PhD-FSTM-2025-009
Faculty of Science, Technology and Medicine
Faculty of Science and Technology

DISSERTATION

Defence held on 10 January 2025 in Esch-sur-Alzette (Luxembourg)

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

AND

DOCTEUR DE L'UNIVERSITÉ DE LILLE EN INFORMATIQUE ET APPLICATIONS

by

Guillaume HELBECQUE
Born on 12 August 1998 in Valenciennes (France)

PGAS-based Parallel Branch-and-Bound for Ultra-Scale GPU-powered Supercomputers

Dissertation defence committee:

Dr Pascal BOUVRY, Co-Supervisor
Dean of the Faculty of Science, Technology and Medicine, UNIVERSITÉ DU LUXEMBOURG

Dr Nouredine MELAB, Co-Supervisor
Full professor in Computer Science, UNIVERSITÉ DE LILLE

Dr Nicolas NAVET, Chair
Full professor in Computer Science, UNIVERSITÉ DU LUXEMBOURG

Dr Imen CHAKROUN, Vice-Chair
Senior researcher, IMEC

Dr Frédéric SAUBION, Member
Full professor in Computer Science, UNIVERSITÉ D'ANGERS

Dr Enrique ALBA, Member
Full professor in Computer Science, UNIVERSITY OF MALAGA

PhD-FSTM-2025-009
Faculté des Sciences, des Technologies et de Médecine

Faculté des Sciences et Technologies

THÈSE

Soutenue le 10 janvier 2025 à Esch-sur-Alzette (Luxembourg)

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

ET

DOCTEUR DE L'UNIVERSITÉ DE LILLE EN INFORMATIQUE ET APPLICATIONS

par

Guillaume HELBECQUE
Né le 12 août 1998 à Valenciennes (France)

Branch-and-Bound Parallèle Basé sur PGAS pour les Supercalculateurs Ultra-Scale Dotés de GPUs

Comité de soutenance de thèse :

Dr Pascal BOUVRY, Co-Directeur
Doyen de la Faculté des Sciences, des Technologies et de Médecine, UNIVERSITÉ DU LUXEMBOURG

Dr Nouredine MELAB, Co-Directeur
Professeur en Informatique, UNIVERSITÉ DE LILLE

Dr Nicolas NAVET, Président
Professeur en Informatique, UNIVERSITÉ DU LUXEMBOURG

Dr Imen CHAKROUN, Examinatrice
Chercheur senior, IMEC

Dr Frédéric SAUBION, Rapporteur
Professeur en Informatique, UNIVERSITÉ D'ANGERS

Dr Enrique ALBA, Rapporteur
Professeur en Informatique, UNIVERSITY OF MALAGA

À ma mère, à mon père

Abstract

Branch-and-Bound (B&B) algorithms are widely used for the exact resolution of many combinatorial optimization problems. Their parallel implementation to solve increasingly large instances presents several challenges related to the dynamic generation of large, highly irregular trees. With the advent of the exascale *era*, modern supercomputers are now composed of thousands of hybrid compute nodes, each integrating multi-core processors coupled with GPU accelerators. This hierarchical organization, providing multi-level parallelism (intra-node, GPU, inter-node or cluster, *etc.*), makes exascale parallel implementation complex. To address this complexity, most existing works employ the "evolutionary" MPI+X approach, which extends the MPI standard used for inter-node level with environments for intra-node parallelism (OpenMP, CUDA, *etc.*). In this thesis, we investigate the PGAS (Partitioned Global Address Space) approach, an alternative to MPI+X, in the context of implementing B&B algorithms for exascale. This "revolutionary" approach provides a higher level of parallelism abstraction, unifying intra-node and inter-node levels.

The first contribution of this thesis focuses on the design and implementation of a PGAS-based data structure, named `distBag_DFS`, dedicated to depth-first exploration of large, irregular trees. This multi-pool data structure integrates a dynamic load-balancing mechanism based on large-scale work-stealing, operating at both intra- and inter-node levels. This mechanism, which required sophisticated synchronization, promotes locality in work-stealing, enabling scalability. The data structure and its load-balancing mechanism are implemented in Chapel and provided as a module in this PGAS-based language designed for exascale. The second contribution of this thesis extends the proposed work to the multi-GPU context to accelerate the extensive and costly evaluation of tree nodes being explored. The challenge of ensuring implementation portability across multi-vendor GPU architectures (NVIDIA and AMD) is addressed.

The algorithms developed in this thesis have been designed to be generic and to promote their reuse. This is evidenced by the application of these algorithms to various combinatorial optimization problems, including permutation flow-shop scheduling, binary knapsack problems, the N-Queens problem, and the Unbalanced Tree Search benchmark. Experimental validation was conducted on two TOP500 supercomputers (MeluXina and LUMI), among others. The results show that our PGAS-based algorithms are competitive in terms of scalability at both intra-node and inter-node levels compared to those obtained with the MPI+X approach. Additionally, the results con-

firmed the optimality of solutions for some of the largest flow-shop instances, utilizing up to 400 compute nodes, or 51,200 CPU cores. Furthermore, scalability concerning the number of GPUs was evaluated on 128 compute nodes, totaling 1,024 GPU accelerators. Overall, our results demonstrate the competitiveness of PGAS approaches compared to MPI+X, while identifying opportunities for further enhancement.

Keywords:

Parallel Branch-and-Bound, Combinatorial optimization, PGAS programming, Chapel, Ultra-scale supercomputers, GPU

Résumé

Les algorithmes Branch-and-Bound (B&B) sont couramment utilisés pour la résolution exacte de nombreux problèmes d'optimisation combinatoire. Leur mise en œuvre parallèle pour la résolution d'instances de plus en plus grandes pose plusieurs défis liés à la génération dynamique de grands arbres fortement irréguliers. Avec l'arrivée de l'ère exascale, les supercalculateurs modernes sont désormais composés de milliers de nœuds de calcul hybrides, chacun intégrant des processeurs multi-cœurs couplés à des accélérateurs graphiques (GPUs). Cette organisation hiérarchique, fournissant un parallélisme multi-niveau (intra-nœud, GPU, inter-nœud ou cluster, *etc.*), rend complexe l'implémentation parallèle exascale. Pour faire face à cette complexité, la majorité des travaux existants utilise l'approche « évolutionnaire » MPI+X, qui consiste à étendre le standard MPI utilisé pour le niveau inter-noeud avec des environnements pour le parallélisme intra-nœud (OpenMP, CUDA, *etc.*). Dans cette thèse, nous investiguons l'approche PGAS (Partitioned Global Address Space), alternative à MPI+X, dans le contexte de la mise en œuvre des algorithmes B&B pour l'exascale. Cette approche « révolutionnaire » fournit un niveau d'abstraction du parallélisme plus élevé, unifiant les niveaux intra-nœud et inter-nœud.

La première contribution de cette thèse porte sur la conception et l'implémentation d'une structure de données PGAS, nommée `distBag_DFS`, dédiée à l'exploration en profondeur d'abord d'arbres irréguliers de grande taille. Cette structure de données multi-pool intègre un mécanisme d'équilibrage de charge dynamique basé sur le paradigme de vol de tâches large échelle, opéré aux deux niveaux intra- et inter-nœud. Ce mécanisme, qui a nécessité une synchronisation sophistiquée, favorise la localité des vols de tâches permettant son passage à l'échelle. La structure de données et son mécanisme d'équilibrage de charge sont implémentés en Chapel, et fournis comme module dans ce langage basé sur PGAS et conçu pour l'exascale. La deuxième contribution de cette thèse porte sur l'extension des travaux proposés au contexte multi-GPU pour accélérer l'évaluation massive et coûteuse des nœuds de l'arbre exploré. Le défi de la portabilité de l'implémentation sur architectures GPU multi-fournisseurs (NVIDIA et AMD) est considéré.

