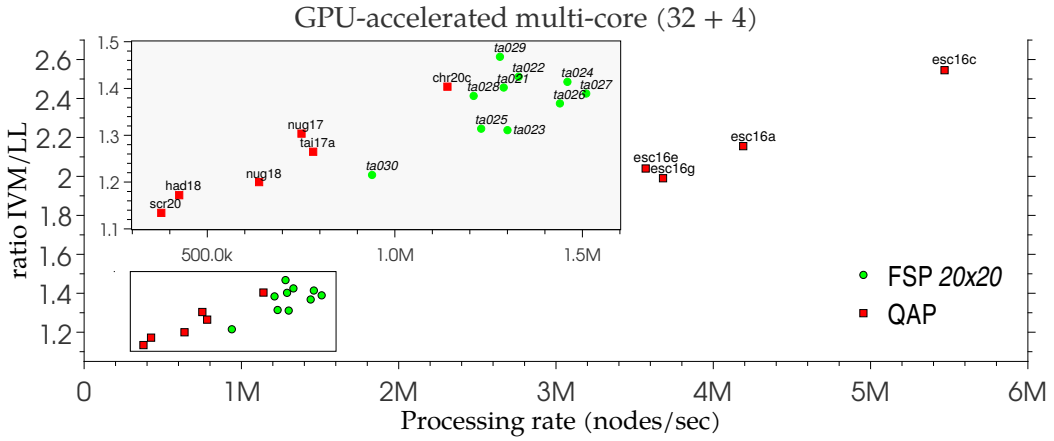


Figure 2.10: Node processing rates for IVM-based GPU-accelerated MC-B&B and ratio with LL-based counterparts (32 threads on 2×E5-2630v3 + 4×GTX980, $M = 600$ pools per thread). Notice that a zoom on the lower left corner is shown.

cases. In Table 2.1 the column on the left indicates the instance being solved, the number of decomposed nodes (i. e. the size of the critical tree), as well as the measured sequential execution time and processing speed (in kn/s). All reported results are averages obtained from 5 independent executions. In order to evaluate the scalability of the proposed work stealing approach up to 32 threads, results are reported for 16 and 32 threads. The goal is also to verify whether the proposed multi-core B&B algorithm can take benefit from the CPUs hyper-threading capabilities.

Three metrics are considered to evaluate the efficiency of the work stealing strategies.

- The speedup compared to a sequential execution using a single thread on the same system, computed as $\frac{T_1}{T_p}$ where T_p designates the parallel execution time using p threads.

- The cumulated time (over all threads) that is spent waiting for new work. For each work stealing operation this timer starts before the victim receives a request and stops after the request is answered or the algorithm terminates. Therefore, the total waiting time also gives an upper bound on the time spent by all threads to answer requests.

- The *relative load imbalance* (RLI) metric [SG97] is computed as follows:

$$RLI = \frac{W_{max} - \frac{W_{tot}}{p}}{W_{max}} = 1 - \frac{W_{tot}}{pW_{max}},$$

where $W_{tot} = \sum W_i$, $W_{max} = \max W_i$ with W_i the number of nodes decomposed by thread i, and p the number of threads. It should be noted that the number of decomposed nodes is an imprecise measure for the amount of "useful work" because nodes require a variable amount of time for evaluation. However, if we suppose that threads spend, on average, an equivalent amount of time per node evaluation, this should metric should still be a valid indicator for comparing the achieved load balance. A value of $RLI = 0$ means that all threads have decomposed exactly the same amount of nodes, i.e. $W_{max} = W_i = W_{tot}/p$. Supposing that the most heavily loaded thread performs twice as many node decompositions as the average, the value of RLI is equal to 0.5. If one thread does all the work, then RLI equals its maximum value $1 - \frac{1}{p}$.

From Table 2.1, one can observe that - in most configurations - the *ring*-1/2 and *ring*-1/T strategies are both clearly inferior to the three other strategies. An exception is the application of the algorithm to the FSP test-case using 16 threads. In that case, using *ring*-1/T allows to also reach near-linear speedup and good load balance. However, using 32 threads, the same strategy achieves only 20.5× speedup compared to 22.4× using *honest*-1/2, *random*-1/2 or *largest*-1/2. For all test-cases one can observe that the performance of the *ring*-1/T strategy degrades when using 32 instead of 16 threads. Indeed, when increasing the number of threads from 16 to 32, the cumulative waiting time and load imbalance increase dramatically for both *ring* strategies. This indicates that the scalability of *ring* selection policy is limited. For all test-cases the 1/T granularity policy is clearly better suited for the *ring* topology than the 1/2 granularity.

A comparison of the three better-suited work stealing strategies, *honest*-½, *random*-½ and *largest*-½, shows that they perform similarly for the FSP and QAP test cases. Using 16 threads, the three strategies achieve near-linear speedup ratios of 14.8× to 15.7×. Using 32 threads, the exploitation of hyperthreading capabilities allows to speed up the execution by an additional 25% for n-Queens and 40% for the FSP.

Differences between these three strategies become apparent for the fine-grained 16-Queens problem. Using the *largest* strategy and 16 threads, a speedup of 13.4× is reached, compared to 15.2× using the *honest* strategy. The relatively poor performance of the *largest* strategy is observed despite low waiting times and good load balance according to the RLI metric. This is most likely due to the higher computational overhead of the *largest* selection policy, which becomes significant compared to the very low node evaluation cost. It also appears that the *honest* victim selection policy is particularly well adapted to the n-Queens problem. Indeed, for all strategies one can observe that the achieved speedup decreases as the granularity (cost of evaluating one node) becomes

finer, except for the *honest*-1/2 strategy.

GMC-B&B: Table 2.2 reports the experimental results obtained for the GPU-accelerated MC-B&B using the five adapted work stealing strategies. For all experimentations 32 threads are used. Each thread has its own CUDA stream and threads are mapped to GPUs in round-robin fashion. No particular measures are taken to control the affinity of threads to cores, in particular with respect to NUMA effects.

As explained, offloading the node evaluation for the *n*-Queens problem is not envisioned because it consumes less than half of the algorithms total execution time. For the FSP results are reported as averages over three groups, *small*, *medium* and *large*, as shown in the first column of Table 2.2. The objective is to analyze how the total workload (tree size) impacts the considered metrics. Three QAP instances are selected: *scr20*, *nug18* and *esc16a*. For the first two instances, *scr20* and *nug18*, B&B develops a critical tree of almost the same size, about 25×10^6 nodes. For the third QAP instance, *esc16c*, the critical

Table 2.1: Comparison of work stealing strategies for MC-B&B using $p = 16, 32$ threads (2×E5-2630v3). RLI = relative load imbalance = $1 - {}^{W_{tot}}/_{pW_{max}}$

instance		p	*honest*-1/2	*random*-1/2	*largest*-1/2	*ring*-1/2	*ring*-1/T
*Ta*030	T_1/T_p	16	15.7	15.6	15.4	10.0	15.6
nodes:		32	22.4	22.4	22.4	10.3	20.5
1.6 M nodes	Elapsed (sec)	16	57.4	57.5	58.4	89.6	57.6
sequential:		32	40.2	40.1	40.1	87.5	43.9
899 sec	Wait (sec)	16	0.3	0.6	0.9	470	3.5
1.8 kn/s		32	1.3	0.7	0.2	1 915	187
	RLI	16	0.09	0.09	0.10	0.33	0.08
		32	0.10	0.10	0.12	0.67	0.18
*nug*16*a*	T_1/T_p	16	14.9	14.8	14.8	8.3	14.6
nodes:		32	19.7	19.8	19.7	7.3	17.4
0.84 M nodes	Elapsed (sec)	16	6.9	7.0	7.0	12.7	7.1
sequential		32	5.2	5.2	5.2	14.6	5.9
103 sec	Wait (sec)	16	0.07	0.04	0.05	96	2.5
8.2 kn/s		32	0.33	0.32	0.30	360	41
	RLI	16	0.14	0.16	0.18	0.53	0.21
		32	0.17	0.19	0.21	0.79	0.43
16-Queens	T_1/T_p	16	15.2	13.1	13.4	3.2	8.7
nodes:		32	17.0	16.8	16.0	3.0	10.8
1.1 G nodes	Elapsed (sec)	16	32.1	37.4	36.7	172.7	57.3
sequential:		32	28.8	29.1	30.6	183.2	46.7
489 sec	Wait (sec)	16	5.7	75.4	8.6	2 260	368
2216 kn/s		32	5.3	67.6	12.1	5 355	871
	RLI	16	0.04	0.16	0.08	0.80	0.40
		32	0.02	0.10	0.02	0.89	0.59

tree is more than 14 times larger (356×10^6 nodes) but its processing requires (for most work stealing strategies) an execution time equivalent to the one of *scr20*. In other words, compared to *scr20* the node processing speed for *esc16c* is approximately 14 times higher.

Table 2.2 does not show acceleration factors compared to either a sequential CPU execution or a single-threaded GPU-accelerated execution. The reason is that these numbers would not be very meaningful, considering that the presence of idle threads frees GPU-resources for active threads, and improves the processing speed of the latter. In all experiments the number of IVMs per thread is fixed to $M = 600$ according to Figure 2.9. Using a lower value for M does not improve the overall execution time, but it improves the acceleration compared to using a single CPU-thread with GPU-acceleration, as it reduces competition among threads for shared GPU resources. Instead, Table 2.2 shows the achieved node processing rate (in Mn/s).

As for the MC-B&B algorithm, the two *ring*-based strategies are clearly outperformed by the *honest*-$1/2$, *random*-$1/2$ and *largest*-$1/2$ strategies among which only marginal differences are observed. For instance, one can notice that *largest*-$1/2$ allows to reach about 10% higher node processing rate when solving small FSP instances. Solving medium-sized, respectively large-sized FSP instances the three better performing strategies all allow to reach an average processing rate of 1.33 Mn/s, respectively 1.42 Mn/s.

Again, the *ring* strategy yields better results with the $1/T$ granularity policy than with the $1/2$ granularity policy. The results reported in Table 2.2 are obtained on a 4-GPU system composed of 4 Maxwell GTX980 devices. In [GLM+16] we performed similar experiments using a single Kepler K20m device. While the results reported therein are qualitatively the same, the performance gap between the *ring*-$1/T$ and the better work stealing strategies is narrower. This is due to the fact that the algorithm presented in [GLM+16] uses the interval-splitting procedure which generates more computational overhead, but allow a finer control of the stealing granularity - as discussed in Subsection 2.2.4. Using the subtree-based interval splitting procedure, control of the work stealing granularity is too coarse to make the *ring* strategy work efficiently.

2.5.4 Performance evaluation on Intel Xeon Phi

To evaluate the vectorization of the FSP bounding function we compare the time spent for the resolution of instance *Ta030* using different compiler options. Table 2.3 shows the time required for solving *Ta030* using (1) the original version of the bounding function (Algorithm 3), (2) the modified bounding function (Algorithm 4) with disabled vectorization (`-no-vec`), (3) the modified bounding function with enabled vectorization (`-xHost`).

Table 2.2: Comparison of work stealing strategies for GPU-accelerated MC-B&B using $p = 16, 32$ threads (2×E5-2630v3 + 4 GTX980). RLI = relative load imbalance = $1 - {}^{W_{tot}}/_{pW_{max}}$

instance		honest$^1/_2$	random$^1/_2$	largest$^1/_2$	ring$^1/_2$	ring$^1/_T$
Ta028, 029, 030 Avg: 5.5×10^6 nodes	Mn/s	1.09	1.14	1.23	0.63	0.91
	Wait (sec)	5.5	3.4	2.7	207	105
	RLI	0.36	0.37	0.21	0.91	0.78
Ta021, 022, 024, 025 Avg: 36.2×10^6 nodes	Mn/s	1.33	1.33	1.34	0.87	1.10
	Wait (sec)	7.4	4.3	4.2	988	531
	RLI	0.10	0.12	0.05	0.86	0.72
Ta023, 026, 027 Avg: 89.8×10^6 nodes	Mn/s	1.44	1.42	1.42	0.94	1.21
	Wait (sec)	8.2	4.1	4.5	2 185	1 055
	RLI	0.07	0.07	0.03	0.86	0.70
scr20 25.3×10^6 nodes	Mn/s	0.37	0.38	0.38	0.31	0.35
	Wait (sec)	34.2	8.8	10.0	1 975	888
	RLI	0.12	0.08	0.05	0.85	0.61
nug18 25.0×10^6 nodes	Mn/s	0.64	0.64	0.64	0.52	0.58
	Wait (sec)	16.3	7.7	7.0	1 160	588
	RLI	0.11	0.10	0.06	0.85	0.66
esc16c 356.1×10^6 nodes	Mn/s	5.35	5.47	5.68	2.87	4.09
	Wait (sec)	3.2	2.3	3.2	2 835	1 156
	RLI	0.02	0.06	0.03	0.84	0.60

All experiments are performed with the maximum number of available hardware threads, i.e. 32 threads on the dual-socket Xeon system and 240 threads on Xeon Phi 5110P. The random-$^1/_2$ strategy is used. The compiler is Intel `icc`, version 17.0.

As shown in Table 2.3, when vectorization support is disabled, the original version of the FSP bounding function is faster than the new one. On Xeon Phi the proposed implementation of the bounding operator is even more than twice as slow when vectorization support is disabled. This can be explained by the increased memory requirements of the revisited bounding operator. However, enabling AVX-256 vectorization, the revisited implementation of the bounding function allows an improvement of +24.7% compared to the initial version. On Xeon Phi the vectorization allows to an improvement of +42.3% compared to the non-vectorized implementation of the bounding operator.

Table 2.3: Resolution of FSP instance *Ta030* using original non-vectorized (old), proposed non-vectorized (new -no-vec) and proposed vectorized lower bounding procedure (new -vect). Xeon: 2×E5-2630v3, Xeon Phi: 5110P. Compiler: Intel `icc 17.0`

	old	**new -no-vec**	**new -vect**
2×Xeon (32 threads)	36.0	41.1	27.1
Xeon Phi (240 threads)	42.5	98.6	24.5

The second part of our experiments on Xeon Phi concern the scalability of MC-B&B. As the experiments with up to 32 threads conducted in the previous subsection do not reveal any significant performance differences between the *random*, *largest* and *honest* work strategies, we verify whether they become more apparent for a larger number of threads. For each of the three work stealing strategies FSP instance *Ta*030 is solved using 1, 15, 30, 60, 120, 180 and 240 threads. The achieved parallel efficiency is shown in Figure 2.11. Parallel efficiency is computed as $\frac{T_1}{pT_p}$, where p designates the number of threads (IVMs) and T_p is the measured execution time using p threads on Xeon Phi 5110P.

As one can see in Figure 2.11, the algorithm scales linearly up to 60 threads, which equals the number of cores of Xeon Phi 5110P. For more than 60 threads, the parallel efficiency decreases and drops to 0.44, for a speedup of 106× using 240 threads. Again, no significant difference between the three work stealing strategies can be observed. The same performance evaluation for QAP instances *esc*16*a* and *nug*18 leads to almost identical results, i.e. linear scalability up to 60 threads and 0.44 efficiency for 240 threads. For the sake of readability those results are not shown in Figure 2.11.

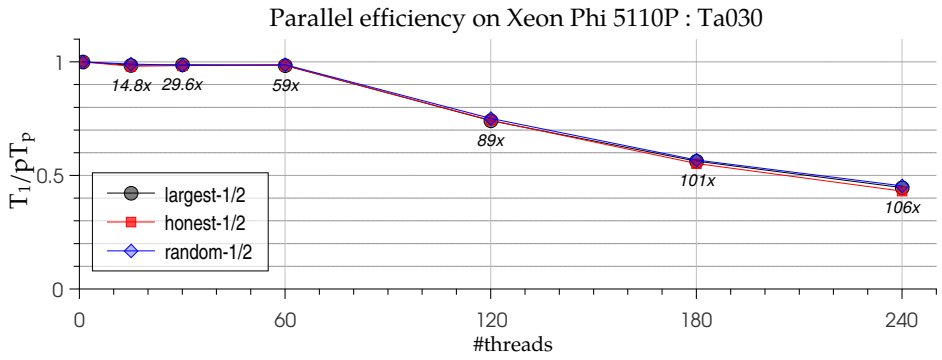


Figure 2.11: Parallel efficiency ($\frac{T_1}{pT_p}$) obtained for work stealing strategies *oldest*, *random* and *largest* solving FSP instance *Ta030* on Intel Xeon Phi 5110P. For QAP instances *nug18* and *esc16a* almost identical results are obtained. The labels show the parallel speedup T_1/T_p (T_p: execution time using p threads on Xeon Phi 5110P, $T_1 = 2\,607$ sec). Average over 5 runs.

Figure 2.12 shows the parallel efficiency obtained when solving 15-Queens on Xeon Phi using up to 240 threads. In contrast to the FSP and QAP problems, efficiency is lower than 1 even for less than 60 threads and the three work stealing strategies behave differently. In accordance with the results reported earlier in Table 2.1, best efficiency is obtained when using the *honest*-1/2 strategy. Using 60 threads, for this strategy a speedup of 53× is observed, against 48× for *largest*-1/2 and 45× for *random*-1/2. Using 240 threads, each of the three strategies allows to reach a speedup of 80× for 0.33 efficiency.

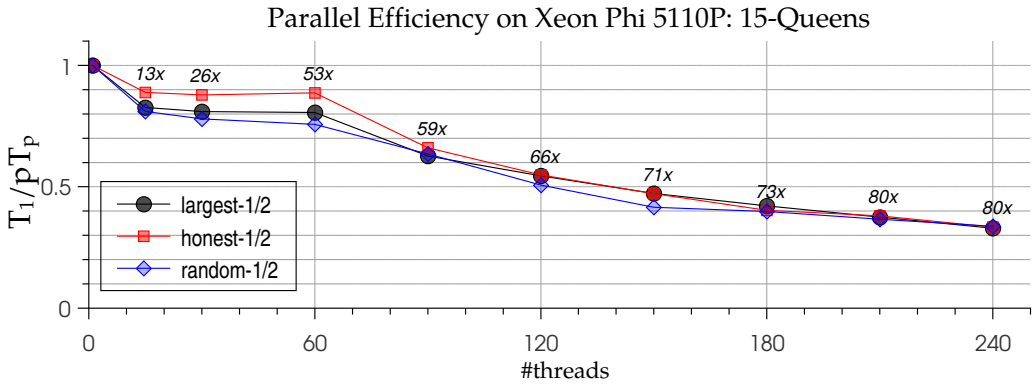


Figure 2.12: Parallel efficiency ($\frac{T_1}{pT_p}$) for work stealing strategies *oldest*-1/2, *random*-1/2 and *largest*-1/2 for 15-Queens. The labels show the parallel speedup T_1/T_p (T_p: execution time using p threads on Xeon Phi 5110P, $T_1 = 735$ sec). Average over 10 runs.

2.5.5 MC-B&B: performance on different multi-core CPUs

In this subsection we evaluate the performance of the IVM-based MC-B&B on different multi-core processors and the many-core Xeon Phi 5110P processor. Only the *random*-1/2 strategy is considered. The goal of this subsection is to compare the performance reached by MC-B&B on different multi-core architectures.

In Figure 2.13, the node processing rate (in kn/s) obtained for FSP instance *Ta028* and the following configurations is shown.

(1) 2×E5-2680v4 (Broadwell, 2 × 14 cores@2.4 GHz), not vectorized, `gcc` 5.4

(2) 2×E5-2630v3 (Haswell, 2 × 8 cores@2.4 GHz, AVX2), vectorized, Intel `icc` 17.0

(3) 2×E5-2630v3 (Haswell, 2 × 8 cores@2.4 GHz), not vectorized, `gcc` 5.4

(4) 2×Power8+ (2 × 10 cores@2.86 GHz, SMT8), not vectorized, IBM `xlc++` 13.1

(5) 5110P Xeon Phi (60 cores@1.05 GHz, 512-bit SIMD) , vectorized, `icc` 17.0

More detailed hardware specifications are provided in Appendix A.3.

The fine dashed lines in Figure 2.13 show the respective linear speedups, based on the node processing rate achieved on a single core. The left-hand side shows the three Xeon CPU configurations and the right hand side shows the Xeon Phi and dual-Power8+ configurations. One can notice that near-linear speedup is achieved up to the number of cores on all Xeon and Xeon Phi processors. In terms of single core performance, at equal clock rates the configuration (1) performs slightly better than configuration (2),

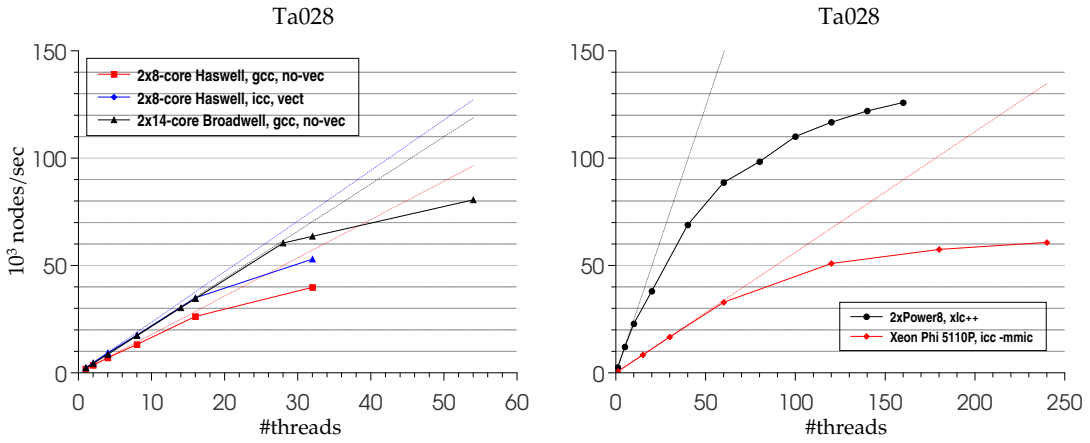


Figure 2.13: Node processing speed (in kn/s), solving FSP instance *Ta028* (8 088 505 nodes).

even though the latter is vectorized. This indicates that the performance of MC-B&B (applied to FSP) can benefit from larger caches (35 MB vs. 20 MB) in configuration (1). Best overall performance is achieved in configuration (4) using 160 hardware threads.

2.6 Conclusions

In this chapter, we have revisited the design and implementation of B&B algorithms for multi-core and many-integrated core (MIC) architectures. We have presented a parallel multi-core B&B algorithm (MC-B&B) based on an innovative data structure, called Integer-Vector-Matrix (IVM).

IVM is a compact data structure, dedicated to permutation problems, which allows to store and manage the pool of subproblems more efficiently than conventional data structures (e. g. stacks, deques, priority queues) referred to as linked-lists (LL). Existing IVM-related works in the literature compare IVM-based and LL-based parallel B&B for multi-core systems and show that IVM significantly reduces memory requirements and pool management time. Also, previous works have shown that the work stealing mechanism using factoradic-based work units achieves good load balance while requiring less inter-thread synchronization time than node-based work stealing mechanisms. However, the IVM-based B&B algorithm has been only been applied to the FSP, for which pool management amounts for barely 1% of the sequential execution time. Consequently, using IVM instead of conventional data structures has not allowed a significant reduction of the algorithm's execution time. The following points summarize our contributions presented in this chapter.

- A first contribution consists in revisiting and validating of the IVM data structure by applying it to other permutation problems, showing experimentally that IVM-based B&B algorithms can perform significantly better than LL-based ones. Namely, in addition to FSP, we applied the IVM-based B&B to the QAP and the n-Queens puzzle problem. An extensive experimental evaluation using these problems in addition to FSP shows that the question whether a B&B application can benefit from using IVM essentially depends on the granularity of the tackled problem, i.e. the cost of evaluating a subproblem.

- We have proposed a hybrid GPU-accelerated version of the MC-B&B algorithm (GMC-B&B). GMC-B&B accelerates the bounding operations performed by each thread by offloading the computation of lower bounds to the GPU. Using the FSP as a test-case, we show that even when the sequential execution time of B&B is strongly dominated by the evaluation of subproblems, an efficient implementation of the pool management can become important. Indeed, for the FSP the cost of evaluating subproblems can be reduced dramatically by offloading this computation to GPUs, increasing the relative importance of handling the subproblems and thus the benefits of using IVM.

- As the performance of accelerator devices like Intel Xeon Phi heavily relies on 512-bit wide vector processing units, the bounding operator for the FSP is revisited and an efficient vectorization mechanism is proposed.

- Besides increasing per-thread performance through the use of many-core accelerators, the focus of this chapter is put on maximizing parallel efficiency by the means of efficient load balancing mechanisms. In the IVM-based approach, work units exchanged between threads are intervals of factoradics instead of sets of nodes. For the MC-B&B and its GPU-accelerated extension GMC-B&B five work stealing strategies are presented. These factoradic-based work stealing schemes are characterized by different victim selection and granularity policies. For all three test-cases, QAP, FSP and n-Queens the work stealing strategies are evaluated experimentally, with and without GPU-acceleration and at different scales.

A summary of the main experimental results is given in the following.

- For FSP and QAP, MC-B&B scales linearly almost linearly on up to 28 threads on dual-socket multi-core CPU systems and up to 60 threads on Xeon Phi. Furthermore, the MC-B&B takes good benefit from hyperthreading/SMT capabilities:

on Intel Xeon processors improvement rates of about 30% are observed. On 20 IBM Power8+ cores MC-B&B reaches speedups of 50× and more, exploiting 8-way simultaneous multithreading.

- For the very fine-grained n-Queens problem, the IVM-based MC-B&B algorithm allows to explore 3 times as many nodes per second as its linked-list (LL) based counterpart. Solving more coarse-grained problems like FSP or QAP, less than +10% improvement is observed. For the GMC-B&B algorithm significant improvements ranging from 1.2× to 2.5× are observed even for these problems.

- The GPU-acceleration considerably speeds up the exploration process. For FSP, GMC-B&B achieves 30-40 times higher node processing rates than MC-B&B without GPU-acceleration. The lower bounding procedure for QAP is less suited for parallel evaluation on the GPU, with acceleration factors between 6× and 10×.

- The vectorized version of the FSP lower bounding procedure is about 1.3× faster than the non-vectorized version using 256-bit AVX2 vector instructions, respectively 1.7× using 512-bit vector instructions on Intel Xeon Phi. This relatively low gain from vectorization can be explained, at least partly, by a highly irregular, thus inefficient memory access pattern in the bounding function.

Chapter 3

GPU-centric Branch-and-Bound

Contents

3.1 Introduction 76

3.2 Discussion of design choices 77

3.3 GPU-B&B and GPU-backtracking 81

3.3.1 GPU-B&B: 2-level parallelization 81

3.3.2 Thread-data mapping and branch divergence reduction 86

3.3.3 GPU-BT: 1-level parallelization 90

3.4 Work stealing strategies for GPU-B&B 92

3.4.1 Victim Selection policies 92

3.4.2 Work stealing for multi-GPU-B&B 96

3.5 Experiments 97

3.5.1 Evaluation of Mapping approaches 97

3.5.2 Evaluation of Work Stealing strategies 101

3.5.3 Scalability analysis 103

3.5.4 Multi-GPU-B&B performance evaluation 107

3.5.5 Hybrid CPU-multi-GPU-B&B 111

3.6 Conclusions 114

Related publications

- Gmys Jan, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel, "IVM-based Work Stealing for Parallel Branch-and-Bound on GPU" in *Parallel Processing and Applied Mathematics (PPAM'15). Lecture Notes in Computer Science*, 9573, 548-558 (2016), https://doi.org/10.1007/978-3-319-32149-3_51 [**Best Paper Award** in workshop on GPU computing at PPAM'15]

- Gmys Jan, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel, "IVM-Based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems" in *Concurrency & Computation : Practice & Experience, (Special Edition PPAM'15)* (2016), 29(9), https://doi.org/10.1002/cpe.4019.