Les algorithmes développés dans cette thèse ont été conçus pour être génériques et favoriser leur réutilisation. Ceci est attesté par l'application de ces algorithmes à différents problèmes d'optimisation combinatoire, notamment les problèmes d'ordonnancement Flow-Shop à permutation, de sac à dos binaire, des N-reines ainsi que le benchmark Unbalanced Tree Search. La validation expérimentale a été réalisée, entre autres, sur deux

supercalculateurs du classement TOP500 (MeluXina et LUMI). Les résultats obtenus montrent que nos algorithmes basés sur l'approche PGAS sont compétitifs, en termes de passage à l'échelle aux deux niveaux intra- et inter-nœud, en comparaison de ceux obtenus avec l'approche MPI+X. De plus, les résultats ont confirmé l'optimalité des solutions pour certaines des plus grandes instances du Flow-Shop, en utilisant jusqu'à 400 nœuds de calcul, soit 51 200 cœurs CPU. D'autre part, le passage à l'échelle par rapport au nombre de GPU a été évalué sur 128 nœuds de calcul, totalisant 1 024 accélérateurs GPU. De manière générale, nos résultats montrent la compétitivité des approches PGAS par rapport à MPI+X, tout en mettant en lumière certaines perspectives d'amélioration.

Mots-clés :

Branch-and-Bound parallèle, Optimisation combinatoire, Programmation PGAS, Chapel, Supercalculateurs ultra-scale, GPU

Acknowledgments

My first acknowledgments naturally go to my two thesis supervisors, Prof. Nouredine MELAB and Prof. Pascal BOUVRY. I am deeply grateful for the unwavering support they have provided me throughout these years. Their expertise, availability, and trust were decisive elements in the completion of this thesis. I have learned a great deal from them, both scientifically and personally.

Since this thesis is also the result of many collaborations, I would like to thank my main collaborators, Jan GMYS, Tiago CARNEIRO, and Ezhilmathi KHRISNASAMY, with whom I had the opportunity to work on this project. Each of them contributed uniquely to the advancement of my research through their ideas, constructive critiques, and valuable advice. I would also like to thank Grégoire DANOY for the time he dedicated to discussing this project, for his invaluable advice, and his ever-present good humor.

I would like to warmly thank Prof. Enrique ALBA and Prof. Frédéric SAUBION for the time they spent reviewing my thesis manuscript, as well as Prof. Nicolas NAVET and Dr. Imen CHAKROUN for examining my thesis defense. I am profoundly grateful for their valuable feedback and support, which have greatly contributed to improving my work.

This thesis in cotutelle gave me the opportunity to meet and work with many people (too many to name all of them), with whom I had the pleasure of exchanging ideas, collaborating, and learning. The stimulating discussions and exchanges of ideas were essential in nurturing my reflection and deepening my research. Each of these individuals contributed, in their own way, to making this experience both enjoyable and unforgettable, and I leave with wonderful memories.

Finally, I thank my partner for accompanying me on this journey, for all the support, patience, and love she has shown me throughout these years. Her constant encouragement and presence by my side have been an invaluable source of motivation and comfort, especially during times of doubt and fatigue.

My final thoughts go to my parents, my family, and my friends.

Remerciements

Mes premiers remerciements vont tout naturellement à mes deux co-directeurs de thèse, Prof. Nouredine MELAB et Prof. Pascal BOUVRY. Je leur suis profondément reconnaissant pour le soutien indéfectible qu'ils m'ont apporté au cours de ces années. Leur expertise, leur disponibilité et leur confiance ont été des éléments déterminants dans la réalisation de cette thèse. J'ai énormément appris à leurs côtés, tant sur le plan scientifique que sur le plan humain.

Puisque cette thèse est également le fruit de nombreuses collaborations, je souhaite ici remercier mes principaux collaborateurs, Jan GMYS, Tiago CARNEIRO et Ezhilmathi KHRISNASAMY, avec qui j'ai eu l'opportunité de travailler sur ce projet. Chacun d'entre eux a contribué de manière unique à l'avancement de mes recherches, par ses idées, ses critiques constructives et ses conseils avisés. Un grand merci également à Grégoire DANOY pour le temps qu'il a consacré à échanger autour de ce projet, pour ses précieux conseils et son infaillible bonne humeur.

Je tiens à remercier chaleureusement Prof. Enrique ALBA et Prof. Frédéric SAUBION pour le temps qu'ils ont consacré à rapporter mon manuscrit de thèse, ainsi qu'à Prof. Nicolas NAVET et Dr. Imen CHAKROUN pour l'examen de ma soutenance. Je leur suis profondément reconnaissant pour leurs précieux retours et leur soutien, qui ont grandement contribué à l'amélioration de mon travail.

Cette thèse en cotutelle m'a offert l'opportunité de rencontrer et de travailler avec de nombreuses personnes (trop nombreuses pour toutes les nommer), avec qui j'ai eu le plaisir d'échanger, de collaborer et d'apprendre. Les discussions stimulantes et les échanges d'idées ont été essentiels pour nourrir ma réflexion et approfondir mes recherches. Toutes ces personnes ont contribué, chacune à leur façon, à rendre cette expérience agréable et inoubliable, et j'en repars avec de merveilleux souvenirs.

Je remercie enfin ma compagne de m'avoir accompagné dans cette aventure, pour tout le soutien, la patience et l'amour qu'elle m'a prodigués tout au long de ces années. Ses encouragements constants et sa présence à mes côtés ont été une source inestimable de motivation et de réconfort, surtout dans les moments de doute et de fatigue.

Mes dernières pensées vont à mes parents, ma famille et mes amis.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
B&B	Branch-and-Bound
BFS	Breadth-First Search
COP	Combinatorial Optimization Problem
CPU	Central Processing Unit
deque	Double-Ended Queue
DFS	Depth-First Search
FIFO	First-In First-Out
GPU	Graphics Processing Unit
HPC	High-Performance Computing
HPCS	High Productivity Computing Systems
IVM	Integer-Vector-Matrix
LICM	Loop-Invariant Code Motion
LIFO	Last-In First-Out
MIP	Mixed Integer Programming
MPI	Message Passing Interface
MW	Master-Worker
NUMA	Non Uniform Memory Access
OOP	Object-Oriented Programming
PFSP	Permutation Flowshop Scheduling Problem
PGAS	Partitioned Global Address Space
RNG	Random Number Generator
SIMD	Single Instruction Multiple Data
UMA	Uniform Memory Access

UTS	Unbalanced Tree Search benchmark
WS	Work Stealing

Contents

Chapter 1

Introduction

Contents

1.1 Motivations and objectives

Combinatorial optimization plays a crucial role in enhancing efficiency across various domains, such as logistics, telecommunications, finance, and manufacturing, where it helps optimizing processes, reducing costs, and improving overall performance. However, the increasing scale of real-world problems, driven by the exponential growth of data, has led to a significant rise in search space size and landscape complexity. This makes solving Combinatorial Optimization Problems (COP) more challenging than ever before. To tackle efficiently these problems, various scientific challenges must be addressed across multiple levels: *modeling level*, focusing on landscape analysis to better understand the structure of the problem; *algorithmic level*, optimizing the algorithmic components to improve solution quality and efficiency; and *mapping-to-hardware level*, emphasizing parallelization strategies to leverage modern computing infrastructures. This thesis focuses on the third level, specifically on the design and implementation of parallel exact optimization algorithms for modern ultra-scale supercomputers.

One of the most widely used exact methods to solve COPs is the Branch-and-Bound (B&B) algorithm. This method implicitly enumerates all possible solutions by dynamically constructing and exploring a search tree, where each node represents a simpler and more constrained version (subproblems) of the initial problem. B&B operates through four key operators: *branching*, *bounding*, *selection*, and *pruning*. Branching involves dividing a given problem into several smaller subproblems by creating new branches in the tree. Bounding calculates a bound on the cost of a subproblem, indicating whether further exploration is necessary. Selection determines which node to explore next, based

on predefined criteria, such as depth-first or best-first search strategies. Finally, pruning eliminates subproblems that cannot yield a better solution than the current best, having a worse bound than the current optimal solution. Collectively, these operators allow B&B to systematically reduce the search space while ensuring the discovery of an optimal solution. However, the pruning of branches produces highly irregular and unpredictable search trees, in terms of size and shape. Additionally, for large-scale problem instances, B&B still generates very large trees. For example, one of the largest Permutation Flowshop Scheduling Problem (PFSP) instances solved to optimality requires an equivalent computing power of 64 CPU-years, exploring a tree composed of 339×10^{12} nodes [Gmy22]. For this reason, the design and implementation of parallel B&B algorithms have been extensively studied in past decades.

Parallelizing B&B presents several challenges related to the dynamic generation of large, highly irregular trees. The most commonly used approach is to have independent B&B processes for the parallel exploration of different parts of the search space. This is generally implemented using Central Processing Unit (CPU) parallelism and involves a data structure to store generated but not yet evaluated subproblems. In such methods, the irregularity of B&B requires load-balancing mechanisms to ensure the full utilization of available cores. Hybrid CPU-Graphics Processing Unit (GPU) approaches have also been proposed, using available GPU devices for the acceleration of the bounding operator, which is often the most time-intensive part of B&B. In such configuration, managing efficiently data transfer and synchronization between CPU and GPU, especially given the irregularity and dynamic nature of B&B workloads is crucial for performance.

With the advent of the exascale *era* in June 2022 [TOP24], modern supercomputers are now composed of thousands of hybrid compute nodes, each integrating multi-core processors coupled with GPU accelerators. This hierarchical organization, providing multi-level parallelism (intra-node, GPU, inter-node or cluster, *etc.*), makes exascale parallel implementation complex. At least four roadblocks are identified: scalability, GPU-heterogeneity, portability, and fault-tolerance. Algorithms and communication patterns must be designed to scale effectively with larger numbers of processing units, ensuring that overhead does not negate the benefits of added resources. Additionally, GPU-heterogeneity and portability issues arise from the need to adapt applications to various GPU architectures, each with unique performance characteristics, while ensuring that they run efficiently across diverse hardware platforms without extensive rework. Finally, fault tolerance is another crucial issue, as the sheer scale of exascale systems increases the likelihood of hardware failures [Cap09].

Exascale computing has greatly stimulated scientific research recent years. Ambitious national and international research initiatives, such as the French Priority Research Programs and Equipment "Digital for Exascale" (NumPEx)[1], aim to promote research and development of new architectures, new software and new computing technologies to achieve exascale. This includes exploring new computational paradigms, optimizing existing algorithms, and developing new methods to solve complex problems. In the context of optimization, the "Ultra-scale Computing for solving Big Optimization Prob-

[1]The PEPR NumPEx: a French program dedicated to Exascale; see: `https://numpex.org/`.

lems" (UltraBO) bilateral France / Luxembourg international scientific project, within which this thesis is situated, aims to investigate approaches for the exascale-aware design and implementation of algorithms for solving challenging optimization problems[2]. Most existing works employ the "evolutionary" MPI+X approach, which extends the Message Passing Interface (MPI) standard used for inter-node level with environments for intra-node parallelism (OpenMP, CUDA, *etc.*). In this thesis, we investigate the Partitioned Global Address Space (PGAS) approach, an alternative to MPI+X, in the context of implementing B&B algorithms for exascale. This "revolutionary" approach provides a higher level of parallelism abstraction, unifying intra-node and inter-node levels.

1.2 Contributions

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

- The design of B&B algorithms for large-scale systems highlights the critical need for scalable data structures to manage large solution spaces. Therefore, the first contribution of the thesis consists in the **design and implementation of a PGAS-based data structure, called `distBag_DFS`, dedicated to depth-first exploration of large irregular trees**. This multi-pool data structure integrates a dynamic load-balancing mechanism based on large-scale Work Stealing (WS), operating at both intra- and inter-node levels. This mechanism, which required sophisticated synchronization, promotes locality in WS, enabling scalability.

- **We provide a `distBag_DFS`-based parallel B&B algorithm targeting CPU-based clusters**. The algorithm is developed with re-usability in mind and is thus generic with regards to the tackled optimization problem. This contrasts with the few existing PGAS-based parallel B&B algorithm that benefit from problem-specific design and optimizations, such as a problem-specific data structure [Car+20]. The algorithm is instantiated on the three following problems: the PFSP, the 0/1-Knapsack problem, and the N-Queens problem. The different characteristic features of these problems, in particular the shape of the explored tree and the computational complexity of the node evaluation function, allow us to establish the performance and limitations of the approach. **The experimental results confirmed the optimality of solutions for some of the largest PFSP instances, using up to 400 compute nodes, or 51,200 CPU cores.**

- GPU computing in the PGAS context is in its infancy, especially in the context of optimization. This third contribution aims to pave the way by extending the **design and implementation of PGAS-based parallel B&B to the multi-GPU context**, accelerating the extensive and costly evaluation of tree nodes being explored. A vendor-neutral approach is proposed and its code performance and portability is experimented on several GPU architectures. For comparison purpose,

[2]The UltraBO research project: Ultra-scale Computing for solving Big Optimization Problems; see: `https://sites.google.com/view/ultrabo`.

baseline implementations based on CUDA/HIP are also provided. **Extensive experimentation on a pre-exascale system is presented, investigating the scalability of our approach solving large problem instances using up to 1,024 GPU accelerators**.

- Implementing parallel B&B algorithms appeals for a software platform that gathers re-usable components together and sets down the foundations to ensure components interchangeability. **We come up with `pBB-chpl`, a modular Chapel platform facilitating PGAS-based parallel B&B implementation.** All the approaches investigated in this thesis are provided in `pBB-chpl` as skeletons, and the extensibility of the platform to other optimization problems is discussed. Freely available, open-source, and exemplified, `pBB-chpl` is documented through a dedicated webpage to maximize its accessibility to the scientific community.

1.3 Outline of the thesis

The remaining of the thesis is organized in five chapters.

Chapter 2 establishes all the necessary prerequisites for a comprehensive understanding of this thesis. It covers a background on the B&B algorithms and the challenges related to their parallel design and implementation. The architecture of modern supercomputers is presented, highlighting the challenges in parallelizing B&B in the context of exascale programming. This chapter also provides an extensive review of the state-of-the-art, along with a description of the benchmark problems under consideration in this thesis.

Chapter 3 provides the design and implementation of a PGAS-based parallel B&B for CPU-based clusters. More precisely, it first introduces the PGAS-based `distBag_DFS` data structure, specifically designed for the conception of unbalanced depth-first tree-based algorithms. `distBag_DFS` consists of a parallel-safe multi-pool equipped with a dynamic load balancing mechanism and an advanced synchronization scheme. Then, a `distBag_DFS`-based parallel B&B algorithm is provided. Extensive experiments showing the strong scaling efficiency of the approach solving various COPs, including at scale, are reported.

Chapter 4 extends the design and implementation of the PGAS-based parallel B&B algorithm to deal with GPU-powered heterogeneous architectures. The proposed algorithm exploits GPUs to accelerate the most compute-intensive part of the B&B, while the CPU manages the dynamic exploration of the tree. The chapter specifically describes the CPU-GPU interactions and discusses aspects related to dynamic load balancing. The experimental evaluation first demonstrates the code performance and portability of our approach on several GPU architectures. Then, the strong scaling efficiency is investigated on a pre-exascale supercomputer solving hard problem instances.

Chapter 5 presents the `pBB-chpl` software Chapel platform for the multi-level PGAS-based parallelization of B&B algorithms developed in this thesis. Details are given on the modular design of the platform, highlighting its scalable architecture and showcasing its publicly accessible documentation. Furthermore, the flexibility of the latter is discussed, demonstrating how it can be extended to tackle a wide range of other optimization problems.

In Chapter 6, the general conclusions of this thesis are drawn and several perspectives are identified. The chapter also summarizes all the scientific contributions made in the course of this PhD thesis.

Chapter 2

Background and Related Works

Contents

In this chapter, we provide a comprehensive overview of all the prerequisites essential for understanding the challenges and contributions presented in this thesis. We begin by exploring the fundamentals of solving COPs in Section 2.1, where we introduce key concepts and methodologies relevant to this domain. Following this, Section 2.2 examines the exact B&B methods, focusing on their potential for efficient parallel execution and the associated challenges.