- Gmys Jan, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel,"A GPU-based Branch-and Bound algorithm using Integer-Vector-Matrix data structure", In *Parallel Computing, (Special Issue: Theory and Practice of Irregular Applications)*, Vol. 59, 2016, p. 119-139, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2016.01.008.

- Pessoa Tiago Carneiro, Gmys Jan, Melab Nouredine, de Carvalho Junior Fransisco Heron, Tuyttens Daniel, "A GPU-based Backtracking Algorithm for Permutation Combinatorial Problems" in *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'16). Lecture Notes in Computer Science*, 10048, 310-324 (2016), https://doi.org/10.1007/978-3-319-49583-5_24

3.1 Introduction

In this chapter the B&B algorithm is revisited for GPUs. In contrast to the previously presented GPU-accelerated algorithm, this chapter deals with a GPU-centric implementation of B&B, meaning that it completely bypasses the CPU. To the best of our knowledge, the IVM-based GPU-B&B algorithm presented in this chapter is the first to perform all four B&B operators on the device. The key to achieving this is to use the IVM data structure for pool management, which is much better suited for GPUs than dynamic linked-list data structures.

Section 3.2 motivates fundamental design choices for the proposed implementation, taking alternative designs into account and considering hardware characteristics as well as memory requirements for the three test-cases, FSP, QAP and n-Queens. In Section 3.3 a detailed description of the implementation is provided, considering different mappings of the algorithm. Two variants of the GPU-centric algorithm are proposed, both based on the parallel tree exploration model. In addition to parallel tree exploration, the first variant uses a second level of parallelism, where generated nodes are evaluated in parallel. The second variant is designed for fine-grained problems where the addition of a second level is inefficient, using the parallel tree exploration model alone. Challenges addressed in this section include the reduction of thread divergence which results from control flow irregularities and the placement of data in the hierarchical GPU memory.

A central issue for both variants of GPU-B&B is the implementation of load balancing mechanisms on the device. Section 3.4 presents five work stealing strategies for the GPU-centric B&B. We propose a GPU-based trigger-mechanism which adapts these work stealing strategies to the 1-level GPU-B&B variant for fine-grained problems. Furthermore, in order to enable GPU-B&B to exploit multi-GPU systems an hierarchical work stealing approach is proposed for inter-GPU load balancing. An approach for load balancing in a hybridized version combining MC-B&B and GPU-B&B is presented.

Finally, in Section 3.5 the GPU-centric B&B is experimented, using the FSP, QAP and n-Queens problems as test-cases. The experimental study includes an evaluation of different mapping choices, a comparison of work stealing strategies and an analysis of scalability and stability. The performance of multi-GPU-B&B and its hybridization with MC-B&B is also evaluated.

3.2 Discussion of design choices

In the previous chapter we presented a GPU-accelerated approach for the MC-B&B algorithm, where the parallel evaluation of lower bounds is offloaded to the GPU. In order to reduce the amount of data transferred to the device only parent nodes are offloaded and branching is performed on the device. Is such approaches it is challenging to find an optimal value for the size of the offloaded pool (i.e. the number of IVMs per thread), because it requires finding a compromise between GPU occupancy and host-device communication overhead. To solve this problem, [Cha13] propose an auto-tuning heuristic which automatically adjusts the size of the offloaded pool at runtime. Similar to our approach presented in the previous chapter, [VDM13] use streams and asynchronous copies to overlap pool management and GPU-based bounding. However, the efficiency of these approaches also depends on the problem being solved, in particular on the amount of work performed per transferred piece of data. Indeed, in order to hide communication overhead and overlap sequential host processing with parallel node evaluations, the offloaded workload must be large enough. An alternative is the implementation of the entire algorithm on the device. This eliminates all CPU-GPU communication and reduces the sequential portion of the algorithm to initialization and output. Especially problems with less costly a node evaluation functions, like n-Queens, could benefit from such an implementation.

One of the key features of IVM is its constant memory footprint. As dynamic memory allocations in device-code are known to perform very poorly, this makes IVM much more suitable to GPU than LL data structures. The basic idea is to allocate a *fixed* number (T) of IVM structures and all data required for bounding in device memory before starting the exploration process. Then, the T IVMs are used to explore the interval $[0, n![$ in parallel, without requiring any further allocations or host-device data transfers. In the following we analyze memory requirements of the IVM-based B&B and provide further motivation of fundamental design choices.

Memory requirements

The dynamic allocation of memory on the GPU heap is possible, but the efficient and scalable implementation of SIMD-parallel memory allocators is still an active field of research [VH15; WWWG13]. Moreover, using LL-based data structures there is a risk that the size of the work pool(s) exceeds the size of available global memory. Assuming that nodes are encoded using 1-byte integers, the size of a node is at least n bytes. The work pool of one DFS-B&B process may contain up to $\frac{n\times(n-1)}{2}$ nodes, which amounts to $\frac{n^3-n^2}{2}$

bytes, i.e. ∼500 kB for a problem of size $n = 100$. Using parallel tree exploration, this number has to be multiplied by the number of independent B&B processes. Assuming that this number equals the number of maximal concurrent threads on a GTX980 GPU, a LL-based parallel B&B algorithm would require up to $\sim$ 32 000× 500 kB=16 GB of memory, which exceeds the 6 GB of on-board memory.

In contrast to LL data structures, IVM needs only one allocation of contiguous memory. For a problem instance with n jobs, the storage of the matrix requires n^2 bytes of memory (for $n < 127$, using 1-byte integers). Moreover, $3n$ bytes are needed to store the position-, end- and direction-vectors, 1 byte to store the integer and n bytes to store permutations before calling the bounding operator. In total, the IVM data structure requires a constant amount of $1 + 4n + n^2$ bytes of memory, i. e. 10.4 kB per IVM for $n = 100$ and 332 MB for 32 000 IVMs .

It is also possible to store only the upper triangular part of the matrix, requiring $1 + 4n + \frac{n(n+1)}{2}$ bytes per IVM. However, this is only beneficial if this allows storing the IVM structures in shared or global memory where they would otherwise not fit in. Also, using 1-byte instead of 4-byte integers is an unnecessary optimization unless it allows to fit some data structures in shared memory. The kernels and device functions presented in the following sections are templated and can be instantiated for different integer types.

From a programming perspective the IVM-structures are easy to handle. The components of all IVMs are merged into single, one-dimensional arrays. In other words, a structure-of-arrays (SoA) layout is used, in constrast to the multi-core implementation which uses AoS. For instance, solving a n-job instance using T IVM structures, the matrices are stored in a one-dimensional array `matrices` of size $T \times n^2$, allocated in global device memory. The element $M(i, j)$ of the kth IVM is accessed by `matrices[indexM(i,j,k)]`, where `indexM` is a wrapper-function defined as in Equation (3.1).

$$indexM(i, j, k) = k \times n \times n + i \times n + j \tag{3.1}$$

For many problems, like the FSP and QAP, some read-only arrays are needed for the computation of lower bounds. Depending on the problem and the problem size, this read-only data may be stored in the GPUs constant memory space – residing in global device memory but accessed through a cache on each streaming multiprocessor. The amount of memory required for the bounding operation, depending on the problem, is shown in Table A.2. Some of the data structures used for the bounding may be loaded to shared memory during the computation of the lower bounds. For the FSP, the issue of finding the best placement of data structures in the hierarchical GPU memory is

addressed in [Cha13].

The amount of shared memory used per block can be allocated at kernel launch-time. In contrast, the size and type of arrays residing in constant memory must be known at compile-time. Therefore, in order to use constant memory for the bounding data structures, it is unfortunately necessary to create different executables for different problems/problem sizes.

Parallelization model

The parallel tree exploration model is used because of its potentially very high degree of parallelism. The question remains whether the subproblems generated by each IVM at each iteration are evaluated sequentially or in parallel.

Nesting the parallel evaluation of bounds model in the parallel tree exploration yields a finer granularity during the bounding phase. As each IVM generates a different number of subproblems at each iteration, this is likely to produce a more regular work load. It also increases the degree of parallelism, meaning that less IVMs are needed to reach sufficient GPU occupancy. For instance, the maximum number of concurrent threads on the GTX980 GPU is 32 768 (16 SM×max. 64 warps/SM×32 threads/warp). Although in some cases higher performance can be reached at lower occupancy [Vol16], at least $25-50\%$ GPU occupancy should be achieved. Therefore, using the parallel tree exploration model alone, at least $T = 10\,000$ IVMs should be used. On the other hand, if generated subproblems are evaluated in parallel and if each IVM generates on average 10 subproblems per iteration, then $T = 1\,000$ IVMs are enough to reach the same level of occupancy during the bounding operation. Keeping the number of IVM structures low also facilitates load balancing.

However, the two-level parallelization can only be beneficial if the node evaluation is at least costly enough to hide the overhead incurred by the second level. Therefore, two variants of GPU-B&B are proposed:

- GPU-B&B : using a two-level parallelization model, designed for costly bounding functions, like the ones used for FSP and QAP.

- GPU-BT : using the parallel tree exploration model alone, designed for B&B with cheap node evaluation functions, like n-Queens. We call this variant GPU-BT - for GPU-backtracking - because DFS-B&B with very simple node evaluation functions is also known as heuristic backtracking [RK93].

Global synchronization

Let's assume for the moment that all four B&B operators can be efficiently implemented on the device. Is it preferable to implement the entire algorithm in one monolithic kernel or to have separate kernels for each operator? In other words : should the outer while-loop of the B&B algorithm be placed inside a B&B kernel or should it be placed around separate select, branch, bound and prune kernels?

There are strong arguments in favor of the monolithic solution, which corresponds to the programming paradigm known as "Persistent Threads" (PT) programming [GSO12]. The idea of the PT programming style is to launch the maximum number of threads that can be resident on the device and keep them alive throughout the entire application. One advantage of this solution is that it avoids kernel launch overhead. The asynchronous launching of an empty kernel is a very low-cost operation, in the order of a few microseconds[1], so the pure kernel launch overhead can be neglected for many kernel workloads. However, for very fine-grained workloads kernel launch overhead may become relevant. Moreover, the loading of data from global memory to shared memory, registers and caches should be included in the kernel launch overhead, because only global, constant, and texture memory spaces are persistent across kernel launches by the same application. Indeed, the advantage of using a single B&B kernel is that frequently used data structures need to be loaded from global memory only once and can reside in registers, shared memory, L1 and L2 caches for the duration of the algorithm.

There are also arguments in favor of the multiple-kernel solution, or rather, against the PT programming solution. We recall that the parallel tree exploration model requires load balancing between independent B&B processes, each represented by one IVM[2]. Therefore synchronization between arbitrary threads is necessary. In other words, a global synchronization barrier is needed. Although CUDA does not natively provide inter-block synchronization primitives, such barriers have been proposed in the literature [XF10]. The proposed GPU-synchronization primitive uses atomic compare-and-swap operations to construct a shared mutex barrier and avoid deadlocking by ensuring a one-to-one mapping between SMs and the thread blocks – which corresponds to the paradigm of PT programming. Indeed, if one thread block is pending because the other blocks occupy all SMs, the resident blocks will wait on the barrier forever, because the pending block has no chance of reaching it. Ensuring that a kernel never uses more blocks than can

1. https://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html (accessed: Oct 6, 2017)

2. In GPU-B&B, contrary to the multi-core algorithm, a B&B process does not necessarily correspond to one thread, but rather to a distinct portion of data, namely an IVM structure. In the following, we designate by "IVM" the data structure, as well as the logical independent B&B exploration process.

be concurrently scheduled on the device (called a *maximal launch*) may be difficult. As occupancy is partially determined by register and shared memory usage, even a slight code change may require changing the kernel configuration. Also, porting the code to a different device may require readjusting the kernel configuration. Moreover, the register usage of the bounding operators for QAP and FSP are quite high, limiting the maximal launch configuration and the degree of concurrency.

For these reasons, we choose to avoid this solution, which leaves implicit synchronization through kernel termination as the only possibility to achieve global synchronization. Finally, breaking the algorithm into multiple kernels allows the individual optimization of each operator: different mappings and memory layouts can be used for each operator, without considering performance trade-offs between different parts of the algorithm.

To summarize, while the option of using a single monolithic B&B kernel is attractive, it limits the flexibility of adapting the algorithm to different problems, makes it more difficult to write portable code and may compromise other performance optimization opportunities. Therefore, we decide to follow the classical CUDA programming model where global synchronization is achieved through kernel termination.

3.3 GPU-B&B and GPU-backtracking

According to the discussion in Section 3.2, the B&B `while`-loop is placed around multiple kernels which implement tree-exploration and load balancing phases. In this section two variants of the exploration phase are presented. Subsection 3.3.1 presents the GPU-B&B variant using a two-level approach combining the parallel tree exploration and the parallel evaluation of bounds model. In Subsection 3.3.2 different mapping schemes for GPU-B&B are presented. Subsection 3.3.3 presents the GPU-backtracking for fine-grained permutation problems.

3.3.1 GPU-B&B: 2-level parallelization

At each point of the algorithm an IVM can be in one of three states: *exploring*, *empty* or *initializing*. Figure 3.1 provides an overview of the GPU-B&B algorithm in the form of a flowchart.

The algorithm starts by reading user-defined parameters (e. g. problem size (n, problem and instance to solve, number of used IVMs (T)) and initializes all required data structures on the GPU (e. g. IVM data structures, constant data used in bounding operation). After the initialization, exploration and load balancing phases alternate until the termination condition is met.

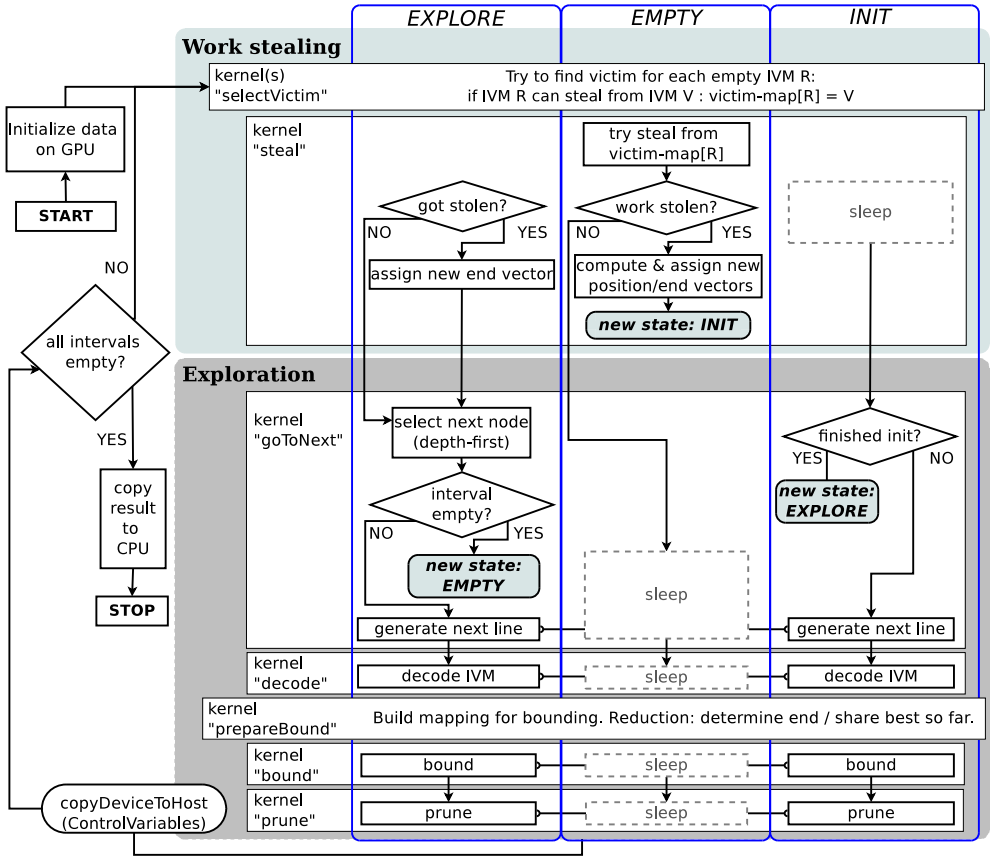


Figure 3.1: Flowchart of GPU-centric B&B algorithm.

The load balancing phase consists of victim selection and work transfer. In this phase, IVMs in the empty state can switch to the initialization or exploring state [3].

The exploration phase consists of four kernels: `goToNext` , `decode` , `bound` and `prune`. The selection and branching operators are merged into a `goToNext` kernel, which consists in performing the next node decomposition for all non-empty IVMs. The kernel `decode` reads the IVM structures and produces subproblems of the form 2/13/4 which can be evaluated by the bounding operator. Kernels `bound` and `prune` implement the B&B operators corresponding to their names.

Moreover, an auxiliary kernel `prepareBound` is used to build the mapping for the bounding operation (explained later on in Subsection 3.3.2). In this phase, the best solution found so far is determined by a min-reduce of the best solutions found by all IVMs. In the same reduction procedure the termination of the algorithm is detected by

3. This depends on the initialization procedure that is used (see Subsection 2.2.4).

searching the maximum of a per-IVM state variable where the `empty` state is encoded as 0 and the two other states as 1. In order to stop iterating through the B&B loop this information needs to be copied to the host at each iteration. Therefore, at the end of each exploration phase a boolean variable indicating the end of the search is copied from device to the host. In order to monitor the algorithm's progress some counter variables, like the number of active IVMs, are also copied to the host. In the following a detailed description of these kernels is provided.

Selection and Branching kernel

The `goToNext` kernel corresponds to the selection and branching operators. Algorithm 5 shows the pseudo-code of this kernel. It performs the selection operator for both, exploring and initializing IVMs. It also updates the IVM-states if necessary. For each exploring IVM it performs the `select-and-branch` procedure described in Chapter 2, Algorithm 1. If an exploring IVM finds no promising node (Algorithm 5, Line 14), then its state variable is set to *empty*. If the end of an IVM's initialization process is detected (Algorithm 5, Line 5) it switches to *exploring*. It is possible that, within one iteration, an empty IVM receives an interval, finishes initializing and returns to the *empty* state.

As explained in Subsection 2.2.4, the initialization process differs from the normal exploration process only in the selection operator. The initialization-selection consists in choosing the node pointed by the position-vector. Thus, only the `generate-next-line` branching procedure is performed by IVMs in the *initializing* state. Each IVM is handled by a single thread, as the operations that modify each IVM structure are essentially of sequential nature. This kernel contains a high number of conditional instructions depending on the state of an IVM as well as on its current depth in the B&B tree. In order to avoid thread divergence the mapping of threads onto the IVM structures (Algorithm 5, Line 3) must be chosen carefully. This mapping is discussed in Subsection 3.3.2.

Bounding kernels

The bounding kernel is designed to work in combination with the polytomic branching scheme described in Subsection 1.3.1. We recall that that a parent node is decomposed in two sets of children nodes, one obtained by fixing unscheduled jobs in the beginning, and one by fixing them in the end. In the bounding kernel all generated children nodes in both sets are evaluated, but only the set for which the average lower bound is higher is retained. The set with the higher average lower bound is likely to develop smaller subtrees as the pruning of branches is more likely to occur.

Algorithm 5 Kernel: `goToNext`

```
1:  kernel goToNext
2:      thdIdx ← blockIdx.x*blockDim.x + threadIdx.x
3:      ivm← map(thdIdx)                                  ▷ Map threads to IVMs
4:      if (state[ivm]=init) then
5:          if (init-finished(ivm)) then
6:              state[ivm]← exploring
7:          else
8:              generate-next-line(ivm)                   ▷ branch
9:          end if
10:     end if
11:     if (state[ivm]=exploring) then
12:         select-and-branch(ivm)                        ▷ Alg. 1
13:         if (exploration-finished(ivm)) then
14:             state[ivm]← empty
15:         end if
16:     end if
17: end kernel
```

The computation of the lower bounds is performed by a device function `computeLB`. This device function is a sequential implementation of the lower bounding procedure, which returns a lower bound (LB) value for a subproblem provided in the form 2/13/4 (`schedule`= 2134, `limit1`= 0, `limit2`= 3). In principle any lower bounding procedure can be used in place of `computeLB`.

Each thread is responsible for the computation of one bound. It is not necessary to generate all subproblems before calling this kernel. Instead, a pointer to the father nodes is passed as parameter to the kernel. Using a mapping policy, detailed in Subsection 3.3.2 each thread generates a distinct subproblem from the father node and computes its lower bound. These lower bound values are stored and atomically added to the values `sumBegin` and `sumEnd`, which are used to decide which decomposition is retained. For each father subproblem of depth I, the lower bounds for $2 \times (n - I) = 2 \times (\texttt{limit2} - \texttt{limit1} - 1)$ children are computed. The parent-children relation and the bounding procedure are illustrated in Figure 3.2.

The number of active threads in the bounding kernel is therefore given by

$$2 \times \texttt{todo} = 2 \times \sum_{\substack{ivm=0 \\ ivm \neq \emptyset}}^{T} (n - \texttt{row}[ivm]) \leq 2 \times T \times n.$$

At a given iteration, this quantity depends unpredictably on the number of non-empty IVMs and on their depth in the B&B tree. The maximum $2 \times T \times n$ occurs in the case where all IVMs have non-empty intervals at level 0.

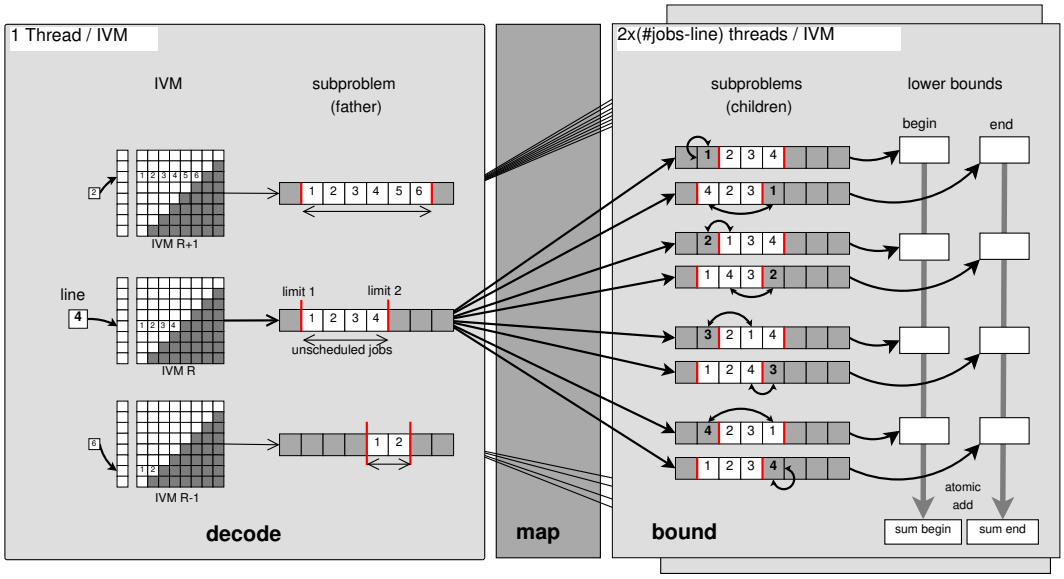


Figure 3.2: Illustration of the decode and bounding phases for GPU-B&B

Each thread that computes a lower bound must be provided the following information: (1) on which IVM it is working, (2) which unscheduled job it is scheduling and (3) on what end of the partial permutation to schedule. A static mapping of threads onto potentially generated children nodes (thus launching $2 \times T \times n$ threads at each invocation) is possible. As this mapping is critical for the performance of the bounding kernel, and thus for the entire algorithm, a remapping phase should precede the calling of the bounding kernel. Building such a mapping generates extra overhead which must be kept low. The mapping and implementation details of the bounding kernel are further discussed in Subsection 3.3.2.

Elimination kernel

In a first step the pruning kernel compares the values `sumBegin` and `sumEnd` for each IVM. Depending on this comparison it uses the set of lower bounds `costBegin` or `costEnd` to perform the pruning of nodes. Then the corresponding nodes, i. e. the cells in row I of the matrix, are sorted according to the corresponding lower bounds.

The sorting is performed sequentially. In this case, a simple, memory efficient sorting procedure is needed. We compared the performance of different sorting algorithms for small arrays of < 100 elements and found that "stupid sort" [Sar10] (also known as "gnome sort") is the best suited for this purpose. An important advantage of this algorithm is that it requires only space for a single auxiliary variable.

After sorting, the pruning operation itself consists in multiplying the corresponding cell in the matrix by -1 if the associated lower bound is greater than the best found solution so far. This kernel is the computationally less intensive one.

3.3.2 Thread-data mapping and branch divergence reduction

Control flow refers to the order in which the instructions, statements or function calls are executed in a program. This flow is determined by conditional and loop instructions, e. g. while-do , switch-case , if-then-else. If the control flow of threads within the same warp diverges, i. e. if they follow different execution paths, the execution of these threads is serialized. This serialization of executions for threads in the same warp is called *thread divergence* or *branch divergence*. The following piece of code is an example where branch divergence is likely to occur:

```
if(state[threadIdx.x] == 1)
   for(int i=0; i<line[threadIdx.x]; ++i)
      foo(i);
```

The `if`-condition and the `for`-loop termination condition depend on values indexed by the thread identifier `threadIdx.x`. Unless these values are identical for all threads inside a warp, branch divergence occurs. Threads for which the `if`-condition evaluates to `false` are disabled until their control flow re-converges with the other threads. Among those threads who execute the `for`-loop, the iteration-count may vary and threads with lower iteration-counts are disabled while the thread(s) with the highest count complete the loop. The negative impact of thread divergence is difficult to quantify, because it depends on the actual time spent in the diverging branches. For example, if one half of a warp execute 100 iterations of the `for`-loop and the other half 101 the penalty is less significant compared to the situation where the other half would execute only 1 iteration. The example illustrates how thread divergence is related to the mapping of threads onto data.

Compactified mapping for the bounding kernel

The most straightforward approach probably consists in mapping each thread onto a child subproblem directly from its `threadId`. This naive approach is shown in Algorithm 6.

For instance, launching $2 \times n \times T$ threads (Line 1), the first $n \times T$ threads place unscheduled jobs in the beginning, the second $n \times T$ threads in the end (n designates the problem size and T the number of IVM structures). Regardless of the IVM's state

or current depth in the tree, $2 \times n$ threads are reserved for each IVM. Each thread is assigned an IVM to work on and a job to schedule, like shown in Line 3. The approach of Algorithm 6 has several disadvantages. The `if`-conditionals in Line 5 and Line 6 mask many of the launched threads, precisely $2 \times I$ threads per father subproblem of depth I, plus $2n$ threads per empty IVM. Moreover, different lanes in the same warp work on different IVMs, thus thread divergence occurs due to different values of `limit1` and `limit2`. If $T \times n$ is a multiple of 32 (warp-size), then the `if-else` conditional (Line 7 and Line 11) does not cause any thread divergence.