Next, Section 2.3 delves into the architecture and complexity of modern supercomputers. This section highlights the computational power and architectural hierarchy of High-Performance Computing (HPC) platforms that can support large-scale parallel algorithms, while also examining the challenges they introduce in algorithm design and optimization.

To contextualize our work, we review significant related works in Section 2.4, emphasizing past efforts in designing and implementing parallel B&B algorithms. Finally, Section 2.5 introduces the benchmark problems and datasets used for evaluating our contributions.

2.1 Solving combinatorial optimization problems

Solving a COP consists in optimizing (minimizing or maximizing) an objective function subject to some constraints within a finite set of solutions. These problems typically involve decision variables that take on discrete values, and can be mathematically formulated as:

$$\min_{x \in X} / \max_{x \in X} f(x),$$

where:

- X is the finite (or countably infinite) set of feasible solutions;
- $f : X \to \mathbb{R}$ is the objective function that assigns a value to each solution $x \in X$, measuring its cost (*e.g.*, quality, time, benefit);
- The objective is to find $x^* \in X$ such that $f(x^*) \leq f(x), \forall x \in X$ (in the case of minimization), or $f(x^*) \geq f(x), \forall x \in X$ (in the case of maximization).

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. COPs are generally formulated as Mixed Integer Programming (MIP) problems and most of them are NP-hard [GJ79].

COPs are omnipresent in daily life, impacting everything from logistics to scheduling and resource allocation. For instance, assignment problems involve allocating resources to tasks in a way that minimizes costs or maximizes efficiency, such as matching workers to jobs or tasks to machines. Routing problems focus on finding the optimal paths through a network for agents or vehicles, with the goal of minimizing total travel distance or time. Scheduling problems address the challenge of assigning tasks or jobs to resources over time, often with complex constraints like task precedence or machine availability. Solving these problems efficiently is crucial for optimizing performance, reducing costs, and enhancing productivity across various industries, making them a cornerstone of decision-making in real-world applications.

Approaches to solving COPs can be classified into two main categories: exact and approximate methods [Tal09]. Exact methods aim to find the optimal solution(s) to a problem and prove its (their) optimality. They possess an enumerative nature and

require, in the worst case, a number of iterations which grows exponentially with the problem size (*e.g.*, the number of decision variables). Complete enumeration of feasible solutions is obviously not appropriate for large problems, which would involve a huge amount of time and computational resources. Instead, the more advanced B&B algorithm is generally used. It systematically explores and prunes branches of the solution space by calculating bounds on the objective function, discarding sub-spaces that cannot yield better solutions than the current best.

Approximate methods, on the other hand, provide a practical approach to tackling complex problems when exact solving is computationally expensive. These methods typically involve using metaheuristics to generate near-optimal solutions more quickly, exploring parts of the solution space where good quality solutions are expected to be found [Tal09]. Different types of metaheuristics exist, the single solution-based ones which iteratively improved a given initial solution (*e.g.*, Hill-Climbing, Simulated Annealing), and the population-based ones which operate on a set of solutions which are collectively or independently improved (*e.g.*, Evolutionary Algorithms, Ant Colonies). Approximate methods do not guarantee the optimality of a solution, but can help identify promising regions of the search space, guiding more refined search processes in a B&B framework.

Sophisticated hybrid exact-approximate methods for solving COPs have also been investigated [Meh11]. They combine the strengths of exact algorithms, which guarantee optimality, with approximate techniques, which provide faster convergence to high-quality solutions. For instance, these methods can apply exact B&B algorithms to refine the best solutions found through approximate methods, or integrate metaheuristics to guide B&B towards promising regions of the search space.

In this thesis, the focus is put on exact solving using B&B algorithms.

2.2 Parallel Branch-and-Bound (B&B) algorithms

2.2.1 General principles and terminology of sequential B&B

B&B is one of the most widely used exact methods to solve COPs. It implicitly enumerates all possible solutions by dynamically constructing and exploring a search tree, where each node represents a potential partial solution. More precisely, three types of nodes are distinguished in a B&B tree:

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

All generated but not yet evaluated nodes are stored in a data structure, which initially contains only the root node. The B&B method then iteratively applies four key operators—branching, bounding, selection, and pruning—to implicitly explore the entire tree, as illustrated in Figure 2.1.

- *Branching*: Branching involves dividing a given subproblem into several smaller pairwise disjoint subproblems by creating new branches in the tree, each representing a more constrained version of the original subproblem. Two different decomposition strategies are usually considered: the dichotomous decomposition which generates two subproblems per branching, and the polytomous decomposition which breaks down a subproblem into multiple subproblems in a way that each branching step assign one decision variable [Rou87]. The choice between both strategies depends on the nature of the problem being solved. Considering permutation-based problems, the polytomous strategy is more often used as at each level l of the tree we still have $n - l$ possible values to place in the permutation of size n. In contrast, the dichotomous decomposition is preferred where at each level of the tree, a single binary decision variable x_i is chosen for branching and two values are possible: $x_i = 0$ or 1. For each decomposition method, two sets of child nodes can be generated by fixing values at the first free position at the beginning and at the end, respectively.

- *Bounding*: Bounding calculates a bound on the best possible solution within a subproblem (lower bound for minimization problems and upper bound for maximization problems). In the literature, two kinds of bound evaluation are distinguished: eager and lazy evaluations [CP99]. In the first one, bounds are computed as soon as nodes are generated, *i.e.*, bounding is called after the branching operator. This contrasts with the lazy evaluation, where bounds are only computed if necessary, *i.e.*, after selection and before the branching operator. In this thesis, the eager evaluation mode is used.

- *Selection*: Selection determines which node to explore next based on predefined criteria, such as Depth-First Search (DFS) or Breadth-First Search (BFS) strategies. DFS explores a branch as far down as possible before backtracking (Last-In First-Out (LIFO)), while BFS explores all neighboring nodes at the current depth before moving deeper (First-In First-Out (FIFO)). DFS uses a stack to keep track of the nodes that need to be explored and lead to memory usage that is largely determined by the depth of the structure being traversed. In contrast, BFS employs a queue to manage the nodes at the current level before moving on to the next. This means that BFS needs to store all of the nodes at a given depth simultaneously, which can require much more memory, especially in wide trees where many nodes are present at the same level.

- *Pruning*: Pruning eliminates subproblems that cannot yield an optimal solution, due to having a worse bound than the cost of the current best solution. Actually, the algorithm keeps track of the best solution found so far (referred to as the *incumbent*) and its corresponding cost (known as the *upper bound* for minimization problems or the *lower bound* for maximization problems). This cost is initially set to either $\pm\infty$ or the cost of any feasible solution, if one is known in advance (for example, obtained through an approximate method). If the algorithm is initialized with an optimal solution, it will explore exactly the nodes in the solution space

for which the bound is better than the cost of the optimal solution. Regardless of the search strategy, these nodes must be explored to prove the optimality of the solution. The tree formed by these essential nodes is referred to as the *critical tree*.

At the end of a B&B algorithm, the final incumbent solution, when no more subproblems remain to explore, is guaranteed to be optimal, as it represents the best possible solution within the entire solution space. In practice, the size of the explored B&B tree to achieve optimality depends on several factors, such as the quality of the bounding operator and the search strategy defined by the selection operator.

The simple B&B method led to several, more sophisticated variants of which the most significant are Branch-and-Price (B&P) and Branch-and-Cut (B&C) [Bar+98; Mit11]. B&P integrates B&B with column generation to solve large-scale integer programming problems, while B&C incorporates cutting planes to eliminate fractional solutions and improve the search for optimal integer solutions. Many other B&B-like approaches exist, such as branch-and-peg, branch-and-win, branch-and-cut-and-solve [GGS03; CZ06; PC04]. Some authors consider B&P, B&C, and the other variants as different algorithms than B&B and use B&X to refer to these algorithms. In this thesis, B&B refers to simple B&B or any of its variants. Generally speaking, divide-and-conquer algorithms can also be considered as B&B algorithms, as it is enough to remove the pruning operator from B&B. Additionally, the fundamental backtracking paradigm used to solve constraint satisfaction problems can also be seen as a particular case of a DFS B&B algorithm. The difference is that backtracking does not use a bounding operator to detect unpromising nodes, but it may incorporate pruning mechanisms, for instance based on evaluating the feasibility of a subproblem, which can be interpreted as a binary bounding function. In that perspective, the bounding operator is frequently called *node evaluation function* in this thesis.