Algorithm 6 Kernel: `static-bound`

Input: parent nodes (schedule, limit1, limit2)
Output: lower bounds begin, lower bounds end, sums of lower bounds

```
 1: kernel NAIVE-BOUND(< 2 × n × T threads >)
 2:     thId←blockIdx.x*blockDim.x + threadIdx.x
 3:     ivm← (thId/2n)    job← (thId mod n)    dir← (thId/2)        ▷ static mapping
 4:     load: schedule_thd ← schedule[ivm]
 5:     if (state[ivm] ≠ empty) then                                  ▷ for all non-empty IVM do:
 6:         if (limit1[ivm] < job < limit2[ivm]) then                 ▷ for all unscheduled jobs do:
 7:             if (dir == 0) then                                    ▷ evaluate begin
 8:                 swap(schedule_thd[limit1[ivm]+1], schedule_thd[job])
 9:                 LB-begin[ivm][job]←computeLB(schedule_thd)
10:                 sum-begin[ivm] += LB-begin[ivm][job]              ▷ atomic add
11:             else if (dir == 1) then                               ▷ evaluate end
12:                 swap(schedule_thd[limit2[ivm]-1], schedule_thd[job])
13:                 LB-end[ivm][job]←computeLB(schedule_thd)
14:                 sum-end[ivm] += LB-end[ivm][job]                  ▷ atomic add
15:             end if
16:         end if
17:     end if
18: end kernel
```

Algorithm 7 Kernel: `remapped-bound`

Input: parent nodes (schedule, limit1, limit2), ivm-map, job-map
Output: lower bounds begin, lower bounds end, sums of lower bounds

```
 1: kernel REMAPPED-BOUND(< 2 × todo threads >)
 2:     thId←blockIdx.x*blockDim.x + threadIdx.x
 3:     dir← thId mod 2)
 4:     ivm←ivm-map[thId/2]
 5:     job←job-map[thId/2]
 6:     load: schedule_thd ← schedule[ivm]
 7:     toSwap←(1-dir)*(limit1[ivm]+1) + dir*(limit2[ivm]-1)
 8:     swap(schedule_thd[toSwap], schedule_thd[job])
 9:     LB[dir][ivm][job]←computeLB(schedule_thd)
10:     sum[dir][ivm] += LB[dir][ivm][job]                            ▷ atomic
11: end kernel
```

The goal of the remapping procedure which prepares the bounding is to build two maps `ivm-map` and `job-map` which contain, for `todo` threads, the information which IVM to work on and which job to swap. Using an even/odd pattern these maps provide sufficient information for both groups of threads. After building these maps, the bounding kernel (as shown in Algorithm 7) is called with $2 \times$ `todo` threads, where:

- threads 0 and 1 work on IVM `ivm-map[0]`, swapping job `job-map[0]` respectively to begin/end,
- threads 2 and 3 work on IVM `ivm-map[1]`, swapping job `job-map[1]` respectively to begin/end,
- ...
- threads 2×`todo`−2 and 2×`todo`−1 work on IVM `ivm-map[todo-1]`,...

The remapped bounding kernel is launched at each iteration with a kernel configuration of 2×`todo/blockDim`+1 blocks (simplified in Algorithm 7) which is adapted to the workload. The proposed approach is known as *stream compaction* in the literature. It reduces the number of idle lanes per warp as well as the number of threads launched per kernel invocation. Any thread divergence resulting from the begin-end distinction should also be avoided, as this involves a serialization of the costly `computeLB` procedure. To achieve this, the bodies of the *if-else* conditional (Algorithm 6, Lines 5-17) can be merged into a single one (Algorithm 7, Lines 7-10). Two different arguments of the same type, occurring on the right-hand side of a statement can often be refactored into a single one, like in Algorithm 7, Line 7. The different arrays on the left-hand side are merged into larger ones. This allows to merge the statements of Lines 9, 10 and 13, 14 of Algorithm 6 into single statements (Algorithm 7, Lines 9, 10). The separation of data within these merged arrays is assured by indexing with the variable `dir`, which evaluates differently for even/odd threads.

The computational cost of building the maps `ivm-map` and `job-map` sequentially is prohibitive. Preliminary experiments show that sequential construction consumes more than 25% of the total execution time on the device, and more than 8% on the host (including data transfers).

Figure 3.3 shows an example with $T = 4$ IVMs which illustrates how the maps `ivm-map` and `job-map` are build in parallel on the GPU. First, all active IVMs write the number of jobs that remain to be scheduled (`limit2 - limit1 - 1`) to an auxiliary

array `todo-per-IVM` of length T. Then, the *prefix-sum* of the elements in `todo-per-IVM` is computed. The operation *prefix-sum* is defined as

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

Efficient parallel CUDA-implementations for this operation have been proposed in the literature [HSO07]. The result of this operation indicates for each IVM k at which position of `ivm-map` and `job-map` the data of IVM k starts to be written. Thus, the mapping can be build completely in parallel. Addition of the last values in `todo-per-IVM` and the computed prefix-sum given the total number of jobs to be scheduled for all IVMs, used for configuration of the bounding kernel.

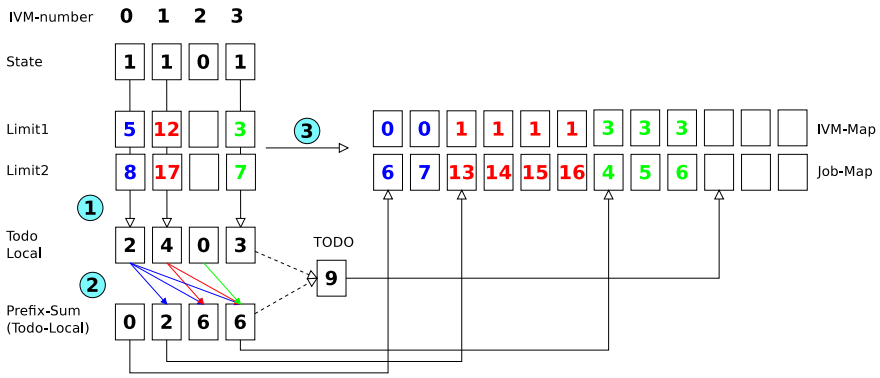


Figure 3.3: Illustration of the remapping phase for the bounding kernel (Algorithm 7).

Reducing thread divergence in the IVM-management kernels

The IVM-management kernels `share`, `goToNext`, `decode` and `prune` require a single thread per IVM. The naive approach consists in launching T threads and mapping thread k on IVM k, for $k = 0, 1, ..., T-1$ (see Algorithm 8). Given the high number of conditional instructions in the IVM-management kernels it is very unlikely that all 32 threads in a warp follow the same execution path if this mapping is used. Indeed, in these kernels control flow divergence results from different IVM-states, different numbers of scheduled jobs at both ends of the active subproblem and from the search for the next node which requires an unknown number of iterations. An alternative mapping, shown in Algorithm 9, can solve this issue.

An entire warp is assigned to each IVM, so all threads belonging to the same warp follow the same execution path. This strategy goes in the opposite direction of the stream

Algorithm 8 Mapping 1

```
1: kernel KERNEL(< T threads >)
2:     ivm←blockIdx.x*blockDim.x + threadIdx.x
3:     DO-SOMETHING-WITH(ivm)
4: end kernel
```

Algorithm 9 Mapping 2

```
1: kernel KERNEL(< warpsize × T threads >)
2:     thId←blockIdx.x*blockDim.x + threadIdx.x
3:     ivm←thId/32
4:     thPos←thId mod 32
5:     //use up 32 threads per IVM for loading data to shared memory
6:     if (thPos == 0) then
7:         DO-SOMETHING-WITH(ivm)
8:     end if
9: end kernel
```

compaction approach proposed for the bounding kernel. As only one thread per IVM is needed, all lanes in a warp except this first are masked. Thus, the kernels are launched with 32 times as many threads as necessary (i.e. 32× T). Using this mapping, the overhead induced by thread divergence completely disappears (although technically, the disabled threads are diverging at Line 6 of Algorithm 9). The drawback is obviously the launching of 31T idle threads.

However, using the two-level parallel model, the degree of parallelism in the pool management kernels is much lower than in the bounding kernel. This, and the fact that the control flow irregularity is very high, justifies the approach of using 1 warp per IVM. Moreover, using only 4-8 IVM structures per block allows to store them in shared memory without limiting the theoretical device occupancy. Also additional threads can be used to load more efficiently from global to shared memory.

3.3.3 GPU-BT: 1-level parallelization

As mentioned, for problems with computationally inexpensive node evaluation functions the parallelization of the bounding phase, as described in the previous section, is not useful. Instead, better performance could be obtained by merging all kernels of the exploration phase into a single kernel. This also means that in order to reach sufficiently high device occupancy, many more IVMs should be used than in the 2-level GPU-B&B. In GPU-BT the evaluation of nodes is not parallelized and directly performed after the generation of subproblems.The B&B while-loop is also moved inside the kernel. A pseudo-code for the GPU-BT exploration phase is shown in Algorithm 10.

As discussed in Section 3.2, the kernel must terminate in order to perform global load balancing. This can be achieved by using a trigger mechanism. A global counter is initialized at 0 and reset to 0 before each exploration phase. During the exploration phase this global counter is incremented for each IVM whose interval is empty. At each iteration

all threads check the value of this counter and break out of the while-loop if it is greater than the value of a statically defined threshold `trigger`$= \gamma \times nbIVM$ ($0 \leq \gamma < 1$), i.e. as soon as more than $\gamma \times 100\%$ of IVMs are empty. This means that the tree exploration kernel terminates only if at least `trigger` explorers have finished exploring their interval.

This approach is similar to the static trigger mechanism proposed by [KK94] for solving the 15-puzzle on the CM-2 SIMD computer. A one-to-one mapping between threads and IVMs is used (Algorithm 10, Line 8) and each thread sequentially evaluates generated subproblems (Line 13). As the number of generated subproblems per IVM is variable, this `for`-loop results in thread divergence. An alternative that could be viable for some problems is to map IVMs to warps and parallelize the bounding operation at warp-level. This can be the best choice for some problems, like in [RS10], where the node evaluation is SIMD-parallelizeable at full warp-size.

Depending on problem size and hardware, some parts of IVM can reside in shared memory. For problems of size $n \leq 20$, we decided to load the vectors of length n, `position`, `end` and `schedule` to shared memory. For $n = 20$ and blocksize 128 this amounts to 7 680 B of shared memory per block. On the GTX980 GPU, 96 kB of shared memory are available per SM. Therefore the number of resident blocks is limited to $\left\lfloor \frac{96}{7.68} \right\rfloor = 12$, which corresponds to $\frac{1536}{2048} = 0.75$ occupancy[4].

It is possible to make this choice automatically, querying device properties, problem size and desired upper bound for occupancy. A parameter corresponding to different data placements can be passed to the kernel, which declares and initializes shared memory pointers accordingly. As this version was tested on a Maxwell GPU of compute capability 5.2 only the configuration cited above was used. However, to improve portability an automated mechanism should be implemented[5].

4. For the considered compute capability, the maximum number of resident threads equals 2 048

5. It can not be assumed that the amount of available shared memory per multiprocessor increases from one compute capability to another. For example, the maximum amount of shared memory per multiprocessor is 96 kB for compute capability 5.2, 64 kB for 5.3-6.0 and again 96 kB for 6.1

Algorithm 10 `GPU-backtracking: exploration phase`

```
1:  function EXPLORE
2:      cudaMemset : count_dev ← 0
3:      kernel«<nbIVM threads»>explore(count,trigger)
4:      cudaMemcpy : count_host ← count_dev
5:      return (count_host = T ?)                    ▷ (count_host = T) ⇔ all T IVM empty
6:  end function
7:  kernel EXPLORE(count,trigger)
8:      ivm ← blockIdx.x*blockDim.x+threadIdx.x
9:      //load to shared memory
10:     repeat
11:         if (not-interval-empty(ivm)) then
12:             go-to-next(ivm)
13:             for (k in #generated-nodes(ivm)) do
14:                 bound and prune(ivm)
15:             end for
16:             state[ivm] ← exploring
17:         else
18:             state[ivm] ← empty
19:             atomicIncrement(count);
20:             break;
21:         end if
22:     until (count<trigger)
23: end kernel
```

3.4 Work stealing strategies for GPU-B&B

Both variants, GPU-B&B and GPU-BT use the same load balancing mechanism. The load balancing phase is carried out in two steps. First, in a victim selection step a one-to-one mapping of empty IVMs onto suitable victim IVMs is built. Then, in the `steal` kernel empty IVMs acquire work in parallel from the corresponding victim IVMs. The thief-to-victim mapping must satisfy the following conditions.

To avoid unpredictable behavior, it must guarantee that no victim-IVM is selected twice during the same work stealing phase. Also, only IVMs in the *exploring* state are allowed to be selected as work stealing victims. Moreover, the victim selection should (1) induce minimal overhead, meaning that the mapping must be built in parallel, (2) select victim IVMs whose intervals are likely to contain more work than others, (3) serve a maximum of empty IVMs during each work stealing phase.

3.4.1 Victim Selection policies

Most of the work stealing strategies proposed for IVM-based multi-core B&B are not directly transposable to the synchronous GPU-B&B. An exception is the *ring* strategy.

Indeed, each empty IVM k can independently attempt to steal work from its neighbor $(k-1)$ mod T, avoiding the situation where an IVM is selected twice. In the following we present victim selection strategies which are based on the ring strategy and aim at overcoming its drawbacks. An array `victim-map` of length T is used to encode the thief-to-victim mapping, where `victim-map[i]=j` indicates that IVM i steals from IVM j.

Ring-based strategies

Ring: In this selection-policy an empty IVM $k \in \{1, ..., T\}$ tries to steal a portion of work from IVM $(k-1)\%T$. If the state of IVM $(k-1)\%T$ is *exploring*, then work can be stolen and `victim-map[k]` is set to $(k-1)\%T$. Like for asynchronous work stealing, stealing all but $1/\mathrm{T}^{th}$ of the victim's interval is a more suitable granularity policy than one-half. This topology connects IVMs in a directed ring and its diameter is equal to T. Starting with the entire interval at IVM 0, this strategy requires at least T iterations until all IVMs have acquired work. Despite the fact that the *Ring* strategy is trivially parallelizeable the work distribution process remains inherently sequential, due to queuing of inactive IVMs.

Search: The idea behind this strategy is to perform successive selection attempts with increasing stride, meaning that an empty IVM k attempts to select $(k-1)\%T$, $(k-2)\%T$, ... , $(k-S)\%T$ successively. After each selection attempts all threads must synchronize, therefore a `stride` parameter is added to the selection kernel which is launched S times. In order to avoid multiple selections, a per-IVM boolean `flag` is set to `true` for successfully matched thief-victim pairs. Searching the entire ring ($S = T$) results in excessive overhead, so the value must be fixed at a lower level. For experimental purposes the value is fixed at $S = \left\lfloor \sqrt{T} \right\rfloor$. Following the same reasoning as for the *Ring* strategy, work is stolen with granularity $1/\sqrt{T}$. Using this value the diameter is reduced to $\sqrt{T}$ and it requires at least as many work stealing phases for all IVMs to acquire work.

Large: The experimental results for multi-core B&B have shown that stealing from the largest interval effectively reduces the number of work stealing operations. This requires computing the length of each interval. In order to avoid the costly operation of sorting the IVM-IDs by their corresponding interval-lengths, the mean interval-length is computed prior to the victim selection phase. Having an interval larger than average is added as a criterion for the eligibility of an IVM as a work stealing victim.

Adding this length-criterion increases the probability that no victim is found in the search window of fixed length S. Therefore the parameter S is allowed to float between S_{min} and S_{max}. If more than 10% of IVMs are empty, then S is multiplied by 2, otherwise

S is divided by 2 (if greater than 1). The goal of this approach is to adapt the work stealing (and the associated overhead) to the phase of the algorithm. The idea is to aggressively load balance during the ramp-up and shut-down phases, while reducing overhead during a relatively stable phase where work stealing operations occur only occasionally. As large intervals are selected, work is stolen with $1/2$ granularity.

Adapt: There is no particular reason for starting the search at $k-1 \mod T$. The *Adapt* strategy shifts the beginning of the search window by the current value for S at the beginning of each work stealing phase.

The four work stealing strategies are described in Algorithm 11 and correspond to different choices for a set of parameters.

Algorithm 11 Ring-based victim selection policies for GPU-based work stealing

```
1: switch strategy do
2:     case Ring:
3:         B = 1; S = 0; C = 0;
4:     case Search:
5:         B = 1; S = √T, C = 0;
6:     case Large:
7:         B = 1; 10 < S < T, C = 1;
8:     case Adapt:
9:         B = iter%T; 10 < S < T; C = 1;
10: function SELECTVICTIM(B, S, C)
11:     for (k = B → B+S) do
12:         TRY-SELECT<<<T THREADS>>>(k, victim-map, C,...)
13:     end for
14: end function
15: kernel TRY-SELECT(k, victim-map, C,...)
16:     ivm ← blockIdx.x*blockDim.x + threadIdx.x
17:     if (state[ivm]=empty) then
18:         V ← (ivm-k) mod T
19:         if (state[V]=exploring AND flag[V]= 0 AND length[V]>C*meanLength) then
20:             victim-map[ivm]← V
21:             flag[V]← 1
22:         end if
23:     end if
24: end kernel
```

Hypercube-based strategies

Hypercube. All work stealing strategies presented until now are based on a directed ring-topology. Other topologies are possible. In a 2D-ring or torus topology, for instance, the IDs of $T_1 \times T_2$ IVMs are written as couples (r_1,r_2) $(0 \leq r_i < T_i)$ with IVM (r_1,r_2)

successively trying to steal from its two neighbors $(r_1 - 1 \pmod{T_1}, r_2)$ and $(r_1, r_2 - 1 \pmod{T_2})$.

This can be further generalized, writing an IVM-ID as a m-tuple $(\alpha_m, \alpha_{m-1}, \ldots, \alpha_i, \ldots, \alpha_1)$ $(0 \leq \alpha_i < L_i)$. Connecting all workers whose ID differ in exactly one digit, i.e., that are within Hamming distance 1, a m-dimensional hypercube is obtained. In general the nodal degree in this topology is $\sum_{i=1}^{m}(L_i - 1)$. If we have $\forall i : L_i = p$ (meaning the the IVM-IDs are written in base p) each of the $T = p^m$ IVMs has $m(p-1)$ neighbors.

In the proposed *Hypercube* victim selection strategy all IVMs attempt to select one of its neighbors. Again, this selection is performed iteratively: first all IVMs try to select the neighbor according to the first ID-coordinate, then according to the second, and so on. This assures that no double selection can occur. In Algorithm 12 the pseudo-code for the *Hypercube* selection policy is shown.

Algorithm 12 Victim selection: *Hypercube*

```
1:  function SELECT-HYPERCUBE
2:      for (i : 1 → m) do
3:          for (j : 1 → L_i-1) do
4:              select-hyper<<<T threads>>>(i,j,...)
5:          end for
6:      end for
7:  end function
8:  kernel SELECT-HYPERCUBE(i, j, ...)
9:      ivm ← blockIdx.x*blockDim.x + threadIdx.x
10:     if (state[ivm]=empty) then
11:         %%%                                   ▷ V is an integer ∈ [0, T − 1]
12:         V ← (α_m, ..., (α_i − j) mod L_i, ..., α_1)
13:         if ((state[V]=exploring) ∧ (flag[V]= 0) ∧ (length[V]>meanLength)) then
14:             victim-map[ivm]← V
15:             flag[V]← 1
16:         end if
17:     end if
18: end kernel
```

To make the pseudo-code more readable the operation in Line 12 of Algorithm 12 is written in terms of hypercube ID-coordinates. However, in practice it should be carried out in terms of integer operations. To do this $(\alpha_m, \ldots, \alpha_1)$ with $0 \leq \alpha_i < L_i$ is seen as a number in a number system where the i^{th} position has weight $w_1 = 1, w_i = \prod_{j=1}^{i-1} L_i$. Using these weights w_i each IVM-ID $r = 0, \ldots, T-1$ has a unique representation $r = \sum_{i=1}^{m} \alpha_i w_i$ [Can69] and we have that:

$$\text{The operation } \Big\{ \textbf{return } (\alpha_m, \dots, (\alpha_i - j) \pmod{L_i}, \dots, \alpha_1)$$

$$\text{is equivalent to } \begin{cases} \textbf{return } r - j \times w_i, & \text{if } \frac{r}{w_i} \pmod{L_i} \geq j \\ \textbf{return } r - j \times w_i + L_i \times w_i, & \text{otherwise} \end{cases}$$

For $T = L_m L_{m-1} \cdots L_1$ IVMs a m-dimensional hypercube topology can therefore be defined using two tables: $[L_m, L_{m-1}, \dots, L_1]$ and the weights $[w_m, w_{m-1}, \dots, w_1]$. Although the algorithm is not limited to any particular number of IVM structures, we use only powers of 2 as values for T and we choose the L_i's such that the product $T = L_m L_{m-1} \cdots L_1$ contains the factor 4 as many times as possible but not the factor 2. For example, using $T = 512 = 2^9$ IVMs, a 4-dimensional hypercube topology is defined by the tables $\{L_i\} = [8, 4, 4, 4]$ and $\{w_i\} = [2^6, 2^4, 2^2, 1]$.

3.4.2 Work stealing for multi-GPU-B&B

multi-GPU: In order to extend the presented GPU-B&B algorithm to multiple GPUs sharing the same host it is necessary to deal with inter-GPU load balancing and sharing of the best solution found so far.

In the multi-GPU algorithm several threads execute GPU-B&B asynchronously. Before launching the threads, the number of detectable GPU devices is queried and GPUs are assigned to threads in round-robin fashion. Each thread issues kernel launches and CUDA copy instructions to a different stream. Therefore, the number of used threads can exceed the number of detected devices, although this might not necessarily be useful. All threads run GPU-B&B with a common configuration. In particular, the number of used IVM structures T is identical.

Obviously, static distribution of the interval $[0, n![$ among the GPUs will result in poor performance due to load imbalance. The proposed load balancing scheme for the multi-GPU algorithm uses a hierarchical work stealing approach, meaning that local work stealing (inside each GPU) has a strict priority over inter-GPU work stealing operations. In this approach an inter-thread work stealing attempt will only be initiated by a thread if all its T IVMs are empty. To select another thread, a random victim selection strategy is used.

An inter-thread work stealing operation is defined as for the GPU-accelerated algorithm presented in Subsection 2.4.1: the i^{th} IVM of the thief attempts to steal the right half-interval from IVM i of the victim GPU.

hybrid multi-GPU/multi-CPU: Typically the number of CPU cores is higher than the number of GPUs attached to the host. Thus, the GPU-centric multi-GPU B&B does not exploit the full processing capabilities of the system, as the host CPU remains mostly idle.

In addition to the GPU-controlling threads, CPU-based B&Bthreads can be used. If CPU- and GPU-explorers are used, device-to-host (D2H) and host-to-device (H2D) work stealing operations must be defined. These operations must take into account that a CPU-explorer thread handles a single IVM structure while a GPU explorer handles T IVM structures.

- In a H2D work stealing operation a part of the host's interval is send to the GPU and assigned to IVM 0 of the thief GPU. In order to take into account the heterogeneity of the thief and the victim the granularity policy can be adjusted according to the relative power of the explorers.

  For instance, a thief steals $[(1-\gamma)A + \gamma B, B]$ where $\gamma \in [0,1]$ is set to $1/k$ if the thief is k times more powerful than the victim. We experiment the steal-half and the steal-1/k granularity policies.

- Finally, a D2H work stealing operation is defined as follows: the thief (host) randomly selects one of the victim's IVMs and checks the status of its interval. If the status is *empty* the steal attempt fails. Otherwise, the thief steals the *whole* interval, including the IVM structure. This seems reasonable as this interval represents on average only $1/T^{th}$ of the victim's total amount of work. Moreover, this aims at reducing the overhead for the victim GPU: it uses an asynchronous copy to send the stolen IVM to the thief and sets the status of the corresponding IVM-ID to *empty*.

3.5 Experiments

As in the previous ones all experiments in this section are performed with an initial upper bound equal to the optimal solution, which ensures that B&B explores exactly the critical tree associated with an instance.

3.5.1 Evaluation of Mapping approaches

In this subsection the two mapping schemes, presented in Subsection 3.3.2, are compared to each other in terms of elapsed execution time of the algorithm. The experiments in

this subsection are performed using an NVIDIA Tesla K20m GPU based on the Kepler (GK110) architecture. The compiler is `gcc` version 4.8.3 with optimization level `-O2` and version 6.5.14 of the CUDA Toolkit is used. As the execution model and warp size have not changed in subsequent architectures we believe the results remain valid for more recent architectures. For the evaluation of the mapping schemes the number of IVMs is set to an ad-hoc value of $T = 768$.

Bounding kernel The dynamic remapping scheme for the bounding kernel (Algorithm 7) is referred to as *remap*, the static mapping (Algorithm 6) as *static*. Figure 3.4 shows the total elapsed time for solving instances *Ta021-Ta030*. For both mappings and for each instance it shows the portion of time spent in the kernel `bound`, in the IVM-management kernels (`share`, `goToNext`, `decode` and `prune`) as well as in the remapping phase (for *remap*). As the building of the mapping consumes only 0.9% of computation time, this portion is barely visible in Figure 3.4. Table 3.1 shows total elapsed time as well as the time spent in the different phases of the algorithm as an average over the 10 FSP instances *Ta021-Ta030*.

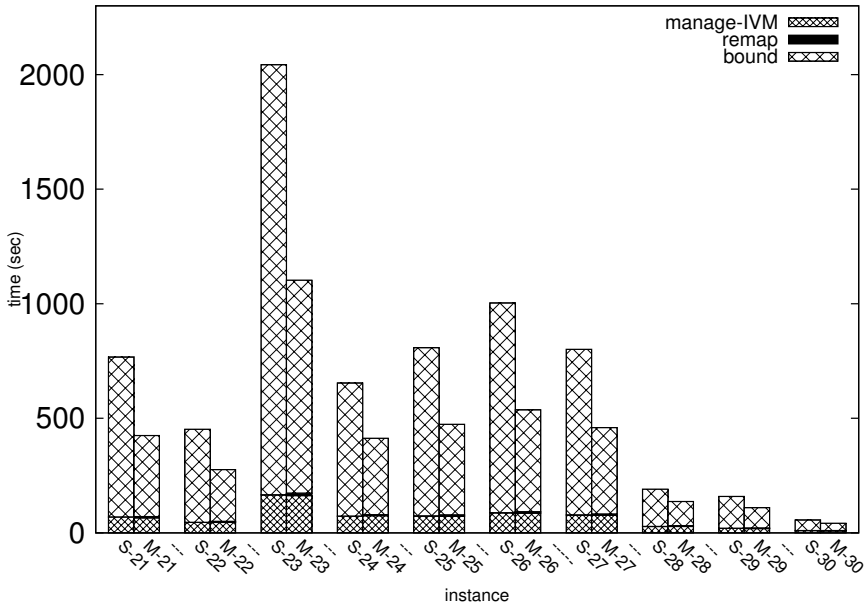


Figure 3.4: Execution time for instances *Ta021-Ta030* for thread-data mappings *static* (S) and *remap* (M) for the kernel `bound`. All results are for one Nvidia K20c GPU (Kepler) using $T = 768$ IVM.

The compacted mapping *remap* is clearly advantageous as it reduces the average time spent in the `bound` kernel by a factor 1.9×. As the bounding operation amounts for more

Table 3.1: Average elapsed time (in seconds) and average repartition of execution time among bounding, IVM management and remapping phases. Average taken over instances *Ta021-Ta030*.

Mapping	elapsed walltime sec	bound sec	bound %	manage sec	manage %	remap sec	remap %
static	696.4	632.9	89.4	63.5	10.6	0.0	0.0
remap	395.7	329.1	82.0	63.4	17.1	3.4	0.9

than 80% of the total execution time, the latter decreases by a factor 1.7×. The overhead induced by compacting the mapping at each iteration is largely compensated by these performance gains. Indeed, thanks to the parallelization of this phase using the parallel prefix sum, the remapping operation amounts for less than 1% of the elapsed time.

Using the more compact mapping *remap* instead of *static* improves the control flow efficiency[6] (CFE) of the kernel. For *static* the average CFE is 0.43, meaning that for an executed instruction on average more than half of the execution slots are wasted. For the mapping *remap* the average CFE is 0.83 - the launched warps are used almost twice as efficiently. The number of warps launched at each kernel call is 960 for mapping *static*, which exceeds theoretical maximum of $13 \times 64 = 832$ resident warps for the K20m. The average number of warps launched with mapping *remap* is 300 (average per kernel call and per instance), the average maximum (per instance) being 825 warps and the minimum 4.