In comparison to complete enumeration, pruning branches considerably reduces the size of the explored tree. However, for many COPs, the execution time of B&B increases significantly with the input size, making it practical to solve only instances of small or moderate size using sequential algorithms. As a result, leveraging parallel computers becomes an appealing approach for handling larger instances of COPs.

2.2.2 Models for parallel B&B

The parallelization of B&B has been particularly studied and four parallel methods can be outlined from the literature [GC94; Mel05]:

- *Parallel tree exploration model*: The parallel tree exploration model is undoubtedly the model that has been studied the most, since it has a high degree of parallelism on big problem instances. It consists in exploring several disjoint search subspaces (branches) in parallel, as shown in Figure 2.2a. More precisely, it allows all four B&B operators—branching, bounding, selection, and pruning—to be executed in parallel across different subproblems. In addition, as this model does not affect the bounding operator, it is independent from the problem to be solved. This model can be implemented either in a synchronous or asynchronous manner. In the first

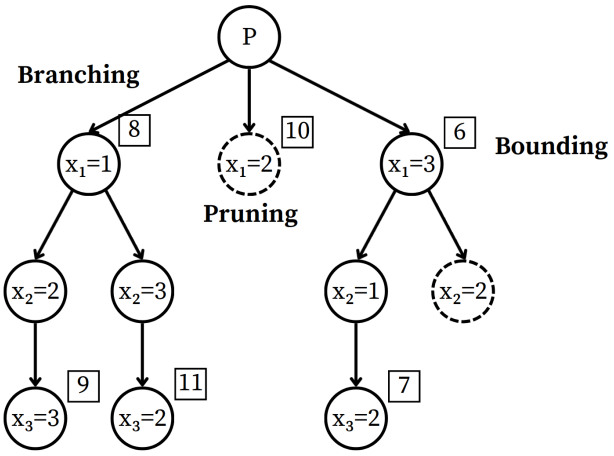


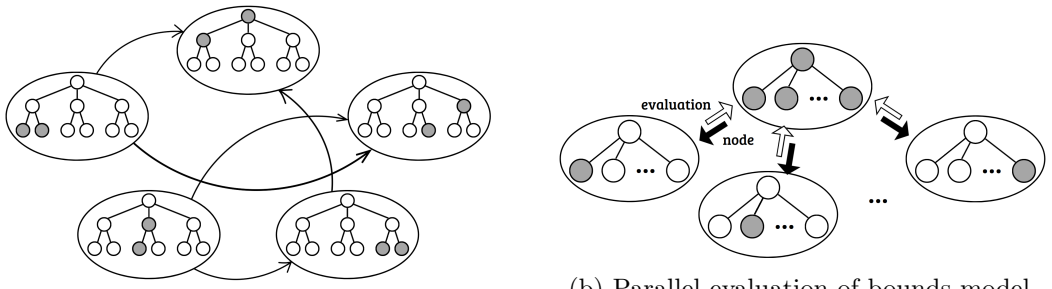
Figure 2.1: Illustration of a Branch-and-Bound algorithm.

one, the B&B algorithm consists of several phases where the B&B processes conduct their exploration independently. Between these phases, the exploration processes synchronize and can share information, including the best solution identified up to that point. In contrast, the asynchronous mode leads to communications between B&B processes in an unpredictable manner.

- *Parallel evaluation of bounds model*: This model consists in bounding several subproblems concurrently in parallel, as illustrated in Figure 2.2b. 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 pending subproblems 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 also has the advantage of being generic and can be nested inside the parallel tree exploration model. It is however more suited for problems with a costly bounding function rather than fine-grained ones, which often penalize it in large-scale environments.

- *Parallel evaluation of a bound model*: This model consists in parallelizing the bounding function itself. As a low-level model modifying only the bounding operator, it can be nested inside both previous models to add one more level of parallelism, or alone for problems having a very compute intensive bounding function. This type of parallelism has no influence on the general structure of the B&B and is specific to the problem to be solved.

- *Parallel multi-parametric model*: This model is certainly the less studied one and consists in launching in parallel several independent B&B processes that explore the same search space but with different algorithm parameters (*e.g.*, different selection and/or branching operators). For instance, in [KK84] only one of the B&B



(a) Parallel tree exploration model.

(b) Parallel evaluation of bounds model.

Figure 2.2: Illustration of the parallelization models used in this thesis.

algorithms uses the actual current upper bound while the others use a hypothetical better solution as upper bound. The main advantage of this model is its genericity, but it has the disadvantage of generating additional exploration costs as many nodes are explored redundantly.

2.2.3 Challenges in parallel B&B

We identified five main challenges that arise when implementing parallel B&B algorithms:

- *Irregularity*: The challenges encountered when implementing a parallel B&B algorithm largely stem from the inherent irregularity of the algorithm. In each of the four parallelization models discussed, this irregularity manifests differently. Due to the unpredictable pruning of branches, some subproblems demand significantly more computational effort than others, creating load imbalance. As subproblems are dynamically assigned to processing units at runtime, efficient dynamic load balancing becomes crucial for optimizing resource utilization.

  While parallel evaluation of tree nodes typically results in a more fine-grained and often more regular workload, the time required for node evaluation can vary depending on the specific problem and the depth of the tree node. Additionally, both the search space management and the problem-specific node evaluation functions are often characterized by highly irregular control flows. These irregularities, such as diverging instruction flows and unpredictable memory access patterns, pose significant challenges for Single Instruction Multiple Data (SIMD) processing. This can hinder the efficient use of GPUs, which are key drivers of HPC systems' recent advancements.

- *Work pool management*: Most of B&B algorithms employ a *work pool* to store subproblems that have been generated but not yet evaluated. This data structure allows the insertion/retrieval of subproblems following a certain order, facilitating

the implementation of a search strategy. For instance, DFS corresponds to processing nodes in LIFO order and is therefore naturally implemented by a stack, while BFS corresponds to processing nodes in FIFO order and is rather implemented by a queue.

In parallel B&B, multiple strategies for implementing the work pool exist, such as single-pool and multi-pool approaches. In single-pool approaches, a centralized work pool is concurrently accessed by all B&B processes to pick subproblems for branching/evaluation. In large-scale scenarios, where synchronization between processes is needed to ensure parallel-safety, bottleneck and memory contention are likely to occur, limiting the scalability of the approach. The multi-pool approaches address this bottleneck by maintaining multiple pools. In the collegial multi-pool model, for instance, each B&B process keeps its own private pool [GC94]. While this approach reduces the bottlenecks found in single-pool models, it introduces the challenge of balancing the workload across multiple pools. Additionally, sharing information among processes—such as the best-known solution and detecting termination—becomes more complex.

- *Load balancing*: As discussed previously, load balancing is a significant challenge in parallel B&B implementations that use a multi-pool strategy. In the literature, the most common load balancing technique is the WS paradigm, which distributes tasks among threads [BL99]. In WS, each thread maintains a Double-Ended Queue (deque) to store tasks. Locally, a thread treats its deque as a stack, popping tasks from the tail to execute and pushing new tasks onto it. Then, when a thread's deque becomes empty due to load imbalance, it acts as a "thief" and steals tasks from the head of another "victim" thread's deque. Deques are primarily used in WS approaches for two main reasons. First, stealing tasks from the head of the deque allows the victim thread to continue working at the tail without interruption from steal operations [ABP98; Din+09]. However, because concurrent operations on deques require expensive memory fences, there is growing interest in WS implementations that rely on non-concurrent (private) data structures [ACR13; DP14]. The second reason relates to the granularity of the WS mechanism. In DFS B&B, as in many task-parallel applications, tree nodes lower in the task stack typically represent a larger workload compared to more recent tasks at the top. Granularity—defined as the number of tasks stolen—is a key characteristic of a WS strategy, along with the policy for selecting a victim for stealing work.