IVM management kernels. In this subsection the two mapping schemes for the IVM-management kernels are evaluated and compared to each other. The kernels concerned by these mapping schemes are the IVM-management kernels (`share`, `goToNext`, `decode` and `prune`). Figure 3.5 shows the time spent for completing the exploration with both mapping schemes. Both, version *one-thread-per-IVM* ($M1$) and version *one-warp-per-IVM* ($M2$) use the same bounding kernel (with remapping). Although the time spent managing the IVM structures is moderate compared to the bounding operation, the mapping $M2$ allows a reduction of the total execution time by a factor 1.1× compared to the mapping $M1$. With respect to $M1$, mapping $M2$ decreases the share of IVM-management operations from 18% to 7.5%. Table 3.2 shows the average duration per call of the kernels `bound` (in msec), `goToNext` and `decode` (in μsec) and their respective share of the elapsed time (in %). The kernels `prune` and `share` amount for at less than 2% of total execution time, so they are not evaluated.

6. defined as the ratio between the number of active threads (not predicated off) and the maximum number of threads per warp for each executed instruction ($CFE = \frac{\texttt{not\_predicated\_off\_thread\_inst\_executed}}{\texttt{32*inst\_executed}}$)

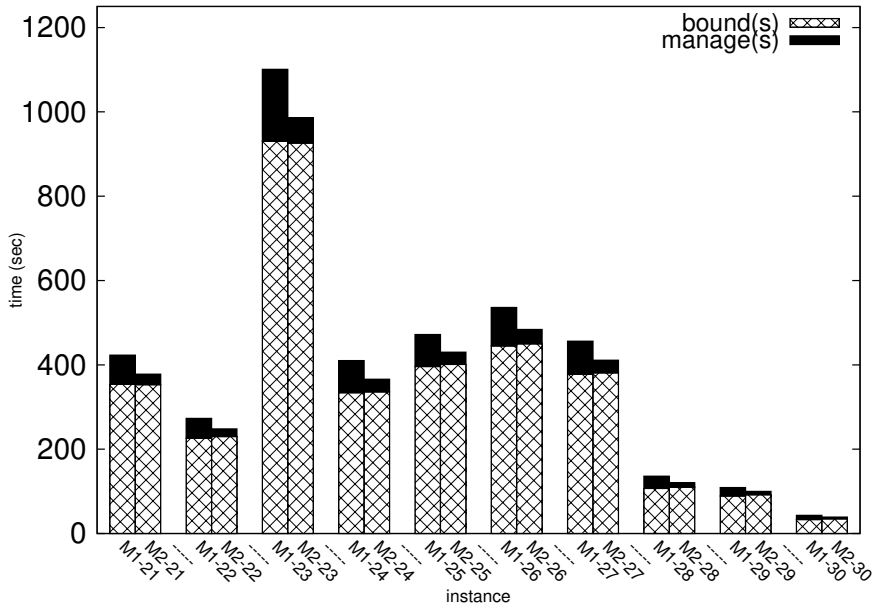


Figure 3.5: Execution time for instances *Ta021-Ta030* for different mapping choices in IVM-management kernels.

The mapping *M*2 allows to use the supplementary lanes for efficient loading of the IVM structures into shared memory. In order to dissociate the impact of shared memory usage from the impact of remapping, the profiling of mapping *M*2 is performed with and without shared memory usage.

Table 3.2: Duration of different kernels per call (in μsec or msec), percentage of total elapsed time (%) and instruction replay overhead (IRO%), total execution time of GPU-B&B. Average values for instances *Ta021-Ta030*.

	goToNext			**decode**			**bound**		**elapsed**
Mapping	**μsec**	**%**	**IRO%**	**μsec**	**%**	**IRO%**	**msec**	**%**	**sec**
1 thread/IVM	380	10.0	40.6	168	4.4	40.3	3.07	82.0	395.7
1 warp/IVM	130	4.0	14.0	94	2.8	14.7	3.07	91.1	364.2
1 warp/IVM (shared)	85	2.6	7.9	79	2.4	12.4	3.06	92.5	356.6

Table 3.2 also shows the instruction replay overhead (IRO%)[7], which is a measure for instruction serialization (due to memory operations, like cache misses). These results show that the fact of spacing the mapping to 1 warp=1 IVM also substantially improves

7. defined as $IRO\% = 100\% \times \frac{\texttt{instructions\_issued}-\texttt{instructions\_executed}}{\texttt{instructions\_issued}}$

the memory access pattern. It should be noted that the control flow efficiency drops from a poor average 0.22 for $M1$ to 0.03 $\approx 1/32$ for $M2$ - as intended. Table 3.3 shows, for the different kernels, the number of branch instructions executed (per call average) and the number of branches that are evaluated differently across a warp. The results show that, as intended, undesired thread divergence completely disappears. Only instance *Ta022* is evaluated as one instance sufficiently illustrates the behavior. The

Table 3.3: Per-call average of branch instructions executed and diverging branches (incremented by one per branch evaluated differently across a warp). Test-case: FSP instance *Ta022*.

kernel	goToNext		decode		share		prune	
Mapping	**branch**	**diverge**	**branch**	**diverge**	**branch**	**diverge**	**branch**	**diverge**
1 thread/IVM (M1)	11 592	802	5 875	860	851	15	404	121
1 warp/IVM (M2)	59 921	1 536	62 020	768	3 655	0	3 131	768
		=2×#IVM		=#IVM				=#IVM

`divergent_branch` counter indicates that the average number of diverging branches is a multiple of the number of IVMs. Indeed, the counter increments by one at the instruction `if(thId%32 == 0)` (Algorithm 9, line 6) which masks all but the leading thread in each warp. However, as the remaining 31 lanes of the warp are simply waiting for lane 0 to complete, no significant serialization of instructions occurs. Besides showing that the spaced mapping $M2$ is better adapted to the IVM-management kernels, the results presented in this subsection illustrate that performance metrics for thread divergence or control flow must be interpreted very carefully.

3.5.2 Evaluation of Work Stealing strategies

For the evaluation of work stealing strategies, a first series of experiments is performed using a single NVIDIA Tesla K20m GPU, version 5.0.35 of the CUDA Toolkit and only FSP instances *Ta021-Ta030*. The number of used IVMs is fixed at $T = 768$.

Table 3.4 shows the exploration time for work stealing strategies *ring*, *search*, *large* and *adapt*. For comparison Table 3.4 also shows the execution time obtained with MC-B&B using 32 threads (2xE5-2630v3). For all instances, best performances are achieved with the *adapt* strategy, which allows to complete the exploration on average 4.2 times faster than MC-B&B. Comparing the different work stealing strategies, one can see that the *adapt* strategy is less sensitive to varying instance-sizes and shapes. *Adapt* provides a relative speedup of $3.3 - 4.5\times$ over MC-B&B, while the spread $1.0 - 3.6\times$ is much larger for the *ring* strategy.

Table 3.4: Exploration time (in seconds) for solving flowshop instances *Ta021-Ta030*. Using $T = 768$ IVM on Kepler K20m GPU. For comparison, the execution time of MC-B&B using 32 threads (2xE5-2630v3) with *random*-1/2 work stealing is shown.

		MC-B&B	*ring*		*search*		*large*		*adapt*	
Inst.	$\times 10^6$ nodes	Time	Time	Rate	Time	Rate	Time	Rate	Time	Rate
21	41.4	1 062	386	2.8	338	3.1	280	3.8	250	4.3
22	22.1	526	247	2.1	229	2.3	146	3.6	129	4.1
23	140.8	3 631	1 002	3.6	934	3.9	915	4.0	813	4.5
24	40.1	884	359	2.5	254	3.5	255	3.5	219	4.0
25	41.4	1 073	431	2.5	327	3.3	280	3.8	250	4.3
26	71.4	1 547	459	3.4	443	3.5	429	3.6	384	4.0
27	57.1	1 294	404	3.2	370	2.6	336	3.9	301	4.3
28	8.1	202	120	1.7	79	1.9	59	3.4	52	3.9
29	6.8	164	95	1.7	88	2.5	50	3.3	42	3.9
30	1.6	40	39	1.0	36	1.1	20	2.0	12	3.3
Avg	**43.1**	**1 043**	**354**	**2.9**	**310**	**3.4**	**277**	**3.8**	**245**	**4.2**

We propose to use the following metric for measuring the efficiency of a load balancing scheme for IVM-based synchronous parallel B&B.

$$\text{IVM-efficiency} = 100\% \times \frac{\#\text{decomposed nodes}}{\#\text{iterations} \times \text{T}}$$

As an average over all iterations and all IVM structures, *IVM-efficiency* indicates the share of IVMs that perform useful work. It is computed as a ratio of the ideal number of iterations ($\frac{\#\text{Nodes}}{T}$) over the number of performed iterations, expressed as a percentage. Table 3.5 shows the IVM-efficiency for the four strategies *ring*, *search*, *large* and *adapt*. The results show that the *adapt* strategy is close to the optimal case where IVM-efficiency= 100%. On average only 2.5% of the $T = 768$ IVMs are either empty or initializing. This is a good indicator for the scalability of the approach, meaning that the *adapt* work stealing strategy is capable of handling a larger number of IVMs. This is particularly important because the number of cores in successive GPU generations continues to increase.

Table 3.5: Average percentage of IVMs in *exploring* state (IVM-Efficiency)

	Ta021	Ta022	Ta023	Ta024	Ta025	Ta026	Ta027	Ta028	Ta029	Ta030	**Avg**
Ring	52.0	38.4	74.5	48.2	45.6	71.0	63.9	24.6	29.1	14.8	**46.2**
Search	67.4	46.3	84.2	84.2	73.8	80.4	79.0	53.8	36.7	19.9	**62.5**
Large	93.3	92.9	93.4	93.2	93.3	93.2	93.4	91.5	90.4	76.4	**91.1**
Adapt	99.4	98.9	99.8	99.4	99.3	99.7	99.5	96.8	96.8	85.3	**97.5**

Next, the overhead induced by load balancing is evaluated. Figure 3.6 shows, for the four strategies, the average time spent in different phases of the algorithm. In all cases the

bounding phase consumes $> 85\%$ of the total execution time. Using the *ring* strategy, on average 327 seconds are spent in the bounding kernel, against 226 seconds for the *adapt* strategy. This corresponds to the increased efficiency of that kernel (Table 3.5). Compared to *ring*, the *search*-window of length $S = 27$ significantly improves IVM-efficiency – at the cost of spending $\approx 3\%$ instead of $\approx 0.1\%$ in victim-selection. The *large* strategy further improves the work load balance, but victim-selection amounts for $\approx 9\%$ as the average value for the auto-tuned parameter S increases to 110. Extending *large* to *adapt* only slightly improves the load balancing, but, more importantly, as the average S decreases to 15, the cost of victim selection decreases to $< 2\%$. Using a fixed number of IVMs ($T = 768$), the *adapt* strategy clearly outperforms the *ring*, *search* and *large* strategies. Considering the low overhead of this strategy there is no reason to believe that this conclusion changes for larger values of T.

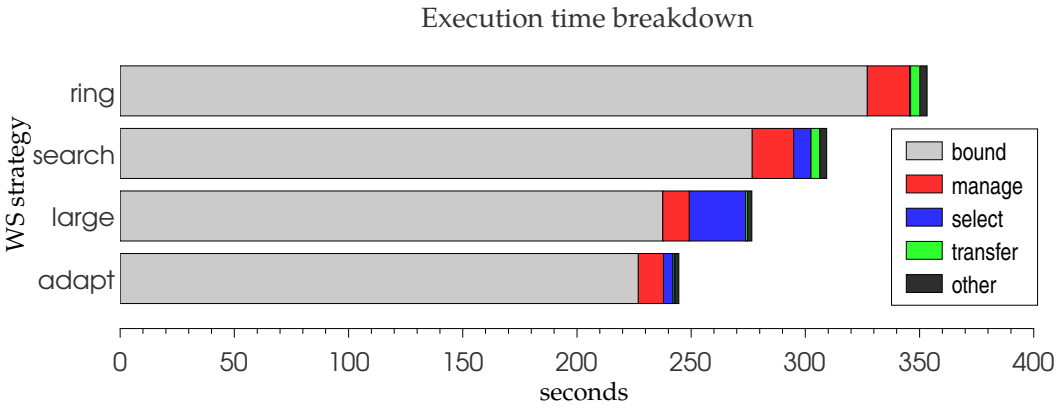
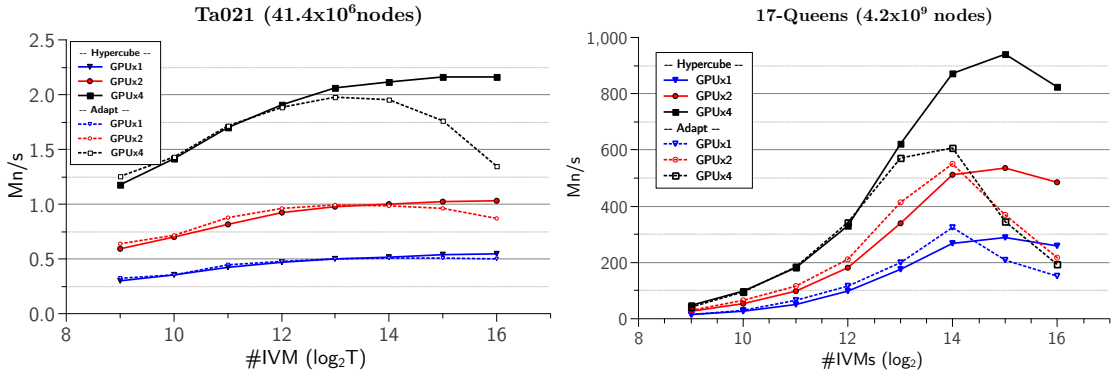


Figure 3.6: Average elapsed time for solving instances *Ta021-Ta030* and its repartition among different phases of the algorithm

3.5.3 Scalability analysis

In this subsection the scalability of the proposed GPU-B&B and GPU-BT algorithms is evaluated. By that we mean in particular the scalability of work stealing mechanism, i. e. their ability to keep IVMs busy as the number of IVMs and GPUs increases. In the following, only the *adapt* and *hypercube* strategies are evaluated and all three test-cases are used. All experiments are run on a computer composed of two 8-core E5-2630v3 processors, and four GeForce GTX 980 GPUs. The four Maxwell GPUs, based on the GM204 architecture are of compute capability 5.2. and version 7.5 of the CUDA Toolkit is used.

The analysis of the scalability of both strategies serves at the same time as experimental calibration of the number of IVMs used per GPU (T). The number of used IVM structures per GPU should not be tuned independently from the used load balancing mechanism. Indeed, there is no point in adding more IVMs if the load balancing mechanism fails to keep them busy. One the other hand, if a work stealing strategy allows to use more IVMs efficiently, increasing T may improve the performance of the algorithm. Moreover, the workload, i. e. the size and shape of the explored B&B tree and the granularity of the problem should be taken into account, as these factors strongly influence the performance of a given work stealing approach.



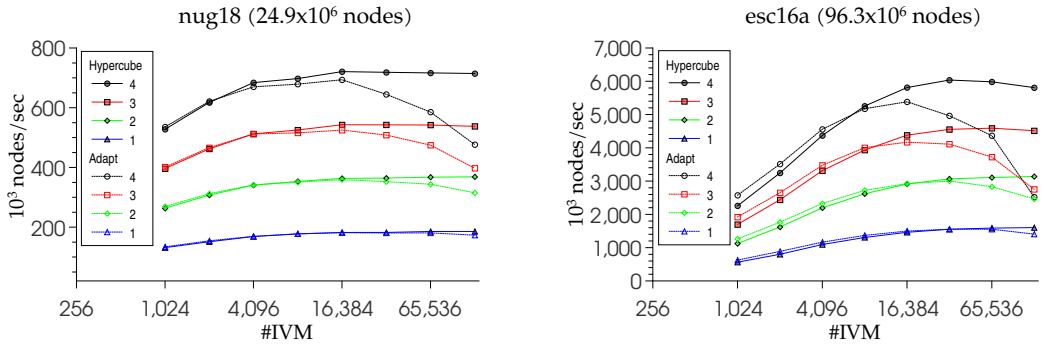
(a) FSP - Ta021 (initialized at optimal solution), 41.4×10^6 nodes

(b) 17-Queens, 4.2×10^9 nodes

Figure 3.7: Node processing rate (in Mn/s) for different values of T (#IVMs) and 1, 2 and 4 GPUs. The dotted lines show the results for the *adapt* strategy and the solid lines for the *hypercube* strategy

Figures 3.7 and 3.8 show the node processing rates (respectively in Mn/s and kn/s) achieved for different values of T ($T = 2^n, n = 9, 10, \ldots, 15$) and the *adapt* and *hypercube* work stealing strategies. The results are shown for 1, 2 and 4 GPUs without using the CPU cores for exploration. The results on the left-hand side (Figure 3.7a) are obtained for FSP instance *Ta021*, using the 2-level GPU-B&B algorithm. On the right-hand side (Figure 3.7b) the achieved node processing rates are shown for the 17-Queens problem, using the 1-level GPU-BT algorithm. As an ad-hoc value, the work stealing trigger for GPU-BT is set to $T/10$. In Figure 3.8 two QAP instances with different node evaluation costs are shown (*nug18* and *esc16a*).

These problem instances are used because they are large enough to justify the use of a multi-GPU system and small enough to be solved repeatedly to collect experimental data. For example, instance *Ta021* requires over 20 minutes of processing using the 16 CPU-threads and about 90 seconds using a single GPU. As one can see in Figure 3.7,



(a) QAP: *nug18* (24.9×10^6 nodes) (b) QAP *esc16a* (94.2×10^6 nodes)

Figure 3.8: Node processing rate (in Mn/s) for different values of T (#IVMs) and 1, 2 and 4 GPUs. The dotted lines show the results for the *adapt* strategy and the solid lines for the *hypercube* strategy. QAP instances *nug18* and *esc16a*.

both work stealing strategies allow to achieve similar performances for a single GPU, especially when a small number of IVMs is used ($T < 4096 (= 2^{12})$).

For small values of T the *adapt* strategy allows slightly better performances than *hypercube*, in particular for instance 17-Queens and *esc16a*, i. e. the more fine-grained of the four test-instances. However, as the number of IVMs increases, the *hypercube* strategy clearly outperforms the *adapt* strategy. As the number of GPUs increases (meaning that the total number of IVMs increases) this effect is even more visible.

For both work stealing strategies and algorithm variants, the node processing rates increase according to the number of IVMs up to $T = 2^{14}$. However, when the *adapt* strategy is used the processing speed increases at a lower rate for $T \geq 2^{12}$ and decreases when $T > 2^{14}$ IVMs are used. This can be explained by the fact that the cost of the victim selection increases significantly as the increasing number of IVMs causes the strategy to search in larger windows. This is not the case in the *hypercube*-based strategy which contacts a constant number of IVMs to select a work stealing victim.

Figures 3.7 and 3.8 show that the *hypercube* strategy is better suited for the multi-GPU-B&B algorithm, and especially for GPU-BT. Using *hypercube*, during the solution counting of 17-Queens almost 1 billion (10^9) nodes per second are decomposed when using 4 GPUs and 32 768 IVMs per GPU, i. e. 131 072 IVMs in total. This is about 50% more than the best rate achieved with the *adapt* strategy which stagnates at 600 Mn/s for 16 384 IVMs.

Calibration of work stealing trigger: For GPU-BT a threshold for the static work stealing trigger must be defined. In the previous paragraph we used $T/10$ as an ad-hoc value

for this threshold, meaning that an intra-device work stealing phase occurs only if 10% of the IVMs have empty intervals. The number of IVMs per GPU is set to $T = 32\,768$. Using 17-Queens for calibration, Figure 3.9 shows the variation of the node processing speed according to the value of the work stealing trigger. For finding all 95 815 104 valid configurations of the 17-Queens problem, a total of 4.2×10^9 valid partial board configurations is decomposed into smaller subproblems. For inter-GPU load balancing a *random*-1/2 work stealing strategy is used.

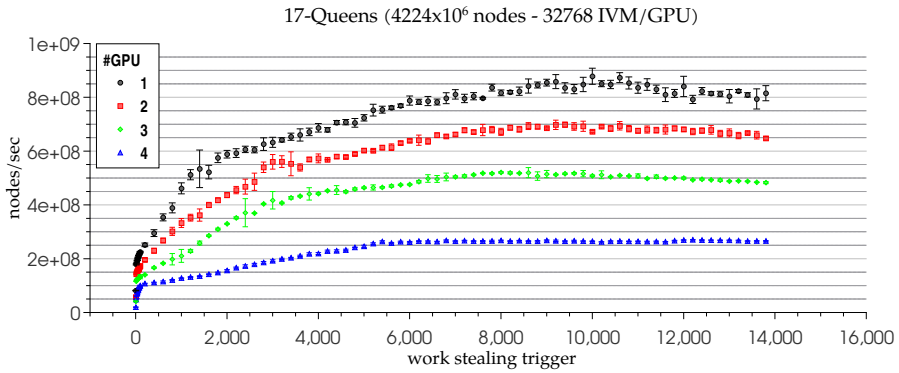


Figure 3.9: Performance of multi-GPU-BT algorithm for 17-Queens problem, for varying work stealing threshold and one to four GPUs. T=32 768 IVM, dots show average over 5 runs, error bars show standard deviation.

One can see that the performance depends strongly on the calibration of the triggering mechanism. The use of this mechanism is clearly beneficial compared to a no-trigger algorithm. Indeed, a trigger value equal to 0 corresponds to a trigger-less version of the algorithm, where a load balancing phase is launched at each iteration. However, careful calibration is necessary, as a too large value deteriorates the algorithm's performance. For both test cases and a single GPU a factor 10 can be gained by using the triggering mechanism and setting is to a suitable value.

The results shown in Figure 3.9 are averaged values over 5 runs and error bars show standard deviation. In both figures one can notice that the results obtained for 3 and especially 4 GPUs are noisy. Analysis of the execution time breakdown reveals that the variation in execution time corresponds mainly to a variation in waiting time. This is due to the use of the random strategy in combination with the trigger mechanism. The random victim selection for inter-GPU work stealing naturally introduces some randomness in the obtained results. However, this effect is amplified by the use of the trigger-mechanism. While a GPU is in the midst of a node expansion phase it cannot respond to incoming requests. Nevertheless, this GPU can still be selected as a work

stealing victim, and the thief thread needs to wait until the selected victim returns to the CPU.

To overcome this issue it would be necessary to send a signal to the victim device, requesting an interruption of the current kernel execution. However, to the best of our knowledge, CUDA currently provides no guarantee for data coherence between a running kernel and concurrent data transfers. These results demonstrate, on the one hand, that the triggering mechanism can be used to accelerate the exploration process. On the other hand they show that it requires careful tuning, especially in a multi-GPU setup.

Ideally, the need for tuning the trigger threshold should be avoided by adjusting the trigger value dynamically, based on the duration of exploration and load balancing phases, as in [KK94]. In all the following experiments the work stealing threshold is set to $\frac{T}{3}$.

3.5.4 Multi-GPU-B&B performance evaluation

In this subsection the scalability with the number of GPUs is evaluated and the performance of the multi-GPU-B&B and BT algorithms is compared to the (GPU-accelerated) MC-B&B algorithm presented in the previous chapter. The three test-cases are used for evaluation.

FSP: Figure 3.10 shows execution time for the largest 20-jobs-on-20-machines instance *Ta023* using 1, 2 and 4 GPUs with varying number of IVMs per GPU. Notice that both axis are in log scale. One can see that time decreases linearly with the number of GPUs used. Using 512 IVMs the instance is solved in about 512 seconds on a single GPU and in about 128 seconds on 4 GPUs. Using 64 times as many IVMs, the resolution time on a single GPU is about 256 seconds on one, and 64 seconds on 4 GPUs. Furthermore, using more than 8 192 IVMs per GPU provides only marginal improvement.

In Table 3.6 we report the achieved node processing rates for the ten instances *Ta021–Ta030* (in kn/s). In order to better show the impact of the instance-size, the instances are sorted in an increasing order according to the number of nodes explored (shown in the second column). Device initialization (e. g. cudaMalloc, cudaSetDevice, cudaEnablePeerAccess calls) amounts to about 0.5 seconds for one GPU and surprisingly lasts up to 4 seconds when done in parallel by four POSIX threads using one device each. This depends on the driver version, toolkit version, the operating system and other factors, which is why device initialization time is excluded from the reported results.

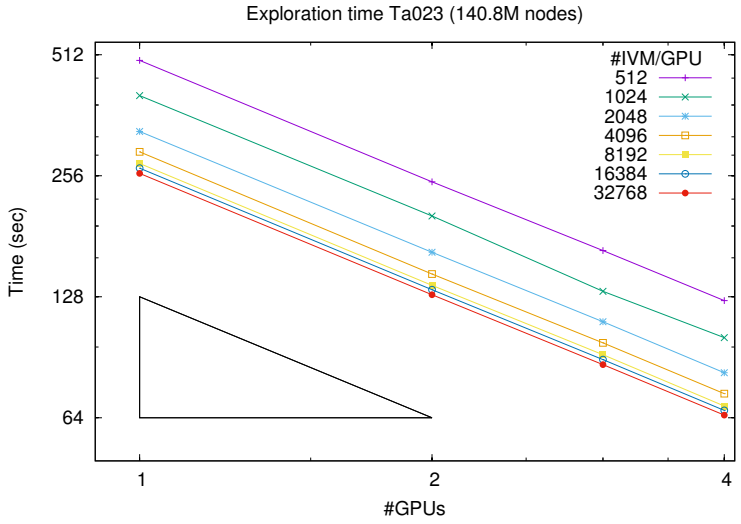


Figure 3.10: FSP instance *Ta023* (140.8×10^6 nodes, 21.5 hours sequential execution time): Execution time for varying number of GPUs and IVMs per GPU.

All results are obtained using the *hypercube* strategy for intra-GPU work stealing, *random*-1/2 for inter-GPU work stealing and 16 384 IVMs per GPU. Table 3.6 also shows the processing speed achieved by MC-B&B using 32 threads on two 8-core E5-2630v3 CPUs, with and without GPU-acceleration. According to the experimental results reported in Chapter 2, this multi-core version is 18×–20× faster than its sequential counterpart.

For example, when solving FSP instance *Ta030*, the multi-core algorithm decomposes 40.5 kn/s without GPU-acceleration and 939 kn/s when 4 GPUs are used to accelerate the bounding operation (as in Chapter 2, using $M = 600$ IVMs per CPU thread). GPU-B&B achieves 556 kn/s using a single GPU and 1737 kn/s using 4 GPUs, which is 3.1 times more than using a single device. All results for the GPU algorithm are averaged over three runs.

For the ten flowshop instances *Ta021-Ta030* the 4-GPU-B&B allows to decompose on average 2178 kn/s which is 52 times more than its multi-core counterpart, 4 times more than the single-GPU algorithm and about 1 000× more than the sequential version of MC-B&B algorithm. Compared to the GPU-accelerated MC-B&B algorithm presented in Chapter 2, GPU-B&B is on average 1.7 times faster, exploiting an equivalent number of GPUs on the same platform.

For all FSP instances, except the smallest (which lasts less than 1 second using 4 GPUs), the algorithm achieves near-linear speedup on up to 4 GPUs. In the 2-level variant of the algorithm each launch of the bounding kernel is configured with a different block-size,

according to the current load. Using multiple GPUs changes the kernel configurations in a way which is beyond the control of the programmer. The effect may, in some cases be beneficial, which can explain the slightly super-linear speed-ups which can be observed when comparing the 4-GPU and single-GPU execution times.

Solving FSP instance *Ta022* using 4 GPUs, the algorithm spends 87.6% of the 9.9 seconds execution time in the bounding phase, 1.8% in intra-device WS, 0.7% of the time waiting for work on the inter-device level, and about 10% in the selection, reduction, remapping and pruning kernels.