- *Data structure*: Both work pool management and load balancing challenges highlight the central role of the data structure used to store the huge number of pending subproblems during execution of a B&B algorithm. Usually, 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, such as stacks or queues. The main advantage of these data structures is their genericity with regards to the problem solved; it is relatively straightforward to adapt B&B to different problems by changing the definition of

a node. In the literature, other types of data structures have been studied, offering higher performance at the expense of this genericity. This is achieved by taking full advantage of the characteristics of the problem being tackled, or more generally the class of problems to which it belongs.

For instance, some data structures have been proposed to exploit the structure of the search space for permutation problems. A first example is bitsets data structure, which allows a very compact implementation of DFS [SRR08; Ric97]. Using such approach, a B&B algorithm can be implemented using only one vector and two integers for the search procedure. The 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. Another example is the Integer-Vector-Matrix (IVM) data structure, which offers a more flexible but less compact alternative to bitsets [Mez+14; Ler15]. It 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.

- *Initial generation and allocation of work units*: The last challenge arises at the beginning of the execution of a parallel B&B algorithm, when only one subproblem, the root node of the tree, is available to all process. In that case, a start-up phase where parallelism is not fully utilized seems difficult to avoid. Additionally, in some scenarios, using parallelism as soon as several work units become available may lead to performance anomalies [GC94]. Several strategies have been proposed to address this issue: (1) assign the original problem to one process, and gradually broadcast among processes the work units as they are created; (2) one process performs a sequential B&B algorithm up to a point where a "sufficient" number of pending subproblems are available; (3) all processes perform a sequential phase in which the same tree is built by every process and, when the number of pending subproblems becomes at least equal to the number of processes, each process selects the subproblems on which it will subsequently work; *etc.* While the third approach is suited mainly for multi-pool algorithms, the first two can be used in all types of algorithms. In practice, the choice of an appropriate strategy depends on the characteristics of the problem to solve and the nature of the parallel architecture being used.

2.3 Architecture and complexity of modern supercomputers

HPC technologies are evolving rapidly, and the architectures of modern computing systems are becoming increasingly complex. To fully harness the computational power of these systems, programmers must have a solid understanding of their hierarchical organization. While a detailed technical analysis of the hardware used in this thesis, or an in-depth discussion of the latest trends and future developments in HPC, is beyond the scope of this chapter, it is important to note that the design of the algorithms pre-

sented in this thesis is driven by the recent evolution of the organization of modern supercomputers. This section aims to provide some context by briefly outlining current trends and challenges in HPC, along with a concise description of the hardware used in this work. For those interested in a more thorough exploration of the rapid evolution of computing systems in recent years, numerous studies are available, such as [GR14; Kec+11; Par+19], among others.

2.3.1 Modern supercomputers: a glimpse into the TOP500

This section discusses the architecture of modern supercomputers through the prism of the TOP500 list. The latter provides a revealing snapshot of the most advanced architectures shaping the landscape of HPC for over 30 years [TOP24]. Since 1993, it compiles twice a year a list of the 500 world's most powerful computer systems, providing statistics on their performance, architecture, diversification, *etc.* Supercomputers are ranked by their $R_{\max}$ performance, which is the maximal LINPACK performance achieved solving dense systems of linear equations.

The systems featured in the TOP500 list derive much of their computational power from advances in CPU design, from single-core processors to today's multi-core architectures. Historically, CPUs were simple, single-core processors and performance gains were achieved by increasing the clock frequency, allowing faster sequential execution of instructions. However, increasing frequencies led to physical constraints, such as power consumption and heat dissipation. These limitations led to a paradigm shift in computer architecture with the introduction in the 2000s of multi-core processors running at slightly lower frequencies, where multiple processing units (cores) were integrated on a single die, enabling parallel execution of multiple instruction streams. Advances in semiconductor technology led to the miniaturization of transistors (up to 3 nanometers in 2022) and a steady increase in transistor count, encapsulated by Moore's Law, which predicted a doubling of transistor density roughly every two years. In the latest edition of TOP500, no system has less than 4 CPU cores per socket and almost half of the systems have at least 24 cores per socket or more [TOP24]. In addition, the Sunway SW26010 processor is composed of 260 cores clocked each at a base frequency of 1.45 GHz.

The rise of GPU accelerators has also significantly enhanced the computational capabilities of modern supercomputers, complementing traditional CPU-based architectures. Originally developed for rendering graphics, GPUs excel in massively parallel processing, making them ideal for handling data-intensive tasks in scientific computing, machine learning, and large-scale simulations. Unlike CPUs, which prioritize sequential task execution with fewer, more complex cores, GPUs feature thousands of simpler cores that can process multiple tasks simultaneously. This ability to perform a high number of parallel operations has driven their integration into HPC systems. The shift toward heterogeneous architectures, combining CPUs with GPUs, has become a defining trend in the TOP500 list, where GPU-powered systems now dominate the highest ranks. Notably, the Frontier supercomputer, which holds the top position for several releases, combines over 37,500 AMD MI250X GPU accelerators alongside its 64-core AMD CPUs to achieve

unprecedented levels of performance.

2.3.1.1 Parallel Architectures

Understanding the parallel architecture of computing systems is crucial for optimizing performance in HPC environments. Efficient parallelism at different levels enables the effective utilization of multiple processing units, including CPUs and GPUs. In this context, *intra-node parallelism* is distinguished from *inter-node parallelism*. Intra-node parallelism refers to the parallel execution of multiple threads simultaneously within a single compute node, leveraging the cores of CPUs and the parallel processing capabilities of GPUs. In contrast, inter-node parallelism operates across multiple compute nodes within a distributed computing environment, requiring effective communication and coordination among the nodes to share data and manage workloads efficiently.

Intra-node parallelism exploits the multiple cores available in a CPU and/or the thousands of cores present in a GPU to execute operations concurrently. However, CPUs and GPUs are fundamentally different in their architectures and operational designs, as shown in Figure 2.3. A CPU features one or several processing core(s), each having three functional units: Arithmetic Logic Unit (ALU), control unit, and memory unit. The ALU performs all the arithmetic and logical operations on the data as specified in the program code. The control unit first decodes the program instructions and directs the ALU to operate on the data. Finally, the memory unit consists of high-speed memory that is used by the core to store the data during the program execution. The memory architecture associated with a CPU plays a crucial role in determining how efficiently data can be accessed and processed. At the top of this hierarchy is cache memory, which consists of three levels: L1, L2, and L3. The L1 cache is the smallest and fastest, located directly on the CPU chip and dedicated to each core, storing the most frequently accessed data. The L2 cache, larger but slightly slower, serves as a secondary storage area for data that may not currently reside in L1. L3 cache, even larger and shared among all cores, acts as a buffer for data not found in L1 or L2, further reducing access times compared to fetching from the main memory[3]. The main memory of a computer, primarily composed of Dynamic Random Access Memory (DRAM), is slower than cache memory but offers significantly greater capacity, allowing it to hold the active data and instructions required by running applications. 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.

Inter-node parallelism, on the other hand, involves coordinating threads across multiple compute nodes within a distributed system or cluster. Each compute node operates as an independent unit with its own CPUs, GPUs, and memory, requiring communication between compute nodes to share workloads. Typically, a message-passing library is used for data exchange, facilitating communication over high-speed networks like Infini-

[3]For a sake of simplicity, L2 and L3 caches have been unified in Figure 2.3

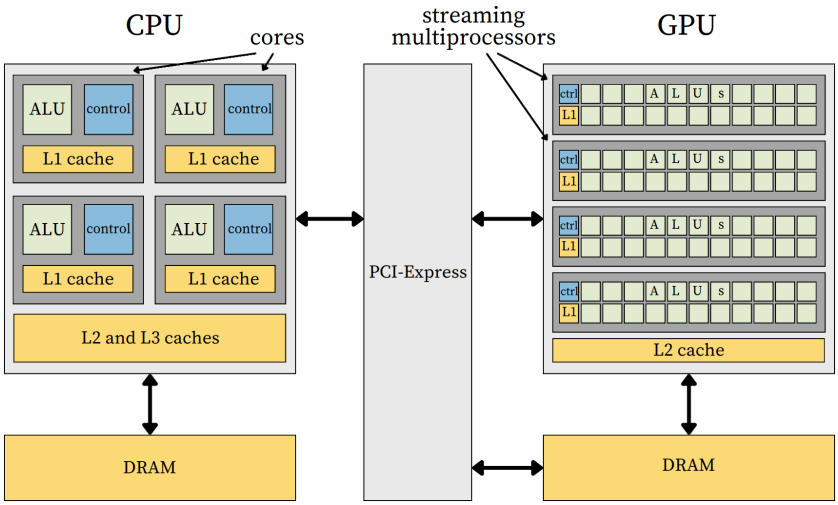


Figure 2.3: Architecture of CPU and GPU processors.