Table 3.6: Node processing rates (in kn/s) obtained when solving **FSP** instances *Ta021–Ta030*, using one to four GPUs, work stealing strategy: *Hypercube/Random*. For comparison the node processing speed obtained with the 32-threaded MC-B&B and GMC-B&B (*random*-1/2 work stealing) are shown.

Inst	#nodes	MC-B&B	GMC-B&B	GPUx1	GPUx2		GPUx3		GPUx4	
	$\times 10^6$	kn/s	kn/s	kn/s	kn/s	$\frac{T_{1GPU}}{T_{2GPU}}$	kn/s	$\frac{T_{1GPU}}{T_{3GPU}}$	kn/s	$\frac{T_{1GPU}}{T_{4GPU}}$
30	1.6	40.5	939	556	1 042	1.9	1 479	2.7	1 739	3.1
29	6.8	41.3	1 275	556	1 099	2.0	1 643	3.0	2 145	3.9
28	8.1	39.9	1 213	533	1 032	1.9	1 555	2.9	2 027	3.8
22	22.1	42.0	1 325	532	1 066	2.0	1 597	3.0	2 206	4.0
24	40.1	45.3	1 465	594	1 174	2.0	1 748	2.9	2 420	4.1
21	41.4	39.0	1 292	523	1 011	1.9	1 532	2.9	2 117	4.0
25	41.4	38.6	1 230	489	958	2.0	1 442	3.0	1 975	4.0
27	57.1	44.1	1 511	610	1 211	2.0	1 825	3.0	2 423	4.0
26	71.3	46.1	1 438	578	1 142	2.0	1 711	3.0	2 314	4.0
23	140.8	38.8	1 302	536	1 029	1.9	1 534	2.9	2 178	4.1
AVG	**43.1**	**41.6**	**1 299**	**551**	**1 077**	**2.0**	**1 607**	**2.9**	**2 155**	**3.9**

QAP: Table 3.7 shows the node processing rates obtained when solving QAP instances *nug16a*, *nug18*, *nug20*, *had20* and *scr20* using one to four GPUs. Like for FSP, the processing speed of the 32-threaded MC-B&B algorithm is shown.

Comparing these results to those obtained for FSP, one can notice that the acceleration factors with respect to the MC-B&B algorithm are significantly less important. For example, solving *nug20* MC-B&B reaches 83 kn/s while GPU-B&B decomposes 118 kn/s using a single device and 472 kn/s using 4 GPUs. Like for FSP, GPU-B&B scales linearly with the number of GPUs (up to 4 devices). However, the relative speedup over MC-B&B using 32 threads is only 1.4× for a single GPU and 5.6× using four GPUs, i.e. about 10 times less than for FSP.

For the six considered QAP instance, GPU-B&B explores the tree on average 5.3 times faster than the 32-threaded MC-B&B. Interestingly, this is only slightly more than the acceleration factor obtained by offloading the computation of bounds to the GPU. For all

Table 3.7: Node processing rates (in kn/s) obtained when solving **QAP** instances *had20*, *scr20*, *nug20*, *nug18*, *nug16a*, *esc16c*, using one to four GPUs, work stealing strategy: *Hypercube/Random*. For comparison the node processing speed obtained with the 32-threaded MC-B&B and GMC-B&B (*random*-1/2 work stealing) are shown.

Inst	#nodes	MC-B&B	GMC-B&B	GPUx1	GPUx2		GPUx3		GPUx4	
	$\times 10^6$	kn/s	kn/s	kn/s	kn/s	$\frac{T_{1GPU}}{T_{2GPU}}$	kn/s	$\frac{T_{1GPU}}{T_{3GPU}}$	kn/s	$\frac{T_{1GPU}}{T_{4GPU}}$
had20	69.9	64	311	81	163	2.0	244	3.0	324	4.0
scr20	25.3	68	376	109	217	2.0	324	3.0	431	3.9
nug20	362.6	83	469	132	263	2.0	395	3.0	526	4.0
nug18	25.0	111	638	182	364	2.0	542	3.0	720	3.9
nug16a	0.84	153	819	248	483	2.0	660	2.7	825	3.3
esc16c	356.0	675	5 473	1 597	3 099	1.9	4 608	3.8	6 122	3.8
AVG	**139.9**	**192**	**1 348**	**392**	**765**	**2.0**	**1 129**	**2.9**	**1 491**	**3.8**

tested QAP instances, GPU-B&B outperforms GMC-B&B, but the margin is much smaller than the 1.7× acceleration observed for FSP. Indeed, using the GPU-accelerated MC-B&B algorithm on average 1348 kn/s are decomposed, against 1491 kn/s for GPU-B&B, i.e. on average GPU-B&B is *only* about 10% faster than the GPU-accelerated version of MC-B&B.

Examination of the execution time breakdown for the two small instances *Ta030* and *nug16a*, shown in Table 3.8, illustrates and explains this problem-dependent discrepancy. In terms of decomposed nodes, the tree associated with FSP instance *Ta030* is about twice as large as the one associated with *nug16a*. However, for both instances the exploration time with GPU-B&B is approximately the same, divided in approximately the same proportions between bounding, IVM-management and load balancing operations. In other words, the node processing rate for the FSP instance is about twice as high as for QAP - despite the fact that a sequential node evaluation on CPU is more costly for FSP. As a consequence of the very efficient acceleration of the FSP bounding operation, the algorithm spends proportionally more time managing the IVM structures and balancing the workload, while for the QAP this part remains relatively small and/or can be efficiently hidden by overlapping computations. Indeed, as one can see in Table 3.8, for *Ta030* GMC-B&B algorithm spends about 50% of time handling and sharing subproblems, while the share is only 20% for *nug16a*. As GPU-B&B essentially accelerates these parts of the algorithm by implementing the entire algorithm on the device, solving FSP allows a better acceleration than QAP. It should be noted that the cost of transferring subproblems to the device is rather negligible, and that the acceleration provided by GPU-B&B over GMC-B&B is a result of massively parallelizing a larger portion of the code.

***n*-Queens:** Table 3.9 reports the achieved node processing rates (in Mn/s) for the n-Queens (n = 15-19) problem counting the total number of solutions. As shown in the

Table 3.8: Execution time breakdown for FSP instance $Ta030$ and QAP instance $nug16a$ using GPU-B&B and GPU-accelerated MC-B&B.

	Instance	*Ta030*	*nug16a*
Algorithm	#Nodes	1 648 102	841 732
GPU-B&B	T_{total}	1.04	1.02
	Average T_{bound}/thread	0.89	0.93
	Average T_{manage}/thread	**0.06**	**0.05**
	Average $T_{loadbal}$/thread	0.08	0.04
GMC-B&B	T_{total}	1.68	1.04
	Average T_{bound}/thread	0.85	0.81
	Average T_{manage}/thread	**0.72**	**0.18**
	Average $T_{loadbal}$/thread	0.11	0.05

second column of Table 3.9 the size of explored tree grows exponentially according to the instance size n. Results for $n < 15$ are not shown as these instances are solved within a fraction of a second. For $n > 19$ and the execution time exceeds the imposed time limit of 30 minutes using a single GPU.

In order to compare the performance of our GPU algorithm to CPU-based performances, Table 3.9 also shows the processing rates obtained a highly optimized sequential n-Queens algorithm [Som] using bit patterns, similar to the n-Queens program presented in [Ric97]. This algorithm is the fastest sequential algorithm for n-Queens we were able to find in the literature. For $n = 15 - 18$ this algorithm decomposes on average 83.7 Mn/s which is approximately $35 - 70\times$ faster than our sequential IVM-based version for n-Queens[8]. Using 32 threads MC-B&B achieves about 30 Mn/s, i. e. $30\times$ less than GPU-BT. For large n-Queens instances ($n = 18, 19$) our 4-GPU-BT algorithm is capable of decomposing up to 1 billion nodes per second, finding all solutions to the 19-Queens problem within 4 minutes.

Profiling shows that the work stealing approach generates small overhead: For larger instances the portion of time spend in work stealing is lower, as explorers run out of work less frequently.

3.5.5 Hybrid CPU-multi-GPU-B&B

To conclude this experimental section, the efficiency of the hybrid multi-core/multi-GPU-B&B algorithm is evaluated. As shown by the experimental results reported in the

8. The bitset-based n-Queens implementation uses bit-level parallelism to evaluate nodes with $O(1)$ computational complexity. Our IVM-based n-Queens implementation evaluates nodes with complexity $O(n)$.

Table 3.9: Node processing rates (in Mn/s) obtained when solving n-**Queens** ($n = 15-19$) using one to four GPUs, work stealing strategy: *hypercube+random*.

Inst	#nodes	CPU	GPUx1	GPUx2		GPUx3		GPUx4	
	$\times 10^6$	Mn/s	Mn/s	Mn/s	$\frac{T_{1GPU}}{T_{2GPU}}$	Mn/s	$\frac{T_{1GPU}}{T_{3GPU}}$	Mn/s	$\frac{T_{1GPU}}{T_{4GPU}}$
15-Queens	91	63.4[1)]	232	364	1.6	397	1.7	437	1.9
16-Queens	563	86.4[1)]	286	497	1.7	633	2.2	710	2.5
17-Queens	4 224	95.1[1)]	290	555	1.9	748	2.6	952	3.3
18-Queens	29 350	89.8[1)]	281	567	2.0	779	2.8	999	3.6
19-Queens	242 419	n/a	274	522	1.9	770	2.8	1 040	3.8
AVG	**55 329**	**83.7**[1)]	**273**	**501**	**1.8**	**665**	**2.4**	**827**	**3.0**

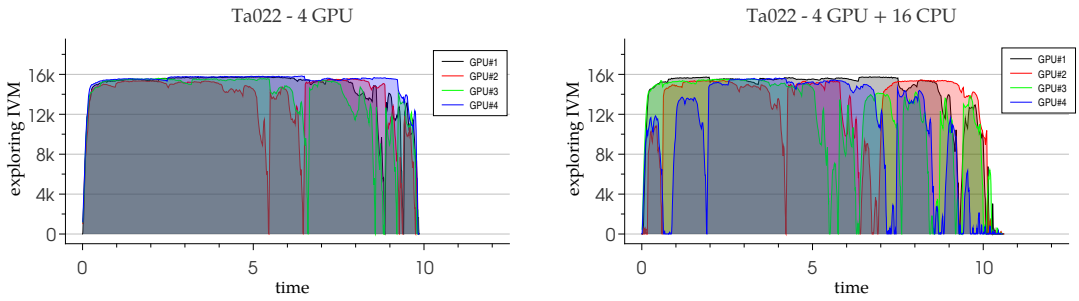
[1)] sequential bitset-based backtracking algorithm (Jeff Somers[Som], on E5-2630v3, gcc 15.0).

previous sections, a single GPU can be used to process about 0.5 Mn/s when solving FSP instances of the group *Ta021–Ta030*. This rate is approximately 250 times higher than the processing rate of a sequential CPU-based algorithm. Hence there is little potential for accelerating the 4-GPU algorithm by using the 16 available CPU-cores in addition to the GPUs. However, in the perspective of integrating the GPU-based B&B algorithm in a larger hybrid cluster-system, the study of this smaller hybrid setup may provide useful insights.

The obtained experimental results show that the proposed approach for the hybrid multi-core/multi-GPU algorithm raises granularity issues. For host-to-device work stealing operations, we have tested both granularity policies described in Subsection 3.4.2. Besides the *steal*-1/2, we tested the policy where a CPU-based work stealing victim keeps only the $1/200^{th}$ part of its interval when the thief is a GPU. For both granularity policies our experiments show that the hybrid algorithm is on average slightly *slower* than its multi-GPU-only counterpart. More importantly, the results show that the hybrid algorithm is less robust than the multi-GPU algorithm. For example, performing 40 resolutions of 17-Queens using the 4-GPU algorithm an average execution time of 4.55 seconds $\pm$2.87% (relative standard deviation) is obtained. For 40 resolutions of the same instance with the hybrid algorithm (4 GPU + 16 CPU) the obtained average execution time is 4.67 seconds $\pm$6.10%, meaning that the hybrid algorithm is less robust than the B&B using multi-GPU only. This robustness problem is amplified when only 1 GPU is used in combination with 16 CPU-explorers. In Table 3.10 the average elapsed time for 100 resolutions of FSP instance *Ta028* is shown for the different combinations of 1 / 4 GPUs and 0 / 16 CPUs. The obtained average is shown with the relative standard deviation (100%× standard-deviation/mean) and the observed minimum and maximum values. While the obtained minimum values are lower for the hybrid algorithm, an average slowdown and less robustness is observed. For a single GPU combined with 16 CPUs

two extreme values are observed, one execution lasting 46.0 seconds, one 27.9 seconds.

These results can be explained with the help of Figure 3.11, showing the evolution of the workload in each GPU (in terms of active IVMs) during the resolution of FSP instance *Ta022*. The left-hand side (Figure 3.11a) corresponds to a resolution without CPU-explorers and the right-hand side (Figure 3.11b) to a resolution using 16 CPU-explorers in addition to the 4 GPUs.



(a) GPU-only

(b) Concurrent GPU/CPU

Figure 3.11: Evolution of workload (#IVMs in *exploring* state) during the resolution of *Ta022*.

A comparison of both shows that, in the presence of CPU-explorers the GPUs run out of work more frequently, because they give away work units which are too large and they need to recover later. As mentioned we attempt to deal with this issue by cutting the work units into different chunk sizes. However, it is impossible to know in advance how much work each interval actually contains. These results show that the hybrid approach which consists in treating CPU and GPU workers symmetrically is questionable because it threatens the robustnessy of the algorithm. While the presented multi-GPU approach shows good results in a multi-GPU-only system, a master-worker approach could be a better approach for dealing with heterogeneous workers.

Table 3.10: Averaged execution times for FSP instance *Ta028* (100 executions) for comparing the multi-GPU only and the hybrid algorithm (using 16 CPU threads)

	GPUx1			GPUx4		
	t_{avg}	t_{min}	t_{max}	t_{avg}	t_{min}	t_{max}
no CPU (+0 CPU)	14.7±0.7%	14.6	14.9	3.9±0.6%	3.87	3.96
hybrid (+16 CPU)	16.1±22.1%	14.4	46.2	4.4±2.8%	3.84	5.09

3.6 Conclusions

In this chapter we presented the design and implementation of GPU-B&B, a GPU-centric algorithm that implements all four B&B operators on the device. To the best of our knowledge it is the first B&B algorithm to run entirely on the GPU. This eliminates data transfers and reduces the portion of sequential or weakly parallel host code. Two variants of the algorithm are proposed, both based on the IVM data structure. Both alternate exploration and work stealing phases, implemented as GPU kernels, until the interval $[0, n![$ is completely explored.

- **GPU-B&B** has two levels of parallelism, nesting the parallel evaluation of bound inside the parallel tree exploration model. On the upper level different parts of the B&B tree are explored simultaneously, performing the IVM-based branching, selection and pruning operators in parallel. For each IVM the bounding operator is in turn parallelized, refining the granularity in the performance-critical bounding phase. At the junction of the two levels a remapping phase is introduced in order to adapt the configuration and the mapping of the bounding kernel to the variable workload.
- **GPU-BT** (GPU-backtracking) performs parallel tree exploration without parallelizing the evaluation of nodes. It is designed for permutation-problems with computationally inexpensive node evaluation functions like, for instance, checking the feasibility of a subproblem with respect to constraints. In order to avoid overhead from kernel launches, GPU threads in GPU-BT are semi-persistent, in the sense that each thread performs several B&B iterations within the same kernel, whose termination is triggered when a threshold of idle threads is attained.

The implementation of irregular tree search algorithms like B&B on GPUs is challenging because the irregular nature of the algorithm is at odds with the SIMD execution model and other architectural constraints. In particular, we identified two challenges raised when revisiting the design and implementation of parallel B&B on GPUs. The first one consists in finding suitable mappings of threads onto the data, i.e. IVM structures and subproblems, in order to reduce thread divergence and detrimental effects of irregular memory access patterns. We propose alternative mapping schemes for the selection, branching and pruning operators on the one hand, and for the bounding operator on the other. The second addressed issue is the implementation of an efficient load balancing mechanism inside the GPU. As the entire exploration process is implemented on the GPU, load balancing must be performed inside the device. A GPU-based work

stealing mechanism, rarely addressed in the literature, which is used by both variants of the GPU-B&B algorithm, is proposed. It consists of a victim selection phase in which IVMs with empty intervals are mapped in parallel onto IVMs with non-empty intervals and a parallel work transfer phase. Different strategies for data-parallel work stealing are presented, based on different topologies, victim selection criteria and granularity policies. Furthermore, we proposed a hierarchical work stealing approach that performs load balancing on the inter-GPU level, enabling GPU-B&B to be executed on multi-GPU systems.

An experimental study using three different permutation problems (FSP, QAP and n-Queens) as test-cases revealed strengths and weaknesses of the GPU-B&B centric approach. Some of the main experimental results are summarized in the following.

- Using 4 Maxwell GPUs the proposed GPU-B&B algorithm solves FSP instances which require 15 minutes of sequential processing in less than 1 second and larger problems that require 21.5 hours of sequential computation in 1 minute. While for the FSP speedups of 60× and more are observed, compared to 32-threaded MC-B&B, acceleration factors for the QAP are about 10 times lower. This can be explained by the irregularity and higher memory requirements of the lower bound used for QAP.

- For the fine-grained n-Queens problem up to 10^9 nodes per second are decomposed by the multi-GPU-BT algorithm. This is about than 10× more than the fastest CPU algorithms exploiting bit-level parallelism and about 30× more than our custom MC-B&B n-Queens implementation. The proposed trigger mechanism substantially increases the performance of GPU-BT, but its tuning is challenging. Adjusting the trigger threshold dynamically would be a useful improvement of the approach. Also, the fact that GPUs cannot answer work stealing requests during the prolonged exploration phase raises stability issues.

- The proposed hypercube-based load balancing mechanism allows to handle over 100 000 IVMs on a single device efficiently. A hierarchical work stealing approach efficiently extends GPU-B&B and GPU-BT to multi-GPU systems, allowing linear scalability with the number of GPUs (up to 4) for all but the smallest instances of the three test-problems.

- Compared to the GMC-B&B offloading approach, the GPU-centric approach allows an acceleration of 1.5×-1.8× for the FSP. For the QAP the benefit of implementing the entire algorithm on the GPU is less clear, as only 10% performance improvement over GMC-B&B are observed. For fine-grained problems like n-Queens the

massive parallelization of the search itself seems to be the only viable GPU-based approach. To further summarize these results, the benefits that can be taken from implementing the entire B&B algorithm depend to a large extend on the irregularity and the memory requirements of the node evaluation function.

Chapter 4

Branch-and-Bound for hybrid HPC clusters

Contents

4.1 Introduction . . . 118

4.2 B&B for hybrid clusters . . . 118

4.2.1 B&B@Grid . . . 118

4.2.2 Design of hybrid distributed B&B . . . 120

4.2.3 Redundant exploration . . . 122

4.2.4 Implementation of worker process . . . 124

4.3 Experiments . . . 127

4.3.1 Experimental protocol . . . 127

4.3.2 Resolution of very large problem instances . . . 127

4.3.3 Scalability: Ouessant . . . 132

4.3.4 Hybrid CPU/GPU scalability . . . 135

4.3.5 Solving other 50×20 FSP instances . . . 137

4.4 Conclusion . . . 138

4.1 Introduction

This chapter presents a hybrid distributed version of the B&B algorithm (HD-B&B) for large-scale heterogeneous clusters or grids, integrating multi-core, many-core and graphics processing units. HD-B&B is based on the B&B@Grid platform [MMT07a]. B&B@Grid allows to efficiently partition the B&B tree search among distant computing nodes, which host one or several workers. Each worker explores a portion of the search space (an interval) using a sequential B&B. Thus, while compute nodes in B&B@Grid may be composed of multi-core processors, the latter are seen as a collection of mono-core processors. Therefore, B&B@Grid is revisited with the goal of adapting it to heterogeneous computing platforms.

4.2 B&B for hybrid clusters

4.2.1 B&B@Grid

The B&B@Grid platform is designed for solving large scale COPs on computational grids. B&B@Grid is based on the farmer-worker approach and uses an interval-based encoding of work units which efficiently reduces communication costs. The approach also includes efficient load balancing, fault tolerance and termination detection mechanisms. While exchanged work units are intervals, like in the IVM-based approaches, the workers execute conventional LL-based B&B processes. The conversion between lists of nodes and intervals is performed using two operations: *fold* and *unfold*. The fold operator deduces an interval from a list of pending nodes, and the unfold operator deduces an unique and minimal list of pending nodes from an interval.

Each B&B process explores an interval of node numbers, and manages the local best solution found so far. The coordinator keeps a copy of all not yet explored intervals, and manages the global best solution found so far. Figure 4.1 gives an example with three B&B processes and a coordinator. In this example, three intervals are being explored, and the fourth one is currently not explored by a B&B process.

When a worker joins the computation or a B&B process has no more local work left, it contacts the coordinator. The farmer answers the request by sending either a non-assigned interval, or the right half of the largest currently explored interval to the worker. Even when local work is still available, workers periodically inform the coordinator about their work progress (according to a fixed, user-defined period). The worker folds its current list of pending nodes and sends the corresponding interval to the coordinator.

Upon reception of an interval $[A, B[$ and its unique identifier, the coordinator updates

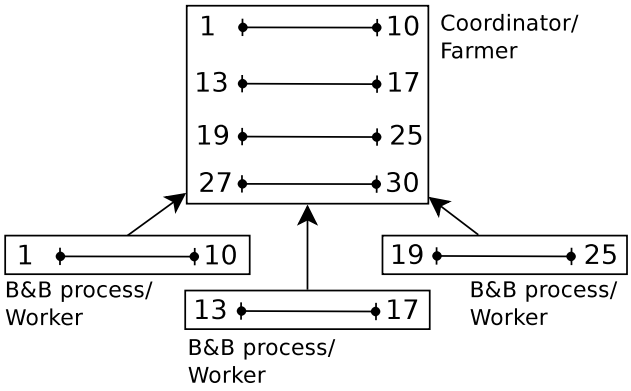


Figure 4.1: Illustration of B&B@Grid with three B&B processes and coordinator (from [MMT07a]).

the copy $[A', B'[$ of the interval. This update consists in an intersection of both intervals. Indeed, since the last synchronization both intervals $[A, B[$ and $[A', B'[$ may have evolved. A B&B process, working on $[A, B[$ increments the value of A and leaves B unchanged. The coordinator, responsible for load balancing, decrements the value of B' and leaves A' unchanged.

The importance of these periodic checkpointing operations is twofold. On the one hand, in the case of node failure the last copy of the interval becomes available for other workers. Also, the checkpointing mechanism periodically saves the set of unexplored intervals to a file, allowing to restart in case of coordinator failure. On the other hand, a worker must be informed if the right half of its interval is assigned to another B&B process. As B&B@Grid uses a pull-approach, where all communications are worker-initiated, periodic polling is the only way to achieve this. The longer a worker proceeds with the exploration without contacting the coordinator, the likelier it becomes that some parts of the search space are explored redundantly. Indeed, in the worst case a worker explores its entire interval even though parts of it have also been assigned to other workers.

B&B@Grid has been successfully used to find and prove the exact solution of an unsolved 50 jobs/20 machines FSP instance (*Ta056*). The initial upper bound was set to 3 680, which was proven to be equal to the optimal cost (3 679) plus one [MMT07a]. The exploration process lasted 25 days, using up to 1 900 processors, belonging to 9 distinct clusters of the french experimental Grid'5000[1] platform. On average 328 processors are exploited with an average usage rate of 97%, while the farmer processor is exploited only 1.7% of the time. Despite setting the worker's checkpointing period at 3 minutes, the rate of redundant exploration is less than 0.4%.

1. https://www.grid5000.fr

4.2.2 Design of hybrid distributed B&B

The experimental results for the hybrid shared-memory B&B reported in Subsection 3.5.5 indicate that load balancing among workers with very disparate computing power is challenging. In particular, the work-first principle should guide the design of load balancing mechanisms. Using a farmer-worker model the burden of servicing work requests is taken off the workers.

As illustrated in Figure 4.2, in HD-B&B workers execute different variants of the B&B algorithm, depending on the underlying hardware. For instance, a GPU can host one or more GPU-B&B processes, a vectorized version of MC-B&B can run on a Xeon Phi processor, and so on. B&B processes use a variable number of IVM-structures ("threads"). However, heterogeneous worker processes use the same communication scheme to exchange work units with the coordinator. The coordinator is unaware of the worker's type.

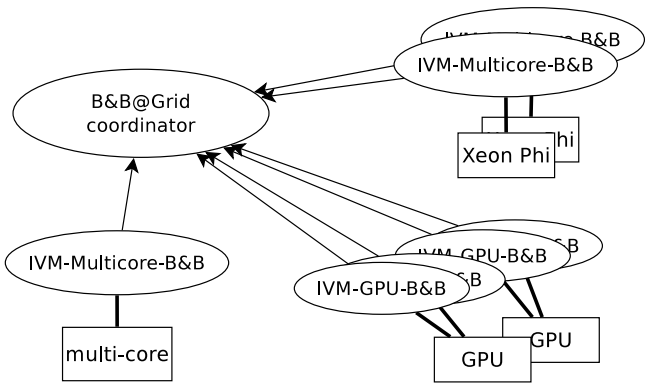


Figure 4.2: Illustration of HD-B&B, integrating IVM-based GPU-B&B and MC-B&B introduced in Chapters 2 and 3.

Locally, IVMs perform a synchronous or asynchronous parallel exploration of the assigned work unit. Using multi-threaded workers and local work stealing, an interval may be locally split into several intervals. The following example illustrates this situation. When the interval $[0, 20[$ is assigned to a worker using 4 threads, exploration and load balancing may reduce this interval to $[3, 5[\cup [6, 10[\cup [13, 15[$ – or to any other finite disjoint union of up to 4 integer intervals included in $[0, 20[$.

With respect to B&B@Grid, work units are redefined at the coordinator-level. In HD-B&B a work unit is a collection of integer intervals contained in $[0, n![$, where n is the

problem size. This collection of intervals is interpreted as a finite union of N intervals:

$$\text{work unit } W = \bigcup_{i=0}^{N} [A_i, B_i[\qquad \text{, where } \forall i : [A_i, B_i[\subset [0, n![\tag{4.1}$$

Basically any parallel B&B algorithm that has a procedure for initializing the algorithm at any work unit can be used as a worker in HD-B&B. However, there are a few assumptions to be made about workers:

- Workers explore sets of intervals. For each worker w there is a number $N_{w,max}$ that indicates the maximum number of intervals the worker can handle.
- All workers explore intervals in a consistent way. By that, we mean each interval corresponds to an unambiguous portion of the B&B tree. This requirement can easily be missed. For example, when a worker sorts nodes at the same depth by the corresponding lower bound values, there must be a consistent handling of equal values. In other words, the sorting algorithm must be exactly the same. In the case workers use different bounding operators, lexicographic DFS should be used.

In order to perform checkpointing and load balancing the coordinator uses two operations: intersection and division (in the sense of splitting) of work units. These operations are adapted to work units as defined in Equation 4.1.

Work unit intersection. The intersection of two intervals is done by considering the maximum between both start points and the minimum between both end points, as shown in Equation 4.2.

$$[A, B[\cap [A', B'[= [max(A, A'), min(B, B')[\tag{4.2}$$

The intersection of two arbitrary unions of intervals requires pairwise intersection of the intervals contained in both unions, as shown in Equation 4.3.

$$\begin{aligned} \left(\bigcup_{i=0}^{N} [A_i, B_i[\right) \cap \left(\bigcup_{j=0}^{M} [A'_j, B'_j[\right) = & \bigcup_{i=0}^{N} \bigcup_{j=0}^{M} \left([A_i, B_i[\cap [A'_j, B'_j[\right) = \\ & \bigcup_{i=0}^{N} \bigcup_{j=0}^{M} [max(A_i, A'_j), min(B_i, B'_j)[\end{aligned} \tag{4.3}$$

From a programming point of view Equation 4.3 is a permutable double for-loop over both sets of intervals. Therefore the intersection of work unit is performed with quad-

ratic complexity $O(MN)$ instead of $O(1)$ (where M and N are the number of intervals contained in both operands).

Only few assumptions can be made on the nature of the intersected work units, but they can help reduce the number of interval-intersections performed. An interval in the currently processed work unit has a non-empty intersection with *at most one* interval in the copy of this work unit. A `break`-statement can therefore be used to stop the execution of the inner `for`-loop once this interval is found. Moreover, if both sets are sorted with appropriate sorting algorithm, the intersecting interval can be found by dichotomic search in log N steps. The complexity of the intersection operator can therefore be reduced to $O(N \log N)$

Work unit splitting. The splitting of multi-interval work units is defined almost as in the GPU-accelerated MC-B&B algorithm (Chapter 2) and multi-GPU-B&B (Chapter 3). In addition, one must take into account that each worker has a fixed maximum capacity $N_{w,max}$ of intervals it can handle. This means that a worker can be assigned less, but not more than $N_{w,max}$ intervals. Worker attach this number to every message transmitted to the coordinator.

Let $W = \bigcup_{i}^{N} [A_i, B_i[$ be the work unit selected for splitting by the coordinator. If the number of intervals contained in this work unit (N) is larger than or equal to the requested number $N_{w,max}$, then the right half of the first $N_{w,max}$ intervals in W forms the new work unit. In the contrary case ($N < N_{w,max}$) the right half of the first N intervals is the new work unit.

To sum up, W is split in two parts W' and W'' as shown in Equation 4.4. In the load balancing process the coordinator replaces W by W' and attributes W'' to the requesting worker.

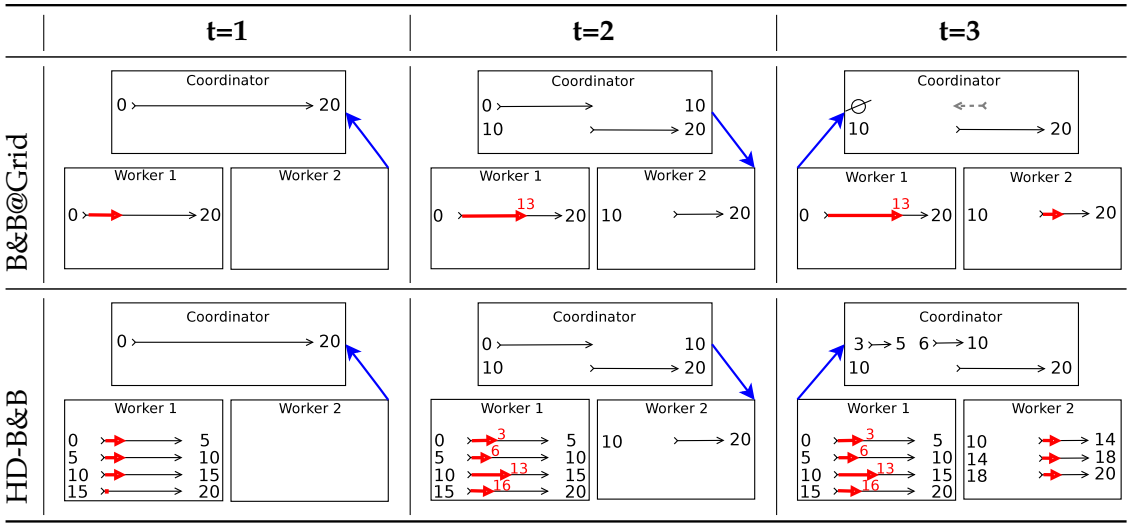
$$\begin{aligned} W &= \bigcup_{i}^{min(N,N_{w,max})} [A_i, C_i[\\ W' &= \bigcup_{i}^{min(N,N_{w,max})} [C_i, B_i[, \text{ where } C_i = \frac{A_i + B_i}{2} \end{aligned} \tag{4.4}$$

4.2.3 Redundant exploration

As mentioned earlier, in the load balancing scheme used by B&B@Grid, redundant exploration of intervals may occur. The original B&B@Grid algorithm is based on the assumption that the amount of redundant work is small because workers only increment

the begin of their interval, while work is only stolen from the end. This assumption does not hold in HD-B&B when workers are allowed to locally divide their assigned work units. In fact, with single-interval work units redundant exploration occurs only occasionally and the amount becomes negligible for large instances. When the same load balancing scheme is used in HD-B&B redundant exploration occurs systematically. Table 4.1 illustrates how redundant exploration for both types of work units occurs.

Table 4.1: Illustration of redundant tree exploration in B&B@Grid and HD-B&B.



The upper row of Table 4.1 illustrates the case of single-interval work units like in B&B@Grid. In the example, worker 1 explores the interval $[0, 20[$ while worker 2 is idle. Worker 2 contacts the coordinator and is assigned the right half $[10, 20[$ of the work unit $[0, 20[$. In the meanwhile, worker 1 continues exploring its interval: in the example up to position 13, before worker 1 contacts the coordinator for checkpointing. In this case, the interval $[10, 13[$ is explored redundantly by workers 1 and 2. The lower row illustrates the case of locally splittable multi-interval work units like in HD-BB. In contrast to the previous example, worker 1 explores interval $[0, 20[$ in parallel by splitting it into 4 parts, for instance $[0, 5[$, $[5, 10[$, $[10, 15[$, $[15, 20[$. Therefore, the second half $[10, 20[$ is explored almost immediately after the reception of interval $[0, 20[$. In contrast to the single-interval case, one cannot expect that the ratio of performed redundant work decreases as the size of the explored tree increases. Instead, it seems reasonable to expect the amount of redundant work to increase with the number of IVMs used per worker, independently from the tree-size.

In preliminary experiments using FSP instance *Ta*022 (22.1M nodes) we observed

more than 60% of redundant exploration with 20 GPU-workers (128 IVMs per worker) and 12% with 20 CPU-workers (4 IVMs per worker). A moderate, but insufficient decrease of redundant exploration is observed when increasing the tree-size.

The proposed approach to tackle this issue is based on the following idea. The origin of redundant exploration, as illustrated in the second row of Table 4.1, is the local load balancing mechanism which performs interval splitting without informing the coordinator. If local interval-splitting operations were performed simultaneously in the copies held by the coordinator, the load balancing mechanism would be conceptually equivalent to the single-interval B&B@Grid approach. In a distributed memory environment this is not possible. However, workers can inform the coordinator about local work stealing operations directly after performing them. The implementation of a worker process, more precisely, the interface used for communication with the coordinator, is described in the following subsection.

4.2.4 Implementation of worker process

As discussed in the previous subsection, redundant search space exploration can be avoided by updating the coordinator's copy of a work unit as soon as this work unit is locally partitioned. Therefore, in HD-B&B, a local load balancing operation triggers a communication with the coordinator. This considerably increases the communication overhead. For example, in the case of a GPU-worker, a communication operation includes the following steps: (1) copy factoradic intervals from device to host, (2) convert factoradic to decimal intervals, (3) send work unit to coordinator, (4) wait for answer, (5) check if local work needs to be updated and (6) perform update if necessary. Only steps (1) and (6) require synchronization with the currently explored work unit (in the factoradic form, used by the worker) and steps (2)–(5) can be performed independently from the exploration process.

In order to decrease communication overhead for the worker, HD-B&B uses asynchronous communications. Each worker is composed of one *communicator thread* and one *exploration thread*. Figure 4.3 shows a flowchart illustrating a B&B process using a communication thread for asynchronous communications with the coordinator. Solid black lines indicate the control flow of both threads and dashed lines show some of the more important data dependencies. In particular it indicates the flow of work units from the B&B exploration thread to the coordinator and back.

If some conditions (named CONTACT in Figure 4.3) are met and if the communicator thread is in the READY-state, the worker thread writes its current intervals to a buffer.

Also written to the communication buffer are: the best solution found so far, exploration statistics, the work unit's identifier (ID) and the maximal number of intervals ($N_{w,max}$) that can be handled by the worker. After writing to the buffer, the exploration thread unlocks the waiting communication thread by incrementing a binary semaphore, and resumes exploring its work unit.

Even if the CONTACT conditions are met again, the exploration thread does not re-fill the buffer until the communication thread finishes the current task and raises its READY-flag. There are two possible outcomes of the communication operation: either the worker thread needs to update its local work unit or not. If an update is available, the communication thread waits until the exploration thread has applied this update. Otherwise, the communicator thread does not interrupt the worker thread and sets its READY flag to true.

The parallel B&B exploration performed by the worker thread can be any of the parallel B&B algorithms described in this thesis. The subroutine called "parallel B&B exploration" in Figure 4.3 incorporates local load balancing and returns whenever the conditions for contacting the server are met or an update is available. The CONTACT conditions determine how frequently a worker checkpoints. Assuming that the communication thread is waiting, a checkpoint operation is triggered if one of the following is true.

1. No more work is locally available. This is the case when all intervals of the work unit are empty. The worker needs to contact the server to request new work.

2. The server has not been contacted for a time longer than a user-defined fixed time period. The coordinator should be regularly informed about the progress of the exploration. By default, this parameter is set to 30 seconds.

3. The best found solution has been improved. It is important to communicate this information as soon as possible to all participating processes because it makes the pruning mechanism of all explorers more efficient.

4. A local load balancing operation was performed. Especially when the worker uses a large number of IVMs this condition may trigger communications very frequently. In order to limit the amount of communications, workers perform local load balancing phases only if more than 20% of IVMs are empty.

These conditions are checked at each iteration of the local exploration process. In addition, the local B&B process checks at each iteration whether an update is available.

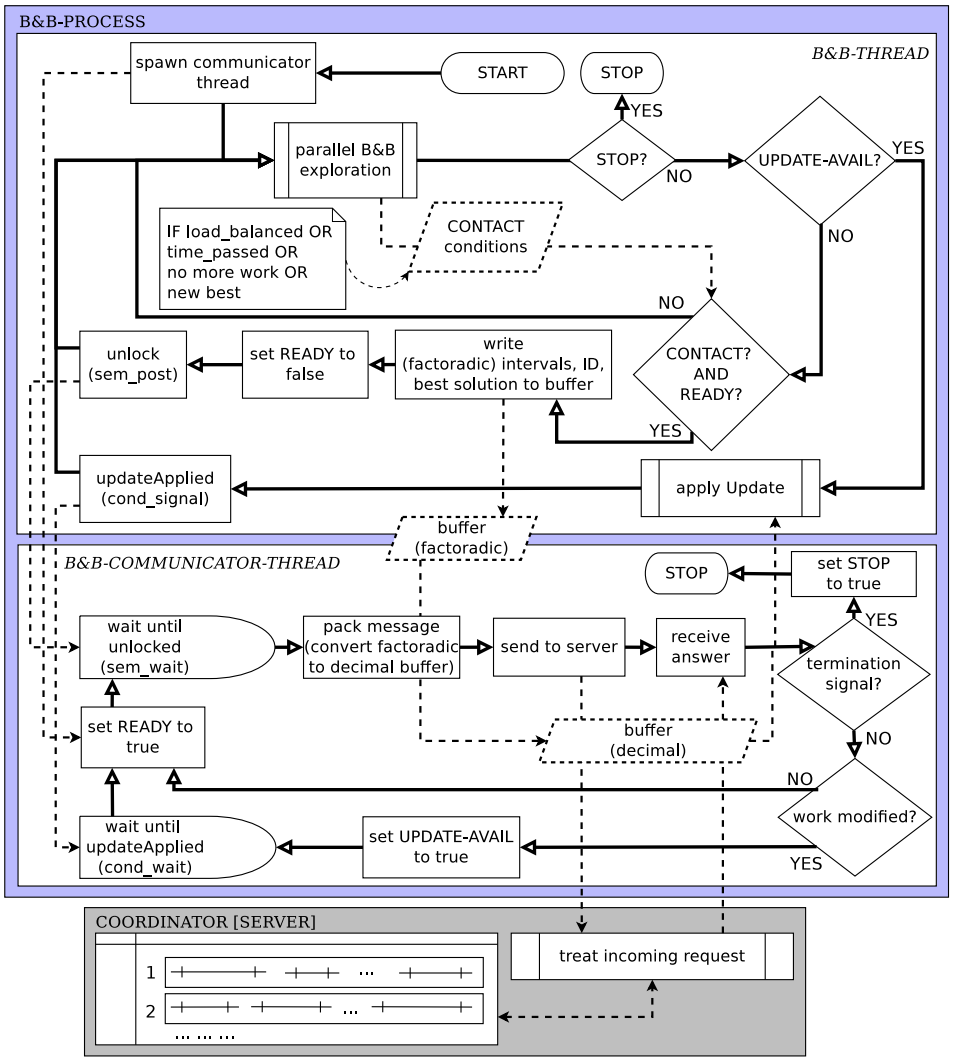


Figure 4.3: Flowchart illustrating a HD-B&B worker process composed of a worker thread and a communicator thread. Solid lines represent control flow and dashed lines show data dependencies. For the sake of clarity some implementation details have been spared out.

4.3 Experiments

4.3.1 Experimental protocol

In previous experiments we mainly used the 20x20 Taillard instances (*Ta021-Ta030*) because they are large enough to require parallel computing, but small enough to be solved within a couple of minutes by a GPU accelerated algorithm. The largest of these instances, *Ta023*, requires 21.5 hours of sequential processing using a single CPU core and is solved within 1 minute on a 4-GPU shared memory system. For experimentations on larger, distributed GPU-equipped systems this instance is too small. Most of the Taillard instances defined by 50 jobs, 20 machines (*Ta051-Ta060*) seem too large to be solved within a reasonable amount of time on the available machines. We have used these 50 jobs-on-20 machines instances to generate smaller instances using the following procedure.

1. Initialize HD-B&B for a resolution of instance $Ta0X$ (with $X = 51, \ldots, 60$ and best known upper bound). Then, run HD-B&B using a single GPU-worker (with 4 096 IVMs) and save remaining intervals to a file after 1, 2, ..., 5 hours (coordinator checkpointing).

2. Read F_h^X, the set of intervals remaining after the exploration of instance X after h hours. Perform the operation $E_h^X = [0, N![/F_h^X$.

3. Instance $Ta0X$-h consists in exploring the search space E_h^X with the processing time matrix defined by base instance $Ta0X$.

As no improvement of the initial upper bound occurs in step 1, different explorations of the search subspace defined by E_h^X visit the same number of nodes.

For the implementation of arbitrary precision integers the GNU Multiple Precision Arithmetic Library (GMP[2]) is used. For inter-node communication TCP/sockets are used.

4.3.2 Resolution of very large problem instances

To the best of our knowledge, of the 10 Taillard instances defined by 50 jobs and 20 machines (*Ta051-Ta060*), *Ta056* is currently the only one for which the best known solution is proven to be optimal. As mentioned, the optimal solution of *Ta056* was found and proven in 2006 using B&B@Grid [MMT07a]. The resolution required 25 days of processing,

2. https://gmplib.org/

exploiting on average 328 processors distributed on 9 clusters of the French experimental testbed Grid'5000. This result is used as a reference for the three resolutions of *Ta056* which are performed under identical initial conditions. As in the B&B@Grid experiment reported in [MMT07a] the initial upper bound is set to the optimal cost plus one unit, i.e. 3 680, which allows one to verify the correctness of the algorithm.

The following three resolutions of *Ta056* found the same optimal cost (3 679), and the same optimal solution.

1. Using the GPU-B&B algorithm on a shared memory 4-GPU system (**gpu-8k**) at the Mathematics and Operations Research Department (UMONS). The system is composed of a dual-socket 8-core Haswell (E5-2630v3) CPU and 4 Maxwell (GTX 980) GPUs. Thus, *gpu-8k* has 16 CPU cores and 8 192 CUDA cores in total.

2. Using HD-B&B on the **chifflet** cluster of the Grid'5000 site in Lille[3]. The cluster is composed of 8 nodes, with 10-Gigabit/s SFP+ interfaces and 768 GB memory per node. Each node is composed of two 14-core Broadwell (E5-2680v4) CPUs and two Pascal (GTX 1080Ti) GPUs. In total, *chifflet* has 224 CPU cores and 57 344 CUDA cores.

3. Using HD-B&B on the prototype **ouessant** cluster located at the *Institut du développement et des ressources en informatique scientifique* (IDRIS[4]). The *ouessant* cluster is composed of 12 OpenPower "Minsky" nodes (S822LC). Each of the 12 "Minsky" nodes is composed of two POWER8+ 10-core CPUs and 4 Pascal (P100) GPUs, fully connected via NVLink. Nodes are interconnected by 100-Gigabit/s Mellanox EDR IB Coherent Accelerator Processor Interface (CAPI). In total, *Ouessant* has about 170 000 CUDA cores and 240 CPU cores. For technical reasons, in our experiments only 9 of the 12 nodes were available.

The measured resolution time for instance *Ta056* is illustrated in Figure 4.4.

Execution time A first GPU-accelerated resolution of *Ta056* was performed on *gpu-8k*. The algorithm finds and proves the optimality of the cost 3 679 in approximately 228 hours (9.5 days). This is about 2.6× faster than the 2006 resolution using B&B@Grid. Based on this result and hardware specifications one can try to predict the execution time on *chifflet* and *ouessant*. *Chifflet* has 7.0× as many CUDA cores as *gpu-8k*, which run at 1.3×

3. Some experiments presented in this chapter were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

4. http://www.idris.fr/

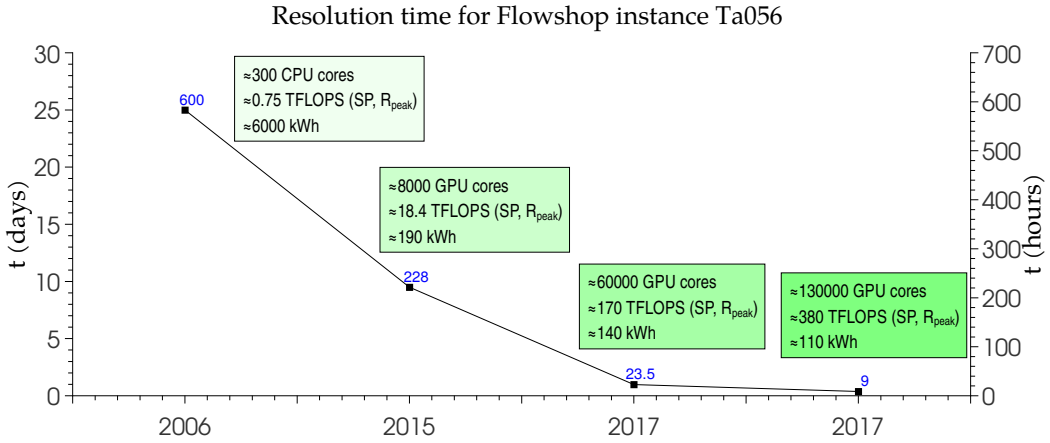


Figure 4.4: Resolutions of FSP instance *Ta056*, initialized at "optimal cost plus one". **2006**: B&B@Grid ([MMT07a]), **2015**: multi-GPU-B&B@*gpu-8k* (Chapter 3), **06/2017**: HD-B&B@*chifflet*, **08/2017** HD-B&B@*ouessant*.

higher base clock frequencies. Assuming good scalability of HD-B&B and neglecting higher memory bandwidth and clock speeds, one can expect to reduce the execution time on *chifflet* by a factor $7.0 \times 1.3 = 9.1$, i. e. to 25 hours. Using 9 nodes of *ouessant*, about 130 000 CUDA cores with a base clock frequency of 1 328 MHz are available. Following the same reasoning, one can expect to solve instance *Ta056* in about 12 hours, i.e. 18.6 times faster than on *gpu-8k*.

Of course, these estimations cannot be completely accurate as they neglect communication costs and architectural differences like HDM2 stacked memory of the P100 GPUs. The actual elapsed wall time measured for both resolutions is smaller than these predictions, but approximately in accordance with them. On *chifflet*, *Ta056* is solved in less than a day (23.5 hours), on *ouessant* in 9 hours. Compared with the B&B@Grid resolution, the execution time is reduced by a factor of about 65×. Compared to the estimated sequential execution time of 22 years, the execution time is reduced by at least four orders of magnitude.

Energy Figure 4.4 also indicates an approximate value for the energy consumption of each resolution. These values are based on the Thermal Design Power (TDP) of CPUs and GPUs, as listed by the respective vendors. For example, the GTX 980 GPU is listed with a TDP of 165 W and the host Xeon CPUs with 85 W, so an indicative value for the energy consumption is given by $(4 \times 165 + 2 \times 85)\text{W} \times 228h \approx 190\,\text{kWh}$.

For the 2006 resolution using B&B@Grid the energy consumption can only be roughly estimated. About $^2/_3$ of processors in the computational pool exploited by B&B@Grid in

2006 are AMD Opteron dual-core CPUs, 90 nm feature size, with clock rates between 2.0 and 2.2 GHz. The remaining 1/3 are Intel Pentium 4 and Celeron mono-core processors with similar clock rates. The most energy-efficient models of this type of CPUs are listed with TDP values below 60 W, but most have TDPs of about 80 W. Taking into account that most CPUs are dual-core, an optimistic estimation for the energy consumption is 328 processors × 30W × 25d × 24h/d ≈ 6000 kWh.

For *chifflet* a measured value is available for comparison, as power monitoring is provided by the Ganglia tool. As shown in Figure 4.5, the Ganglia power monitoring indicates that power remained approximately constant at 5.4 kW during the entire exploration process. For 23.5 hours of processing this amounts to an energy consumption of 130 kWh, which is close to the TDP-based estimation of 140 W shown in Figure 4.4.

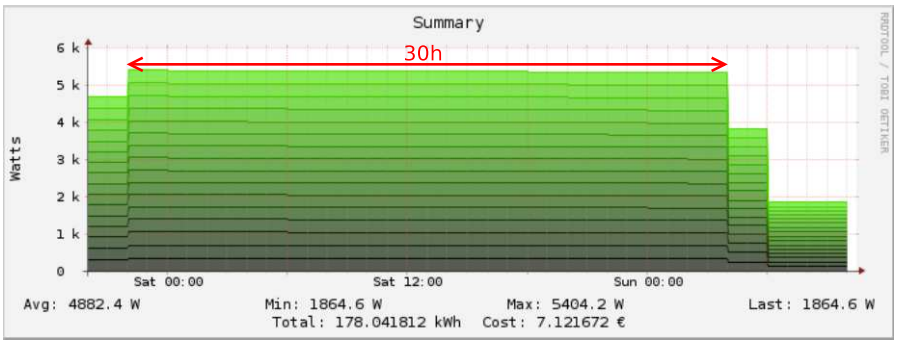


Figure 4.5: Power monitoring on *Chifflet* during the resolution of *Ta056*. NB: in this run HD-B&B is initialized with "optimum plus 2" as the initial upper bound, which explains the longer execution time, compared to Figure 4.4.

Load balance Figure 4.6 illustrates the workload repartition among GPU- and CPU-based workers, in terms of decomposed nodes. In addition to four GPU-workers, on each *ouessant* node one multi-threaded CPU-based worker with 160 IVM is used (2 × 10 cores × 8 threads).

In this configuration, each multi-core B&B process decomposes on average about $\frac{1}{10}$ the amount of nodes decomposed by an average GPU-B&B. Using 4 times as many GPU-based workers as multi-core workers, in total less than 3% of node decompositions are performed on a CPU. One can see in Figure 4.6 that workers of the same type perform a roughly equal amount of work. However, a node decomposition represents a variable amount of work. Therefore, the number of decomposed nodes is only an approximative indicator for load balance.

Another indicator for the load imbalance throughout the execution is the number

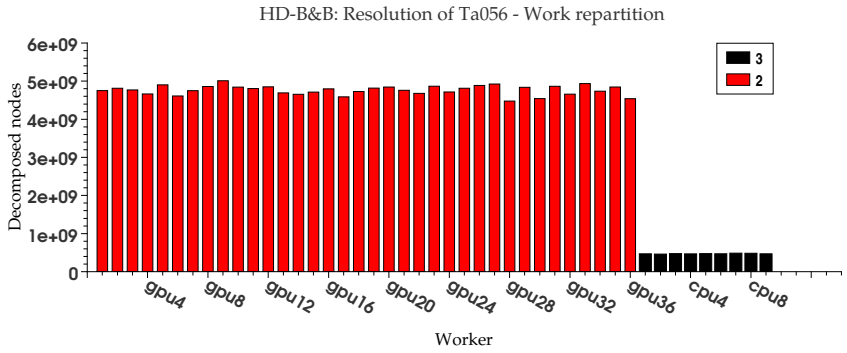


Figure 4.6: Number of nodes decomposed per worker. Resolution of *Ta056* on 9 "Minsky" nodes (*ouessant*), using 9×(4 GPU-based workers with 16 384 IVM + 1 CPU-based worker with 160 IVM).

of work allocations (work stealing operations). A worker only requests new work from the coordinator when no more work is locally available. This indicator does not take into account local load distribution. The results indicate that workers run out of work very rarely. During the resolution on *gpu-8k* only 80 device-to-device work stealing operations were performed during 9.5 days of execution time, i. e. on average a work stealing operation occurs only every three hours, knowing that most operations occur in the beginning and the end of the exploration process.

On *chifflet*, using 16 GPUs and no CPU-workers, the coordinator performs 519 work unit allocations and about 60 000 checkpointing operations (work unit intersections). During the 23.5 hours (1 400 minutes) of execution, the coordinator spends approximately 5 minutes processing worker requests. The remaining 1 395 minutes the farmer processor is idle, as the coordinator waits for incoming messages. In other words, the farmer processor is exploited only 0.4% of the time, which is lower than for B&B@Grid, where the farmer is exploited 1.7% of the time.

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

Tree size Unfortunately, for the *gpu-8k* resolution, the number of decomposed nodes is not available because of a technical problem (counter overflow). The tree exploration on *chifflet* required 174.3×10^9 node decompositions, on *ouessant* 175.8×10^9. In [MMT07a] the reported number of "explored nodes" is 6.508×10^{12}. We believe that this number refers to the number of computed lower bounds. This explanation seems plausible

because of the following. Based on these numbers, one can deduce the average number of bounds computed per node decomposition ($\frac{6.5\times10^{12}}{175\times10^{9}} = 37.14$). Further taking into account that solutions are build from both ends, the average depth of a decomposed node is $\frac{37.14}{2} = 18.57$. Extrapolating from typical tree-shapes of 20-job instances 18.6 is a plausible average node-depth for 50-job instance *Ta056*.

Table 4.2: Exploration statistics for resolution of FSP instance *Ta056*

	gpu-8k	*chifflet*	*ouessant*
GPUs	4×GTX 980	16×GTX 1080 Ti	36×P100
CPUs	0	0	18×Power8+
decomposed nodes	n/a	174.3×10^9	175.8×10^9
elapsed time	229.0 h	23.5 h	9.0 h
$T_{coordinator}$	n/a	5.1 min	38.2 min
Coordinator exploitation (%)	n/a	0.36%	7.1%
#checkpoints	n/a	61 100	3 568 368
#work allocations	n/a	519	33 387

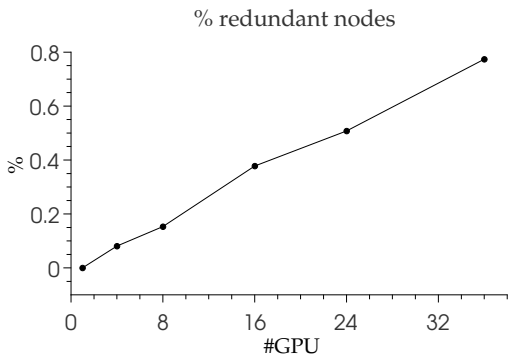
4.3.3 Scalability: Ouessant

In this subsection HD-B&B is experimented with 6 of the "5-hour" instances, *Ta053-5*, *Ta054-5*, *Ta055-5*, *Ta057-5*, *Ta058-5* and *Ta059-5*, generated as described in Subsection 4.3.1. All experiments are performed on the *ouessant* cluster, using up to 36 Pascal P100 GPUs, distributed on 9 nodes.