Band or Ethernet. By utilizing inter-node parallelism, large-scale computations can be efficiently carried out across clusters, making it a crucial element for high-performance computing systems.

2.3.1.2 Exascale computing

The June 2022 edition of the TOP500 marked the beginning of the exascale *era*, 14 years after the entry into the petascale one. With a LINPACK score of 1.102 Exaflop/s, the HPE Cray EX Frontier system not only becomes the most powerful supercomputer to ever exist, but also the first true exascale machine. It consists of 9,408 compute nodes, each equipped with one AMD Milan "Trento" 7A53 Epyc CPU and four AMD Instinct MI250X GPUs, for a total of 8,730,112 cores. Although Frontier remains the world's most powerful computer to date, the HPE Cray EX Aurora has became in June 2024 the second system to break the exascale barrier with an LINPACK score of 1.012 EFlop/s, featuring a total of 9,264,128 cores.

In general, the current trend to achieve exascale systems involves increasing the number of processing cores [TOP24]. As shown in Figure 2.4, there is a correlation between the combined number of CPU and GPU cores in June 2024 TOP500 machines and their ranking. Most systems in the Top 100 feature over 100,000 compute cores, reaching more than a million for the Top 10. Additionally, there is a generally linear trend over the years in the performance growth of TOP500 systems. As indicated in Figure 2.5, the R_{max} power has increased linearly since 1993, allowing researchers to predict the entry into the exascale era in the early 2020s. Finally, while overall performance grows due to the increasing number of resources, a significant diversification of computing architectures can also be seen. One of the most striking examples is the Top 3 of the June 2024 TOP500, shown in Figure 2.6, showcasing three systems (two of which are exascale) each equipped with a GPU architecture from a different vendor: AMD, Intel,

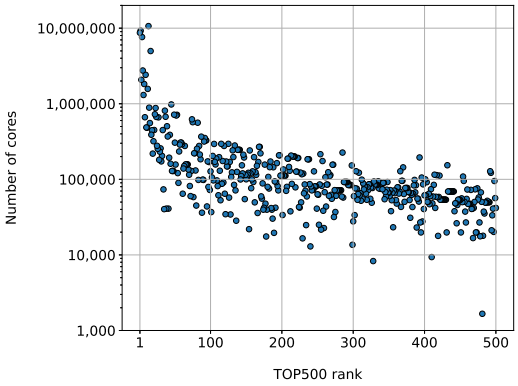


Figure 2.4: Number of combined CPU and GPU cores by TOP500 rank (June 2024).

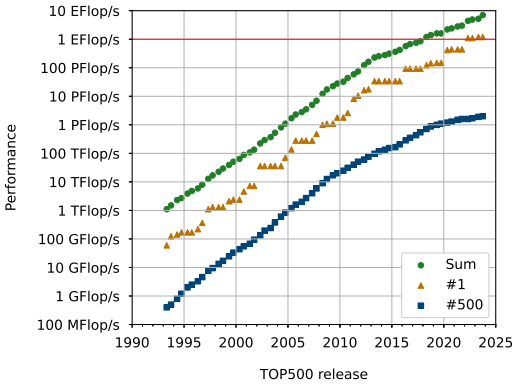


Figure 2.5: $R_{\max}$ performance of TOP500 supercomputers from 1993 to 2024.

and NVIDIA.

At a quintillion (10^{18}) calculations each second, exascale supercomputers will have a profound impact on everyday life in the coming decades, quickly analyzing massive volumes of data and more realistically simulating the complex processes and relationships behind many of the fundamental forces of the universe. This will have practical applications in everything from precision medicine to regional climate, water use to materials science, nuclear physics to national security. Given this context, several ambitious national and international research initiatives have emerged. For instance, the French Priority Research Programs and Equipment "Digital for Exascale" (NumPEx) aims to promote research and development of new architectures, new software and new computing technologies to achieve exascale. This includes exploring new computational paradigms, optimizing existing algorithms, and developing new methods to solve complex problems. Similarly, the "Ultra-scale Computing for solving Big Optimization Problems" (UltraBO) bilateral France / Luxembourg international scientific project, within which this thesis is situated, aims to investigate exascale-aware design and implementation of algorithms for solving challenging optimization problems. Indeed, several key challenges remain to be addressed to fully leverage the potential of exascale computing, particularly in the optimization context.

2.3.2 Key challenges in ultra-scale optimization

Ultra-scale computing is essential for effectively solving the large number of subproblems generated by the decomposition of large optimization problems. Additionally, the inherent parallelism in these problems makes them well-suited for ultra-scale supercomputers. However, leveraging these systems efficiently presents a significant challenge: managing an massive amount of irregular tasks across supercomputers that feature multiple layers of parallelism and a heterogeneous mix of computing resources, including GPUs, multi-core CPUs with varying architectures, and complex network topologies. This section

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	

Figure 2.6: Top 3 of TOP500 ranking (June 2024).

explores the key challenges in ultra-scale optimization that are scalability, heterogeneity and portability, fault-tolerance, and productivity.

- *Scalability*: The scalability issue requires, on the one hand, the definition of scalable data structures for efficient storage and management of the tremendous amount of subproblems generated by decomposition [Sha11]. On the other hand, dynamic load balancing mechanism requiring adaptive strategies to distribute computational tasks across resources effectively is another critical issue. This requires the optimization of communications (in number of messages, their size and scope) especially at the inter-node level. Finally, efficient mechanisms are also needed for granularity management and coding of the work units stored and communicated during the resolution process.

- *Heterogeneity and portability*: Heterogeneity means harnessing various resources including multi-core processors within different architectures and GPU devices. The challenge is therefore to design and implement hybrid optimization algorithms taking into account the difference in computational power between the various resources as well as the resource-specific issues. Hardware resource specific-level optimization mechanisms are required to deal with related issues such as thread divergence and memory optimization on GPU, data sharing and synchronization, cache locality, and vectorization on multi-core processors, *etc.* Portability, on the other hand, ensures that optimization models and tools can run seamlessly across different operating systems, hardware configurations, and software environments. This is particularly challenging in ultra-scale systems, where maintaining compatibility with diverse system setups while achieving optimal performance requires careful design and the use of cross-platform tools that can seamlessly scale without sacrificing performance or functionality across different architectures.

- *Fault tolerance*: Although not investigated in this thesis, fault tolerance represents another major challenge for ultra-scale computing [Cap09; Sni+14]. Failures can arise from various sources, including hardware failures due to component degra-

dation, software bugs, network issues like congestion or disconnections, *etc.* It is also established that as supercomputers increase in size to millions of processing cores, their Mean-Time Between Failures tends to become shorter [Sha+19]. In the optimization context, failures lead to the loss of work unit(s) being processed by some thread(s) during the resolution process. Therefore, a major issue, which is particularly critical in exact optimization, is how to recover the failed work units to ensure a reliable execution.

- *Productivity*: Productivity is an emerging measure of merit for HPC [SD08]. It stems from the increasing complexity of modern supercomputer architectures, which require more advanced software and programming models to fully exploit their capabilities, as described in previous points. Traditional parallel models like MPI+X impose a steep learning curve and necessitate considerable manual effort to optimize for performance, especially when dealing with heterogeneous components such as CPUs and GPUs. More expressive programming models are needed to deal with this issue and simplify the developer's efforts while supporting dynamic parallelism. The scientific computing challenge is retaining expressivity and productivity while also delivering high performance.