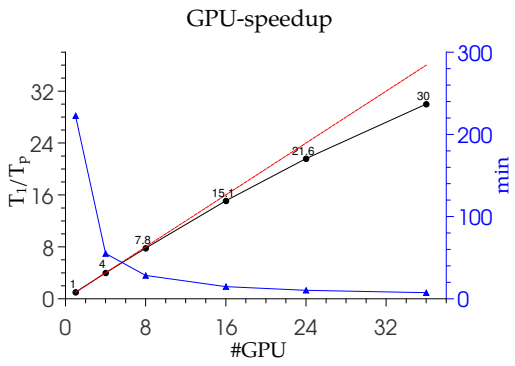
Figure 4.7 shows the experimental results for 1, 4, 8, 16, 24 and 36 GPUs as averages over the six instances. The four Subfigures 4.7a, 4.7b, 4.7c, 4.7d respectively show the rate of redundant node decompositions, the speedup achieved with p GPUs compared to a single GPU, the number of checkpoint operations and a breakdown of the coordinator activity.

One can notice in Figure 4.7a that the rate of redundant node decompositions increases according to the number of GPUs. On a single GPU no redundant node exploration occurs and the number of explored nodes, identical from one execution to another, is used as the reference value. Using 36 GPUs, about 0.8% of node decompositions are redundant.

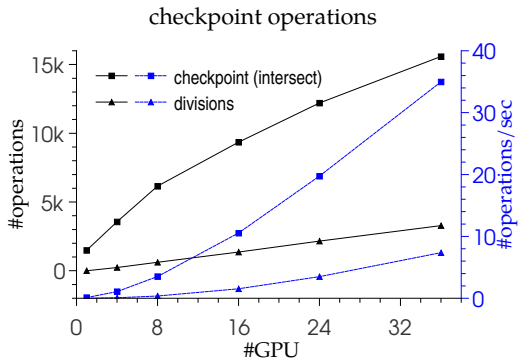
Figure 4.7b shows the elapsed time (in minutes) in blue on the right y-Axis and the speedup with respect to a single GPU in black on the left y-Axis. The red dashed line corresponds to linear speedup with the number of GPUs. The average execution time on a single P100 GPU is about 220 minutes. Using 36 GPUs the average execution time decreases to approximately 7.5 minutes, which corresponds to a relative speedup of 30× over a single GPU, for 83% efficiency. The efficiency rate depends on the size of the



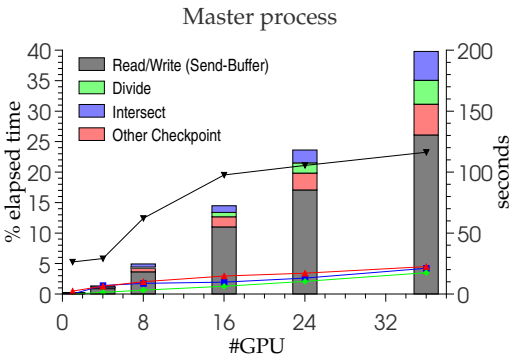
(a) Rate of redundant decomposed nodes.



(b) Elapsed time and speedup over 1 GPU.



(c) Number of checkpointing operations.



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

Figure 4.7: Evaluation of HD-B&B on cluster *chifflet*. Results are shown as averages over FSP instances *Ta053-5*, *Ta054-5*, *Ta055-5*, *Ta057-5*, *Ta058-5* and *Ta059-5*. Using $T = 16\,384$ IVM per GPU and hypercube work stealing strategy for intra-GPU load balancing.

solved problem instance and is studied later on in this subsection.

In Figure 4.7c the number of checkpoint operations (work unit intersections) and work unit divisions performed by the coordinator is shown in black on the left y-Axis. The number of operations per second is shown in blue on the right y-Axis. Naturally, as the number of GPUs increases the number of work unit allocations (resulting from the division of existing work units) also increases. For the considered scale, the rate of increase appears to be constant. A checkpoint operation that results in a work unit division is counted for both metrics. Therefore the number of checkpoint operations is always greater than the number of divisions. One can see that the number of checkpoints increases faster than the number of divisions. However, using more GPUs does not necessarily increase the number of checkpoint operations. Indeed, if all workers contact the

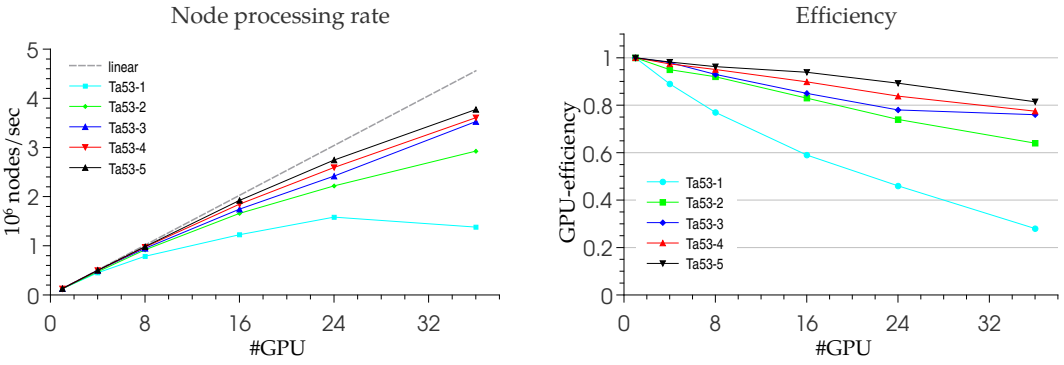
coordinator in fixed and regular intervals, and if the elapsed time decreases linearly with the number of GPUs, then the total number of checkpoint operations would remain constant. In contrast, the results show that workers contact the coordinator more frequently, which is a consequence of more frequent local work stealing operations. At near-linear acceleration factors, the rate at which the coordinator performs checkpoint operations increases quadratically. Many checkpoint operations are performed by replacing the coordinator's copy by the current work unit (if the copy wasn't modified remotely).

Figure 4.7d focuses on the activity of the coordinator, which is split into four parts measured separately. The absolute time (in seconds) spend in these parts is represented by solid lines. Stacked bars represent this time as a percentage of the total elapsed time. Messages are sent and received in the form of a sequence of characters (*stringstream*), which must be written/read to/from valid work units. This manipulation of the send-buffer consumes 65-90% of the coordinator's processing time (decreasing with the number of GPUs). This overhead is significant and should be reduced. Rewriting communication routines with MPI, instead of socket programming, in order to take advantage of optimized derived datatypes [Sun+03] may be a necessary modification of HD-B&B. As the number of GPUs increases, the coordinator spends a greater portion of time actually treating the requests. For 36 GPUs, work unit intersection, division and other checkpoint operations (e. g. periodically saving all work units to the disk) each consume 10-12% of the coordinator's time. In total, the coordinator is exploited 40% of the time when 36 GPUs are used. At this rate it is likely that incoming requests from workers are queuing up, causing the coordinator to become a bottleneck. This exploitation rate is about 100 times higher than the rate observed during the resolution of *Ta056*, which lasts approximately 80 times longer.

In order to evaluate the impact of the instance size on the efficiency of HD-B&B the scaling experiment corresponding to Figure 4.7b is repeated with smaller instances *Ta053-n*, $n = 1, 2, 3, 4, 5$. Figure 4.8 shows the obtained node processing rates (in Mn/s) on the left-hand side (Figure 4.8a) and GPU-efficiencies on the right-hand side (Figure 4.8b). GPU-efficiency is defined analogous to the conventional parallel efficiency definition, replacing processors with GPUs.

The smallest of these instances, *Ta053-1* is solved in 25 minutes on a single P100 GPU and the largest, *Ta053-5*, lasts for 220 minutes on a single device. Unsurprisingly, efficiency and node processing rates increase as the size of the explored tree increases. In order to exploit all 36 available GPUs efficiently ($> 70\%$), the FSP instance to be solved should at least require 10^9 node decompositions (*Ta053-3*, requiring 2 hours of processing on a single GPU). For smaller instances, the repartition of the search space, represented

by the interval $[0, n![$, among $36 \times 16\,384 \approx 600\,000$ IVMs incurs too much overhead. Even for instance *Ta053-3*, solved in 4.5 minutes on 36 GPUs, each IVM decomposes on average only $\sim 1\,500$ subproblems.



(a) Average node processing rate (in Mn/s).

(b) GPU-efficiency for p GPUs ($\eta = \frac{T_1}{pT_p}$).

Figure 4.8: Resolution of instances *Ta053-n* (n=1,2,3,4,5) on *ouessant*.

4.3.4 Hybrid CPU/GPU scalability

As shown in Figure 4.6, in a hybrid resolution of large FSP instances, the largest part of the total workload is processed by the Pascal GPUs, which provide much higher node processing rates than the multi-core CPUs. In that sense, and considering the FSP test-case, the *ouessant* and *chifflet* clusters are highly unbalanced systems. In this case, the exploitation of available CPU cores provides at best a marginal acceleration of the exploration process. This situation may change when a different problem is solved, or another heterogeneous platform is targeted. In order to evaluate the scalability of HD-B&B with GPUs *and* CPUs further experiments are performed on the **'kepler'** cluster located at the Mathematics and Operations Research Department at UMONS University.

- *Kepler* is a cluster of 20 low-power system-on-a-chip (SoC) devices. Each of the 20 nodes is a Nvidia Tegra K1 SoC featuring a 32-bit quad-core ARM Cortex-A15 CPU and a Kepler GK20A GPU containing 192 CUDA cores. Tegra K1 is primarily designed for graphics intensive mobile applications, like gaming, and is used in different tablet computers. Therefore, battery lifetime is a main design objective and Tegra K1 has a TDP of less than 10 W.

We evaluate the scalability of HD-B&B as follows. For a given problem instance, the node processing rate (in nodes/sec) on a single CPU (resp. GPU) is measured. Based on

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

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

.

For instance, solving *Ta*022 on a single CPU (resp. GPU) a node processing rate of $\alpha = 4.90\ kn/s$ (resp. $14.12\ kn/s$) is achieved. Using 8 GPUs and 4 GPUs, the same instance is solved with an average node processing rate of 131.6 kn/s. Supposing linear scalability with CPUs and GPUs one can expect to achieve a node processing rate of $8\times14.12+4\times4.90 = 132.6\,kn/s$. In this case, HD-B&B reaches an efficiency of $\eta_{4,8} = 99.2\%$.

Table 4.3 reports the efficiency $\eta_{m,n}$ achieved for m, n=4, 8, 12, 16, 20 solving FSP instance *Ta022*. The results shown in this table are averages over 5 independent runs and the relative standard deviation (RSD) is shown on the right-hand side. The initial runs on one CPU (resp. GPU) were also performed 5 times. The two tables on top (Tables 4.3a and 4.3b) show results using 128 IVMs per GPU, the two bottom tables (Tables (4.3c and 4.3d) show results for 1024 IVMs per GPU. Each MC-B&B process is a 4-threaded (4-IVM) exploration process. The coordinator process runs on a reserved node (without concurrent exploration processes) except for the runs where #CPUs=20 or #GPUs=20.

The node processing rates for individual workers are the following: (1) 4.90 kn/s for a MC-B&B worker, (2) 14.12 kn/s for GPU-B&B with $T = 128$ and (3) 22.10 kn/s for GPU-B&B with $T = 1024$. For reference, we recall that the sequential processing rate on a Intel E5-2630v3 CPU is about 1.93 kn/s. For $T = 128$, resp. $T = 1024$ IVMs, the maximal rate of the hybrid system is therefore 380 kn/s, resp. 540 kn/s. Using all 20 nodes, the achieved processing rates is 293 kn/s ($\eta_{20,20} = 77\%$) , resp. 373 kn/s ($\eta_{20,20} = 69\%$), meaning that *Ta022* is solved in 75, resp. 59 seconds.

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

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

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

		GPU x					
	Eff	0	4	8	12	16	20
CPU x	0		0.1	0.8	0.6	0.7	2.3
	4	0.1	3.4	0.2	2	1.8	1.2
	8	0.2	4.3	5	2.5	4.2	1.8
	12	0.4	8.9	6.9	3.4	6.9	5.2
	16	0.2	11.4	7.7	8.9	7.5	4.5
	20	0.7	6.5	11.2	3.6	9.2	5.9

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

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

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

		GPU x					
	Eff	0	4	8	12	16	20
CPU x	0		0.8	1.3	0.9	1.1	0.9
	4	0.1	2.1	2.9	2.4	2.3	1.5
	8	0.2	2.5	7.7	2	4.7	4.2
	12	0.4	1	1.5	4.5	4.6	4.7
	16	0.2	9.6	3.7	4.7	3.8	2.3
	20	0.7	13.2	11.7	3.1	8	3

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

Table 4.3: Mixed GPU-CPU efficiency and Relative Standard Deviation (RSD) for resolution of FSP instance *Ta022* (22.1×10^6 nodes) using 20 Tegra K1. The upper (resp. lower) row shows results for $T = 128$ (resp. $T = 1024$) IVMs/GPU. Average efficiency and RSD over 5 runs.

efficient handling of the shutdown phase revealed challenging.

4.3.5 Solving other 50×20 FSP instances

To the best of our knowledge, the only 50 jobs-on-20 machine FSP instances exactly solved up to day is *Ta*056. The irregularity of the B&B-tree associated with the resolution of an instance makes it extremely difficult to estimate its size and thus the computing power required. In order to get an idea of the computational effort required to solve instances from the group *Ta051-Ta060*, we performed partial explorations of these instances, defining a cutoff depth d. All nodes at depth d are treated as leaf nodes which do not improve the upper bound. Performing successive explorations with increasing cutoff depth d ($d = 1, 2, 3, \ldots, d_{max}$) allows to obtain the number of frontier nodes at each level up to d_{max}. Figure 4.9 shows the size of partial trees for instances *Ta052*, *Ta057*, *Ta055* and *Ta056*. The other instances of this group are not shown because they have significantly higher branching factors in the upper part of the tree.

Based on a comparison of partial trees up to depth 14, the smallest instance after *Ta056* is *Ta057*. The partial tree up to depth 14 developed for *Ta057* is 21 times bigger than the one of *Ta056*. For *Ta052* the partial tree of depth $d = 14$ it is 108× bigger than for *Ta056*, with a size that almost equals $1/10^{th}$ of the entire *Ta056* tree.

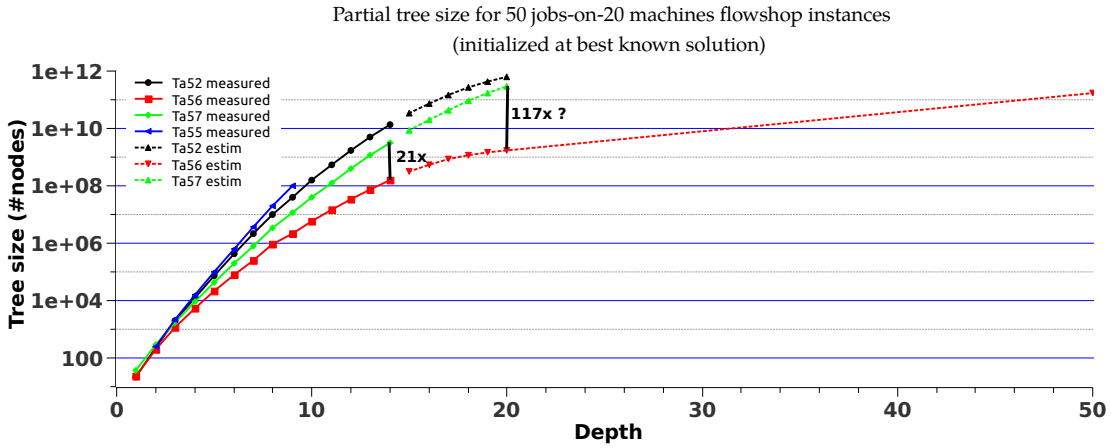


Figure 4.9: Partial exploration of 50 × 20 Taillard instances.

Based on this information we attempt to estimate the tree size five levels further, as follows. Over depths 9-14 the average rate at which the branching factor decreases is computed and applied to predict the branching factors for levels 15-20. The result is shown in Figure 4.9, leading to the estimation that "most attainable" of these unsolved instances *Ta057* is at least 100 times larger than *Ta056*. One indicator for an approximate correctness of that estimate is that the method predicts that the number of nodes per depth for *Ta056* peaks at $d = 19$ - and we know that the average depth of a node during the resolution of *Ta056* is 18.5.

Based on these estimations, exactly solving *Ta*057 on a platform equivalent to *ouessant* requires at least one full month of computation. One factor which may decrease this estimate and which is completely unpredictable is a decrease of the best known solution, as the current one may not be optimal. This might cause the algorithm to terminate earlier than expected. Indeed, for all three resolutions of *Ta056* that were performed, the optimal solution was found after more than 90% of the total execution time had elapsed.

4.4 Conclusion

In this chapter we revisited the design of B&B@Grid to enable the integration of GPU-, MIC- and multi-core-based workers. When B&B@Grid was designed most components of computational grids were mono-core and dual-core CPUs. Today, 10 years later, large-scale HPC platforms are becoming increasingly heterogeneous, integrating GPUs and CPUs with larger core-counts.

The extension of B&B@Grid to HD-B&B, the proposed B&B for hybrid distributed

HPC clusters, includes the redefinition of work units and a modification of the communication scheme to allow asynchronous checkpointing operations that overlap with worker computations.

A very large FSP instance defined by 50 jobs and 20 machines, *Ta056*, was successfully solved on GPU-equipped clusters with a total of up to 130 000 GPU cores. A first resolution of this instance was performed in 2006, using B&B@Grid to exploit on average 328 processors in a computational grid during 25 days. Using HD-B&B the resolution of *Ta056* was performed in 9 hours on a cluster composed of 36 GPUs. Low exploitation rates of the master process and experiments performed with smaller instances are indicators for the good scalability of HD-B&B. For a set of 50-job FSP instances requiring 3.7 hours of computation on a single GPU, a relative speedup of 30× is achieved on 36 GPUs, solving these instance in 7.5 minutes on average. HD-B&B was also experimented on a "mini-cluster" of 20 systems-on-a-chip designed for mobile devices. Experimental results show that the availability of efficient data types for inter-node communication is a key component for the performance of the central coordinator process.

Chapter 5

Conclusions and Perspectives

The latest Top500 ranking confirms that hybrid high performance computing technologies combining multi-core and many-core processors is the road towards exascale computing. In the near future, manufacturers will have to focus on upgrading the memory subsystem in order to reduce the cost, in terms of time and energy, of memory operations. This is likely to make the hierarchical organization of memory even more complex, requiring the awareness of programmers and efficient data structures. Also, the trend towards smaller low-power cores may well continue as the simplification of control logic bears potential energy savings and microprocessor technology is increasingly driven by the mobile market. The design and implementation of efficient algorithms for those computing environments is challenging.

In this thesis the focus is put on exact combinatorial optimization using tree search algorithms. Based on an innovative data structure dedicated to permutation problems, called Integer-Vector-Matrix (IVM), we have revisited the design and implementation of Branch-and-Bound (B&B) algorithms on heterogeneous platforms combining multi-core processors, many-core GPU and MIC coprocessors. As B&B is highly irregular its implementation is particularly challenging in computing environments where performance relies on SIMD processing and regular memory access patterns. In conjunction with the hardware architecture, the problem being solved strongly impacts the performance, and thus the design of B&B. Three well-known permutation-based problems, the Flowshop Scheduling Problem (FSP), the Quadratic Assignment Problem (QAP) and the n-Queens puzzle problem are used as case studies. These three elements, the architecture of HPC platforms, the characteristics of the problem being solved and the B&B algorithm itself, i. e. associated data structures and parallelization models constitute the frame of this work. The ideal situation of a B&B capable of solving many different combinatorial

optimization problems efficiently, exploiting a wide range of heterogeneous platform components, serves as a guiding point at the horizon.

The cornerstone of the proposed algorithms is the IVM data structure, which is used for the storage and management of subproblems, instead of conventional linked-list (LL) data structures. In IVM-based parallel B&B several independent exploration processes use their private IVM structure to explore parts of the search tree and exchange intervals of factoradics to achieve load balance.

As a first contribution we proposed a hybrid GPU-accelerated version of multi-core parallel B&B (GMC-B&B). The bounding operator is accelerated by asynchronously offloading the computation of lower bounds to the GPU. Four different work stealing strategies for IVM-based GMC-B&B were proposed. The approach is implemented for the FSP and QAP and, for comparison, the equivalent approach is implemented using a double-ended queue (deque) for the storage and management of subproblems. Also aiming at the acceleration of the bounding operator in MC-B&B, we revisit the FSP bounding procedure and propose a vectorizeable implementation of the bound, enabling MC-B&B to exploit the 512-bit vector processing units of Intel Xeon Phi MIC processors.

A second major contribution is the first implementation of the entire parallel tree exploration process, including load balancing, on the GPU. The proposed GPU-centric approach (GPU-B&B) requires minimal exchange of information between host and device and all B&B operators are performed massively in parallel on the GPU. We proposed two variants of GPU-B&B.

A first variant uses a two-level parallelization combining parallel tree exploration with the parallel evaluation of bounds model. It is designed for problems like FSP or QAP whose execution time is dominated by the bounding operator. The algorithm consists of alternating load balancing and exploration phases. The latter is implemented as a series of kernels which correspond to different operators and between which workers are implicitly synchronized. Two major challenges were identified and appropriate solutions were proposed. On the one hand, performance can be improved by choosing an efficient mapping of threads onto the data, such that detrimental effects resulting from thread divergence and irregular memory access patterns are alleviated. In particular, GPU-B&B handles the two levels of parallelism by introducing an efficient remapping phase at the interface of both levels. On the other hand, the implementation of all B&B operators on the device makes it necessary to design a GPU-based load balancing mechanism.

A second variant uses the parallel tree exploration model without a second parallelization level. It is designed for problems with inexpensive node evaluation functions as they may appear in DFS backtracking algorithms, where a relatively inexpensive

heuristic function is used to decide whether a subproblem is discarded or kept for exploration. This second variant is therefore called GPU-BT. In contrast to the two-level GPU-B&B algorithm, GPU-BT reduces kernel launch overhead and improves shared memory usage by implementing all operators in a single exploration kernel, and workers perform multiple iterations per kernel launch. When a critical level of idle workers is detected a trigger mechanism launches a load balancing phase which uses the same work stealing strategies as the two-level GPU-B&B. Our contribution for solving the issue of load imbalance are five GPU-based work stealing strategies, which vary in the underlying topology and the victim selection policy.

As a the third main contribution of this thesis we revisit the design of B&B@Grid, a distributed fault-tolerant B&B platform that uses interval-encoding to exchange work units between workers and the master-process. The result is HD-B&B, a hybrid distributed B&B-algorithm, integrating multi-core and many-core-based B&B-workers. HD-B&B can efficiently exploit the computing power provided by GPU-enhanced heterogeneous clusters for exactly solving COPs. The resolution of FSP instance *Ta056*, requiring 22 years of sequential execution time demonstrates this. Using B&B@Grid its resolution on 328 CPU cores requires 25 days, with near-perfect efficiency over 97%. Using HD-B&B on a cluster containing a total of 130 000 GPU cores, the resolution time for this instance is reduced to 9 hours.

One of the main objectives of this thesis was to investigate whether the IVM data structure can be used to build more efficient parallel B&B algorithms for heterogeneous large-scale computing platforms. The presented experimental results show that the answer to that question strongly depends on the input and the target platform. In general, the results show that fine-grained problems with computationally inexpensive node evaluation functions can benefit most from using IVM. For multi-core-based parallel B&B, acceleration factors greater than 3× are observed for the n-Queens problem, comparing the IVM-based algorithm to its LL-based counterpart. For more coarse-grained problems like FSP and QAP, the choice of the data structure, IVM or LL, has no significant impact on the execution time. This changes, when the bounding operator is accelerated by evaluating subproblems in parallel on the GPU. In this case, the IVM-based algorithm outperforms its LL-based counterpart by a factor 1.2×-2.6× for a set of FSP and QAP instances. As the node evaluation cost for n-Queens is already very low, there is no point in offloading this computation to GPU. Using the IVM data structure for storage and management of subproblems allows to implement all B&B operators on the GPU and completely bypass the CPU for computations. While this would have been impossible

- or highly inefficient - with LL-based data structures, the question is whether such a GPU-centric approach has significant advantages over hybrid CPU-GPU approaches. Again, the answer is clearly affirmative for fine-grained problems (e. g. n-Queens) as they can take the most benefit from massively parallelizing the selection, branching and pruning operators. For the FSP, whose associated bounding operator is well-suited for GPU processing, the GPU-centric B&B is about 1.7× faster than the offloading approach, solving 20-jobs-on-20-machines instances up to 1 000 times faster than a sequential algorithm (using 4 GPUs with total of 8 192 GPU cores). In contrast, the resolution of QAP instances benefits only marginally from implementing the entire algorithm on GPU, allowing an average improvement of 10%. This can be attributed to the fact that the SIMD-parallel evaluation of bounds for QAP-subproblems (using the GLB lower bound) is less efficient than for FSP.

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

- The experimental results obtained for the hybrid distributed B&B (HD-B&B) indicate that the algorithm is scalable on larger GPU-enhanced clusters. We plan to verify this by attempting the resolution of previously unsolved FSP instances on a large GPU-equipped supercomputer, like the current number three in the Top500 ranking, Piz Daint. In order to further improve scalability of the approach, the master process, usually running on a multi-core CPU, should be parallelized. Also, the checkpointing mechanism should be revisited. HD-B&B uses the checkpointing mechanism inherited from B&B@Grid, making the approach tolerant against node failures. However, as a large portion is shifted to lower levels it becomes important to make the approach fault-tolerant against failures at the GPU and multi-core level.

- Implementing the entire B&B process on the GPU leaves the host CPU cores available for other computations. In all configurations experimented in this thesis, using these cores concurrently with the GPU revealed inefficient. In the perspective of using GPU-B&B to solve multi-objective COPs these cores could be used to cooperate with the GPU-based exploration process, for instance for maintaining the Pareto archive.

- The IVM data structure revealed itself particularly well-suited for fine-grained permutation problems. For example, sampling methods based on the optimization of latin hypercubes can be modeled as (multi-)permutation problems. As a

future research direction we plan to revisit the IVM-based algorithm to enable the resolution of multi-permutation problems.

- The experimental results obtained for the hybrid distributed B&B indicate that the algorithm is scalable on larger GPU-enhanced clusters. We plan to verify this by increasing the number of used GPUs and attempt the resolution of previously unsolved FSP instances.

- Experimental results showed strong performance variations according to the used node evaluation function. Having different bounds for the same problem matching implementations with underlying hardware.

  A challenging improvement of the HD-B&B algorithm consists in implementing a library of lower bounds for the same problem, in order to enable the different workers to use the node evaluation function which is the best fit for the underlying hardware.

International Publications

International Journals

1. Gmys Jan, Leroy Rudy, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel, "Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms" in *Concurrency & Computation : Practice & Experience*, 28, 18, 4463-4484 (2016), https://doi.org/10.1002/cpe.3771.

2. Gmys Jan, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel,"A GPU-based Branch-and Bound algorithm using Integer-Vector-Matrix data structure", In *Parallel Computing, (Special Issue: Theory and Practice of Irregular Applications)*, Vol. 59, 2016, p. 119-139, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2016.01.008

3. Gmys Jan, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel, "IVM-Based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems" in *Concurrency & Computation : Practice & Experience, (Special Edition PPAM'15)* (2016), 29(9), https://doi.org/10.1002/cpe.4019

4. Melab Nouredine, Gmys Jan, Mezmaz Mohand, Tuyttens Daniel, "Multi-core versus many-core computing for many-task Branch-and-Bound applied to big optimization problems" in *Future Generation Computer Systems* (2017), https://doi.org/10.1016/j.future.2016.12.039

5. Pessoa Tiago Carneiro, Gmys Jan, de Carvalho Junior Francisco Heron, Melab Nouredine, Tuyttens Daniel, *"GPU-Accelerated Backtracking Using CUDA Dynamic Parallelism"* in Concurrency & Computation: Practice & Experience, Https://doi:10.1002/cpe.4374, (2017, accepted, in production)

International Conferences

1. Gmys Jan, Mezmaz Mohand, Melab Nouredine, Tuyttens Daniel, "IVM-based Work Stealing for Parallel Branch-and-Bound on GPU" in *Parallel Processing and Applied Mathematics (PPAM'15). Lecture Notes in Computer Science*, 9573, 548-558 (2016), https://doi.org/10.1007/978-3-319-32149-3_51 [**Best Paper Award** in workshop on GPU computing at PPAM'15]

2. Pessoa Tiago Carneiro, Gmys Jan, Melab Nouredine, de Carvalho Junior Fransisco Heron, Tuyttens Daniel, "A GPU-based Backtracking Algorithm for Permutation Combinatorial Problems" in *International Conference on Algorithms and Architectures*

for Parallel Processing (ICA3PP'16). Lecture Notes in Computer Science, 10048, 310-324 (2016), https://doi.org/10.1007/978-3-319-49583-5_24

Bibliography

[16] *TOP500.org*, 2016. [Online]. Available: http://www.top500.org.

[ABE+16] D. Adel, A. Bendjoudi, D. El-Baz, A. Z. Abdelhakim, *et al.*, "Gpu-based two level parallel b&b for the blocking job shop scheduling problem.," 2016. [Online]. Available: http://dl.cerist.dz/handle/CERIST/794.

[ABGL02] K. Anstreicher, N. Brixius, J.-P. Goux, and J. Linderoth, "Solving large quadratic assignment problems on computational grids," *Mathematical Programming*, vol. 91, no. 3, pp. 563–588, 2002.

[ABP01] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *Theory of Computing Systems*, vol. 34, no. 2, pp. 115–144, 2001.

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

[ACR13] U. A. Acar, A. Chargueraud, and M. 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*, ser. PPoPP '13, Shenzhen, China: ACM, 2013, p. 10. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442538.

[ASW+14] T.-H. Ahn, A. Sandu, L. T. Watson, C. A. Shaffer, Y. Cao, and W. T. Baumann, "A framework to analyze the performance of load balancing schemes for ensembles of stochastic simulations," *International Journal of Parallel Programming*, vol. 43, no. 4, pp. 597–630, 2014.

[BHG15] A. Borisenko, M. Haidl, and S. Gorlatch, "Parallelizing branch-and-bound on gpus for optimization of multiproduct batch plants," in *International Conference on Parallel Computing Technologies*, Springer, 2015, pp. 324–337.

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

[BKR97] R. E. Burkard, S. E. Karisch, and F. Rendl, "Qaplib–a quadratic assignment problem library," *Journal of Global optimization*, vol. 10, no. 4, pp. 391–403, 1997.

[BL99] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[BMT12a] A. Bendjoudi, N. Melab, and E. .-.-G. Talbi, "Hierarchical branch and bound algorithm for computational grids," *Future Gener. Comput. Syst.*, vol. 28, no. 8, pp. 1168–1176, Oct. 2012, ISSN: 0167-739X. DOI: 10.1016/j.future.2012.03.001. [Online]. Available: http://dx.doi.org/10.1016/j.future.2012.03.001.

[BMT12b] A. Bendjoudi, N. Melab, and E.-G. Talbi, "An adaptive hierarchical master–worker (ahmw) framework for grids—application to b&b algorithms," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 120–131, 2012.

[BMT14] A. Bendjoudi, N. Melab, and E.-G. Talbi, "Fth-b&b: A fault-tolerant hierarchicalbranch and bound for large scaleunreliable environments," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2302–2315, 2014.

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

[Can69] G. Cantor, *Ueber die einfachen zahlensysteme*, 1869.

[Cap09] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[CGG+14] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing frontiers and innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[Cha13] I. Chakroun, "Parallel heterogeneous branch and bound algorithms for multi-core and multi-gpu environments," PhD thesis, Université Lille 1, 2013.

[CMGH08] L. G. Casado, J. Martinez, I. García, and E. M. Hendrix, "Branch-and-bound interval global optimization on shared memory multiprocessors," *Optimization Methods & Software*, vol. 23, no. 5, pp. 689–701, 2008.

[CMMB13] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 8, pp. 1121–1136, 2013, ISSN: 1532-0634. DOI: 10.1002/cpe.2931.

[CMMT13] I. Chakroun, N. Melab, M. Mezmaz, and D. Tuyttens, "Combining multi-core and gpu computing for solving combinatorial optimization problems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1563–1577, 2013.

[CMNL11] T. Carneiro, A. Muritiba, M. Negreiros, and G. 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, pp. 41–47. DOI: 10.1109/SBAC-PAD.2011.20.

[CNNdC12] T. Carneiro, R. H. Nobre, M. Negreiros, and G. A. L. de Campos, "Depth-first search versus Jurema search on GPU branch-and-bound algorithms: A case study," *NVIDIA's GCDF - GPU Computing Developer Forum on XXXII Congresso da Sociedade Brasileira de Computação (CSBC)*, 2012, ISSN: 2175-2761.

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

[CP99] J. Clausen and M. Perregaard, "On the best search strategy in parallel branch-and-bound:best-first search versus lazy depth-first search," *Annals of Operations Research*, vol. 90, no. 0, pp. 1–17, Jan. 1999, ISSN: 1572-9338. DOI: 10.1023/A:1018952429396. [Online]. Available: https://doi.org/10.1023/A:1018952429396.

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

[DKT95] A. De Bruin, G. A. Kindervater, and H. W. Trienekens, "Asynchronous parallel branch and bound and anomalies," in *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Springer, 1995, pp. 363–377.

[DLS+09] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis, Article No. 53*, ser. SC '09, New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654113.

[Dre07] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, p. 2007, 2007.

[EBS+11] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 39, 2011, pp. 365–376.

[EHP15] J. Eckstein, W. E. Hart, and C. A. Phillips, "Pebbl: An object-oriented framework for scalable parallel branch and bound," *Mathematical Programming Computation*, vol. 7, no. 4, pp. 429–469, 2015.

[EPH01] J. Eckstein, C. A. Phillips, and W. E. Hart, "Pico: An object-oriented framework for parallel branch and bound," *Studies in Computational Mathematics*, vol. 8, pp. 219–265, 2001.

[EPS09] Y. Evtushenko, M. Posypkin, and I. Sigal, "A framework for parallel large-scale global optimization," *Computer Science-Research and Development*, vol. 23, no. 3-4, pp. 211–215, 2009.

[FLR98] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proc. of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98, New York, NY, USA: ACM, 1998, p. 11. [Online]. Available: http://doi.acm.org/10.1145/277650.277725.

[FRvLP10] F. Feinbube, B. Rabe, M. von Löwis, and A. Polze, "Nqueens on cuda: Optimization issues," in *Parallel and Distributed Computing (ISPDC), 2010 Ninth International Symposium on*, IEEE, 2010, pp. 63–70.

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

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

[GJS76] M. R. Garey, D. S. Johnson, and R. Sethi, "The Complexity of Flowshop and Jobshop Scheduling," English, *Mathematics of Operations Research*, vol. 1, no. 2, pp. 117-129, 1976, ISSN: 0364765X.

[GLM+16] J. Gmys, R. Leroy, M. Mezmaz, N. Melab, and D. Tuyttens, "Work stealing with private integer–vector–matrix data structure for multi-core branch-and-bound algorithms," *Concurrency and Computation: Practice and Experience*, n/a–n/a, 2016, cpe.3771, ISSN: 1532-0634. DOI: 10.1002/cpe.3771. [Online]. Available: http://dx.doi.org/10.1002/cpe.3771.

[GR14] M. Giles and I. Reguly, "Trends in high-performance computing for engineering calculations," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2022, p. 20 130 319, 2014.

[GSO12] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," in *Innovative Parallel Computing (InPar), 2012*, IEEE, 2012, pp. 1–14.

[HSH+17] J. F. R. Herrera, J. M. G. Salmerón, E. M. T. Hendrix, R. Asenjo, and L. G. Casado, "On parallel branch and bound frameworks for global optimization," *Journal of Global Optimization*, Mar. 2017, ISSN: 1573-2916. DOI: 10.1007/s10898-017-0508-y. [Online]. Available: https://doi.org/10.1007/s10898-017-0508-y.

[HSO07] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," in *GPU Gems 3*, H. Nguyen, Ed., Addison Wesley, Aug. 2007.

[JAO+11] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons learned from exploring the backtracking paradigm on the gpu," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par'11, Bordeaux, France: Springer-Verlag, 2011, pp. 425–437, ISBN: 978-3-642-23396-8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2033408.2033458.

[Joh54] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954, ISSN: 1931-9193. DOI: 10.1002/nav.3800010110.

[KB57] T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica: journal of the Econometric Society*, pp. 53–76, 1957.

[KDK+11] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011, ISSN: 0272-1732. DOI: 10.1109/MM.2011.89.

[KGKH13] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2013, pp. 56–65. DOI: 10.1109/IISWC.2013.6704670.

[KK84] V. Kumar and L. N. Kanal, "Parallel branch-and-bound formulations for and/or tree search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 6, no. 6, pp. 768–778, Nov. 1984, ISSN: 0162-8828. DOI: 10.1109/TPAMI.1984.4767600. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.1984.4767600.

[KK94] G. Karypis and V. Kumar, "Unstructured tree search on simd parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1057–1072, 1994.

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

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

[Lai88] C.-A. Laisant, "Sur la numération factorielle, application aux permutations," *Bulletin de la Société Mathématique de France*, vol. 16, pp. 176–183, 1888.

[LdAB+07] E. M. Loiola, N. M. M. de Abreu, P. O. Boaventura-Netto, P. Hahn, and T. Querido, "A survey for the quadratic assignment problem," *European journal of operational research*, vol. 176, no. 2, pp. 657–690, 2007.

[LE12] M. Lalami and D. El-Baz, "GPU Implementation of the Branch and Bound Method for Knapsack Problems," in *IEEE 26th Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, Shanghai, CHN, May 2012, pp. 1769–1777. DOI: 10.1109/IPDPSW.2012.219.

[Ler15] R. Leroy, "Parallel branch-and-bound revisited for solving permutation combinatorial optimization problems on multi-core processors and coprocessors," PhD thesis, Université Lille 1, 2015.

[Li16] A. Li, "Gpu performance modeling and optimization," PhD thesis, Technische Universiteit Eindhoven, 2016.

[LLK78] B. J. Lageweg, J. K. Lenstra, and A. H. G. R. Kan, "A General Bounding Scheme for the Permutation Flow-Shop Problem," *Operations Research*, vol. 26, no. 1, pp. 53–67, 1978. DOI: 10.1287/opre.26.1.53. eprint: http://dx.doi.org/10.1287/opre.26.1.53.

[LLW+15a] L. Li, H. Liu, H. Wang, T. Liu, and W. Li, "A parallel algorithm for game tree search using gpgpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2114–2127, Aug. 2015, ISSN: 1045-9219. DOI: 10.1109/TPDS.2014.2345054.

[LLW+15b] L. Li, H. Liu, H. Wang, T. Liu, and W. Li, "A parallel algorithm for game tree search using GPGPU," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 8, pp. 2114–2127, 2015.

[LMMT07] 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 *Proceedings of Programming Models and Applications on Multicores and Manycores*, ser. PMAM'14, Orlando, FL, USA: ACM, 2007, 111:111–111:119, ISBN: 978-1-4503-2657-5. DOI: 10.1145/2560683.2560694.

[LPRR94] Y. Li, P. M. Pardalos, K. Ramakrishnan, and M. G. Resende, "Lower bounds for the quadratic assignment problem," *Annals of Operations Research*, vol. 50, no. 1, pp. 387–410, 1994.

[Mär14] C. Märtin, "Multicore processors: Challenges, opportunities, emerging trends," in *Proceedings of Embedded World Conference, Germany*, 2014. [Online]. Available: https://www.hs-augsburg.de/Binaries/Binary20964/Multicore-Embeddedfinal-revised.pdf.

[MCA13] X. Meyer, B. Chopard, and P. Albuquerque, "A branch-and-bound algorithm using multiple gpu-based lp solvers," in *20th Annual International Conference on High Performance Computing*, Dec. 2013, pp. 129–138. DOI: 10.1109/HiPC.2013.6799105.

[MCB14] N. Melab, I. Chakroun, and A. 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, pp. 2667–2683, 2014, ISSN: 1532-0634. DOI: 10.1002/cpe.3155.

[Meh11] M. Mehdi, "Parallel hybrid optimization methods for permutation based problems," PhD thesis, Université du Luxembourg / Université Lille 1, 2011.

[Mel05] N. Melab, *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul*, LIFL, USTL, Thèse HDR, Nov. 2005.

[Men17] T. Menouer, "Solving combinatorial problems using a parallel framework," *Journal of Parallel and Distributed Computing*, 2017, ISSN: 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2017.05.019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517301764.

[MLMT14] M. Mezmaz, R. Leroy, N. Melab, and D. Tuyttens, "A Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System," in *28th IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS)*, Phoenix, AZ: May 2014, pp. 1203–1212. DOI: 10.1109/IPDPS.2014.124.

[MMT07a] M. Mezmaz, N. Melab, and E. G. Talbi, "A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–9. DOI: 10.1109/IPDPS.2007.370217.

[MMT07b] M. Mezmaz, N. Melab, and E.-G. Talbi., "A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems," in *21th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, Long Beach, CA: Mar. 2007, pp. 1–9.

[MR90] B. Mans and C. Roucairol, "Concurrency in priority queues for branch and bound algorithms," INRIA, Tech. Rep. RR-1311, Oct. 1990, Projet PARADIS. [Online]. Available: https://hal.inria.fr/inria-00075248.

[MV15] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, 69:1–69:35, Jul. 2015, ISSN: 0360-0300. DOI: 10.1145/2788396. [Online]. Available: http://doi.acm.org/10.1145/2788396.

[OHL+08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

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

[PE17] T. B. Preußer and M. R. Engelhardt, "Putting queens in carry chains, n ọ27," *Journal of Signal Processing Systems*, vol. 88, no. 2, pp. 185–201, Aug. 2017, ISSN: 1939-8115. DOI: 10.1007/s11265-016-1176-8. [Online]. Available: https://doi.org/10.1007/s11265-016-1176-8.

[Ric97] M. Richards, "Backtracking algorithms in mcpl using bit patterns and recursion," University of Cambridge, Computer Laboratory, Tech. Rep., 1997. [Online]. Available: http://www.cl.cam.ac.uk/~mr10/backtrk.pdf.

[RK93] V. N. Rao and V. Kumar, "On the efficiency of parallel backtracking," *IEEE Transactions on parallel and distributed systems*, vol. 4, no. 4, pp. 427–437, 1993.

[RLS04] T. K. Ralphs, L. Ladányi, and M. J. Saltzman, "A library hierarchy for implementing scalable parallel search algorithms," *The Journal of Supercomputing*, vol. 28, no. 2, pp. 215–234, 2004.

[Rou87] C. Roucairol, "A parallel branch and bound algorithm for the quadratic assignment problem," *Discrete Applied Mathematics*, vol. 18, no. 2, pp. 211–225, 1987.

[RS10] K. Rocki and R. Suda, "Parallel minimax tree searching on gpu," in *Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 449–456, ISBN: 978-3-642-14390-8. DOI: 10.1007/978-3-642-14390-8_47. [Online]. Available: https://doi.org/10.1007/978-3-642-14390-8_47.

[Sar10] H. Sarbazi-Azad, *Stupid Sort: A new sorting algorithm*, Newsletter. Computing Science Department, Univ. of Glasgow (599), [Accessed October,13 2017], Oct. 2010. [Online]. Available: http://sharif.edu/~azad/stupid-sort.PDF.

[SG97] R. Sakellariou and J. R. Gurd, "Compile-time minimisation of load imbalance in loop nests," in *Proceedings of the 11th international conference on Supercomputing*, ACM, 1997, pp. 277–284.

[Som] J. Somers, *The N Queens Problem - a study in optimization*, http://users.rcn.com/liusomers/nqueen_demo/nqueens.html, Accessed: 2017-10-08.

[SRR08] P. San Segundo, D. Rodríguez-Losada, and C. Rossi, "Recent developments in bit-parallel algorithms," in *Tools in Artificial Intelligence*, InTech, 2008.

[Sun+03] X.-H. Sun *et al.*, "Improving the performance of mpi derived datatypes by optimizing memory-access cost," in *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, IEEE, 2003, pp. 412–419.

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

[Tal09] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, 1st ed. John Wiley & Sons, Inc., Jul. 2009, ISBN: 978-0-470-27858-1.

[TdB92] H. W. Trienekens and A. de Bruin, "Towards a taxonomy of parallel branch and bound algorithms," 1992.

[VDM13] T.-T. Vu, B. Derbel, and N. Melab, "Adaptive Dynamic Load Balancing in Heterogenous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search," in *7th International Learning and Intelligent OptimizatioN Conference (LION)*, Catania, Italy: Lecture Notes in Computer Science, Jan. 2013. [Online]. Available: https://hal.inria.fr/hal-00765199.

[vDvdP14] T. van Dijk and J. C. van de Pol, "Lace: Non-blocking split deque for work-stealing," in *European Conference on Parallel Processing*, Springer, 2014, pp. 206–217.

[VH15] M. Vinkler and V. Havran, "Register efficient dynamic memory allocator for gpus," *Computer Graphics Forum*, vol. 34, no. 8, pp. 143–154, 2015, ISSN: 1467-8659. DOI: 10.1111/cgf.12666. [Online]. Available: http://dx.doi.org/10.1111/cgf.12666.

[Vol16] V. Volkov, "Understanding latency hiding on gpus," PhD thesis, University of California, Berkeley, 2016.

[WWWG13] S. Widmer, D. Wodniok, N. Weber, and M. Goesele, "Fast dynamic memory allocator for massively parallel architectures," in *Proceedings of the 6th workshop on general purpose processor using graphics processing units*, ACM, 2013, pp. 120–126.

[XF10] S. Xiao and W.-c. Feng, "Inter-block gpu communication via fast barrier synchronization," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE, 2010, pp. 1–12.

[YH17] J. Yang and Q. He, "Scheduling parallel computations by work stealing: A survey," *International Journal of Parallel Programming*, Jan. 2017, ISSN: 1573-7640. DOI: 10.1007/s10766-016-0484-8. [Online]. Available: https://doi.org/10.1007/s10766-016-0484-8.

[Zon02] Q. Zongyan, "Bit-vector encoding of n-queen problem," *SIGPLAN Not.*, vol. 37, no. 2, pp. 68–70, Feb. 2002, ISSN: 0362-1340. DOI: 10.1145/568600.568613. [Online]. Available: http://doi.acm.org/10.1145/568600.568613.

[ZSW11] T. Zhang, W. Shu, and M.-Y. Wu, "Optimization of n-queens solvers on graphics processors," in *Proceedings of the 9th International Conference on Advanced Parallel Processing Technologies*, ser. APPT'11, Shanghai, China: Springer-Verlag, 2011, pp. 142–156, ISBN: 978-3-642-24150-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042522.2042533.

List of Figures

1.1 Illustration: Parallelization models . . . 16
1.2 Hardware view: GPU = coprocessor of CPU. . . . 23
1.3 Software view: Parallel program = weakly parallel/serial host code + massively parallel device code. . . . 24
1.4 Software view: Parallel program = grid(s) of block(s) of threads executed as warps of 32 threads. . . . 25
1.5 Flowshop illustration . . . 31
1.6 Illustration: QAP . . . 33
1.7 Illustration: n-Queens . . . 34
1.8 B&B tree irregularity . . . 35
1.9 Node evaluation irregularity : QAP, FSP . . . 37
1.10 Node evaluation irregularity : n-Queens . . . 38

2.1 Tree representations: IVM/LL . . . 42
2.2 IVM-based B&B: selection operator . . . 43
2.3 IVM-based B&B: branching operator . . . 44
2.4 Illustation: interval-based parallel tree exploration . . . 47
2.5 Illustration: Communication of work units . . . 48
2.6 Illustration: GPU-accelerated B&B . . . 57
2.7 Multi-core FSP/QAP: IVM vs. LL-based B&B . . . 60
2.8 Multi-core n-Queens: IVM vs. LL-based B&B . . . 61
2.9 GMC-B&B: Tuning pool size . . . 62
2.10 GPU-MC-B&B: IVM vs. LL . . . 64
2.11 MC-B&B: Parallel efficiency on Xeon Phi (FSP/QAP) . . . 69
2.12 MC-B&B: Parallel efficiency on Xeon Phi (15-Queens) . . . 70
2.13 MC-B&B: Scalability . . . 71

3.1 Flowchart of GPU-centric B&B algorithm. . . . 82

3.2 Illustration: parallel evaluation of bounds in GPU-B&B 85
3.3 Illustration of the remapping phase for the bounding kernel (Algorithm 7). 89
3.4 GPU-B&B - mapping the bounding kernel: Ta021-Ta030 98
3.5 GPU-B&B - mapping IVM-management kernels : *Ta021-Ta030* 100
3.6 GPU-B&B work stealing strategies: overhead analysis 103
3.7 Multi-GPU-B&B: Calibration of number of IVMs (Adapt vs. Hypercube, FSP/n-Queens) 104
3.8 Multi-GPU-B&B: Calibration of number of IVMs (Adapt vs. Hypercube, QAP) 105
3.9 Calibration of trigger mechanism 106
3.10 Multi-GPU-B&B: scaling 108
3.11 Hybrid CPU-GPU-B&B : Workload monitoring 113

4.1 Illustration of Master-Worker model in B&B@Grid 119
4.2 Illustration of HB-B&B 120
4.3 Flowchart of worker process in HD-B&B 126
4.4 Resolution of 50-job FSP instance *Ta056* 129
4.5 HD-B&B: Ganglia energy monitoring 130
4.6 HD-B&B: Resolution of *Ta056* - Work repartition 131
4.7 Scale HD-B&B 133
4.8 HD-B&B: Impact of instance size 135
4.9 Partial tree size for 50 × 20 FSP instances 138

List of Tables

1.1 Sequential execution time *Ta021-Ta030* 32

2.1 MC-B&B : Comparison of work stealing strategies 66
2.2 GMC-B&B : Comparison of work stealing strategies 68
2.3 Comparison of vectorized/non-vectorized FSP lower bound 68

3.1 GPU-B&B - mapping for bounding kernel: time breakdown 99
3.2 IVM-management: instruction replay overhead 100
3.3 Profiling thread divergence: *Ta022* 101
3.4 Comparison of GPU-B&B work stealing strategies : Execution time *Ta021-Ta030* 102
3.5 Comparison of GPU-B&B work stealing strategies : IVM-efficiency . . . 102
3.6 multi-GPU-B&B: Performance comparison with MC-B&B and GMC-B&B - FSP 109
3.7 multi-GPU-B&B: Performance comparison with MC-B&B and GMC-B&B - QAP 110
3.8 GPU-B&B vs GMC-B&B : execution time breakdown 111
3.9 multi-GPU-B&B: Performance comparison with MC-B&B and GMC-B&B - *n*-Queens 112
3.10 Averaged execution times for FSP instance *Ta028* (100 executions) for comparing the multi-GPU only and the hybrid algorithm (using 16 CPU threads) 113

4.1 Redundant exploration in HD-B&B 123
4.2 Exploration statistics for resolution of FSP instance *Ta056* 132
4.3 HD-B&B: Resolution of *Ta022* using 20 Tegra K1 SoC 137

A.1 Number of decomposed nodes in critical tree (B&B initialized at optimal solution) I
A.2 Complexities and memory requirements : node evaluation II

Appendix A

Appendix

A.1 Tree sizes

Table A.1: Number of decomposed nodes in critical tree (B&B initialized at optimal solution)

instance	#decomposed nodes	instance	#decomposed nodes
Ta021	41 417 881	*nug16a*	841 732
Ta022	22 068 771	*nug16b*	444 579
Ta023	140 848 940	*nug17*	4 817 638
Ta024	40 067 821	*nug18*	24 971 333
Ta025	41 440 440	*nug20*	362 626 645
Ta026	71 376 390	*chr20c*	5 418 529
Ta027	57 111 463	*tai17a*	7 809 792
Ta028	8 088 505	*esc16a*	96 305 057
Ta029	6 778 450	*esc16c*	356 056 705
Ta030	1 648 102	*esc16d*	13 375 109
14-Queens	13 496 479	*esc16e*	768 344 897
15-Queens	90 634 738	*esc16g*	301 589 057
16-Queens	563 208 896	*had18*	3 145 954
17-Queens	4 224 112 371	*had20*	69 910 557
18-Queens	29 349 876 934	*scr20*	25 337 809
19-Queens	242 419 099 083	*rou20*	1 371 489 830

A.2 Lower bounds: complexities

Table A.2: Computational complexities and memory requirements of node evaluation functions used for FSP, QAP and n-Queens. m = #machines, n = problem size

	FSP	QAP	n-Queens
Complexity	$O(n^2 m \log m)$	$O(n^3)$	$O(n)$
Read-only memory ($\times\|int\|$)	$(n+1)m^2 - m$	$2n^2$	–
Read-Write memory ($\times\|int\|$)	$\sim 4n$	$\sim n^2$	1

A.3 Hardware

The following devices are used in the experimental evaluations.

- **Intel Xeon E5-2630v3:** (Haswell), 8 cores, 16 threads, 2.40 GHz base frequency, 20 MB L3 Cache, 85 W TDP, 59 GB/s max. memory bandwidth. AVX 2.0 vector instruction extensions, 22 nm technology.
- **Intel Xeon E5-2680v4:** (Broadwell), 14 cores, 28 threads, 2.40 GHz base frequency, 35 MB L3 Cache, 120 W TDP, 76.8 GB/s max. memory bandwidth. AVX 2.0 vector instruction extensions, 14 nm technology.
- **Intel Xeon Phi 5110P:** (Knight's Corner), 60 cores, 240 threads, 1.053 GHz, 30 MB L2 cache, 225 W TDP, 320 GB/s max. memory bandwidth. IMCI (512-bit) vector instruction extensions, 22 nm technology.
- **IBM Power System (S822LC "Minsky" node)**
  - 2×**Power8+:** 10 cores/CPU, 80 threads/CPU (SMT8), 2.86 GHz, 8 MB L3 / 16 MB L4 Cache, 225 W TDP, 230 GB/s max. memory bandwidth 128 Go, 22 nm technology.
  - **Nvidia Tesla P100** GP100 (Pascal), 3 584 CUDA cores, 1328 MHz base clock, 16 GB VRAM, HBM2 memory, 720 GB/s max. memory bandwidth, 300 W TDP, 16 nm technology.
- **Nvidia Tesla K20m:** GK110 (Kepler), 2 496 CUDA cores, 705 MHz base clock, 225 W TDP
- **Nvidia GeForce GTX 980:** GM204 (Maxwell), 2 048 CUDA cores, 1126 MHz base clock, 165 W TDP
- **Nvidia GeForce GTX 1080Ti:** GP102 (Pascal), 3 584 CUDA cores, 1481 MHz base clock, 250 W TDP, 11 GB VRAM, 22 nm technology.

- **Nvidia Tegra K1** System on a chip, 32-bit quad-core ARM Cortex-A15 MPCore R3, Kepler GPU (1 SM, 192 CUDA cores), TDP 8 Watt.