2.3.3 Enhancing HPC productivity with PGAS

This section introduces key parallel programming models, highlighting the PGAS model as an alternative to the traditional MPI+X approach. It then explores the concept of software productivity in HPC and examines the role of High Productivity Computing Systems (HPCS) languages in tackling this challenge.

2.3.3.1 PGAS as alternative to MPI+X

Parallel programming models can be categorized into three primary groups: shared-memory models, where multiple processes can read from and write to a common shared memory; message-passing models, where isolated processes with separate memories communicate by exchanging messages; and PGAS models, where a global address space is logically partitioned across multiple processes, enabling efficient access to both local and remote memory locations, combining features of both shared and distributed memory systems [DMN12]. These models are illustrated in Figure 2.7 and are discussed in more detail below.

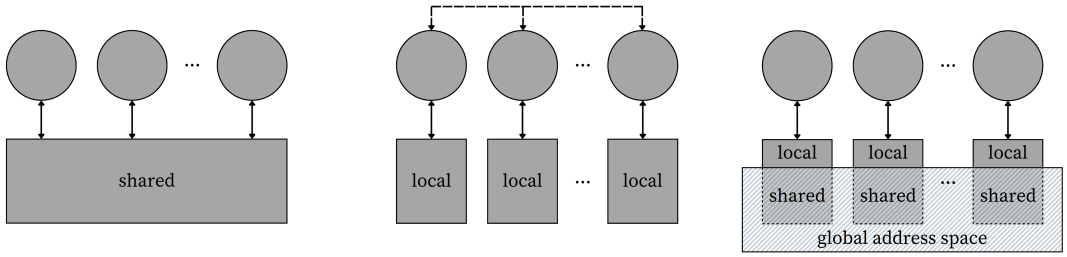
- *Shared-memory model*: This model allows multiple threads to access a common address space, facilitating communication and data sharing. All threads can read from and write to the same physical memory, enabling efficient concurrent execution as multiple threads operate simultaneously on shared data. However, to maintain data consistency and prevent conflicts, synchronization mechanisms like mutexes, semaphores, and condition variables, are employed to manage access to shared resources and prevent data races. While the shared-memory model is prevalent in multi-core architectures, due to its straightforward approach to data access

and communication, its limitations become more pronounced as systems scale, due to increased contention for shared resources resulting in bottlenecks as the number of threads increases.

- *Message-passing model*: In contrast to the previous model, in this one threads communicate and synchronize by sending and receiving messages, rather than sharing a common address space. Each thread has its own address space, and communication occurs through explicit send and receive operations, which can be synchronous or asynchronous. This model is particularly well-suited for distributed systems, where processes may run on separate machines with their own local memory. One of the key advantages of this model is that it naturally supports scalability, as processes can be distributed across multiple nodes without concerns about memory contention. However, the message-passing model also introduces complexity in managing communication, as developers must carefully handle message formatting, delivery, and potential delays, which can affect overall performance.

- *Partitioned Global Address Space model*: The PGAS model aims to merge the strengths of both shared and distributed memory models. It provides a unified global address space that is logically shared among all processes, even though the physical memory is distributed across multiple compute nodes. More precisely, PGAS implementations typically make the distinction between local, shared local and shared remote memory references. Processes are allowed to access both local and remote data using familiar pointer-based syntax, offering control over data locality. A key benefit of PGAS is its improved performance in applications with irregular memory access patterns, minimizing the overhead of traditional message-passing. However, challenges remain in maintaining data consistency and managing communication efficiently, particularly in large-scale systems. Indeed, even though processes can access remote memory, these accesses involve higher latency and additional overhead compared to local memory accesses.

The combination of shared-memory and message-passing programming models offers the potential to leverage the strengths of both: the efficiency, memory savings, and ease of programming of shared-memory models, alongside the scalability of message-passing models. In fact, this hybrid approach is the core objective of the PGAS model. In practice, implementing such a hybrid parallel model is often done in an "evolutionary" manner, by combining existing programming models and tools from each paradigm. Typically, message passing (usually with MPI) is used for communication across distributed nodes, while shared-memory (via OpenMP or Pthreads) is employed within each node. Furthermore, GPUs can be integrated to introduce a third level of parallelism, along with specialized GPU programming models such as CUDA. The combination of message-passing and shared-memory parallelism is often referred to as MPI+X. Alternatively, PGAS is implemented through parallel programming languages and libraries like Unified Parallel C (UPC), Coarray Fortran (CAF), and Titanium, which extend C, Fortran, and Java, respectively; or newer languages such as Chapel, X10, and



(a) Shared-memory. (b) Message-passing. (c) PGAS.

Figure 2.7: Illustration of main parallel programming models. Circles and rectangles represent processes and address spaces, respectively. Plain arrows and dotted arrows represent data accesses and network communications, respectively.

Fortress [Alm11]. Some of these languages, like Chapel, integrate all three levels of parallelism (intra-node, inter-node, and GPU), providing a "revolutionary" alternative to traditional MPI+X approaches and offering significant advantages in terms of software productivity. This motivated us to use the Chapel language for the implementation of our PGAS-based algorithms.

2.3.3.2 Software productivity in HPC

As modern systems tend to become larger, including numerous levels of memory, various forms of parallelism, and an increasing diversity of architectures and configurations, the notion of productivity in the context of HPC is emerging more prominently [SD08]. Some authors proposed formal frameworks to quantitatively measure HPC productivity [SB04; KKS04]. For instance, assuming that T is the time to solution, S the system used, P the problem solved, the productivity Ψ is defined by [SB04] as an utility U over a total cost:

$$\Psi(P, S, T, U) = \frac{U(P, T)}{C_D(P, S, T_D) + C_E(P, S, T_E)},$$

where C_D and C_E are the development and execution costs, respectively, and T_D and T_E their associated times. The utility includes the operations/time peak that can be achieved on the system, the efficiency achieved by the parallel program, the availability of the system, *etc.* The development cost includes code development, testing, and documentation, but also the cost of porting to new systems, the cost of recovering from the failures, the cost of training programmers when a new language or computer system is used, *etc.* In contrast, the execution cost includes the cost of the system (hardware and software), the cost of the maintenance and repair of the system, the cost of electrical power and other supplies, *etc.*

In practice, most of these measurements are hard to evaluate as they are subjective and often maintained transparently to the HPC users. It is possible to make numerous assumptions to reduce the definition of productivity to something simpler to measure, but this raises the question of its relevance. In this thesis, we address software pro-

ductivity via the use of languages implementing the PGAS programming model, which offers the functionalities described in the previous section. More particularly, we put the focus on HPCS languages.

2.3.3.3 HPCS languages

In 2002, the Defense Advanced Research Projects Agency (DARPA) initiated the HPCS project, with the goal of accelerating both the performance of the largest parallel computers and their usability. It was recognized that a significant barrier to the application of computing to science, engineering, and large-scale processing of data was the cumbersomeness of developing software that exploits the power of new architectures. As part of the HPCS project, three computer vendors, Cray, IBM, and Sun, have competed not only in the area of hardware design to address DARPA's performance goals, but also in language design to address the software development productivity goals. Each of these languages—Chapel, X10, and Fortress—is based on the PGAS model [LY07], and is described below.

- *Chapel*: Open-source programming language developed by HPE (previously Cray Inc.) for productive parallel computing on large-scale systems, including multi-core desktops, clusters, clouds, and supercomputers [CCZ04]. It supports multi-threaded execution through high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel's *locale* type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality and affinity. Chapel supports global-view data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. In contrast to many previous higher-level parallel languages, Chapel is designed around a multi-resolution philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require. Chapel supports code reuse and rapid prototyping *via* object-oriented design, type inference, and features for generic programming. Existing code can be integrated into Chapel programs (and *vice-versa*) via interoperability features.

- *X10*: Java-derived, type-safe, parallel object-oriented language developed by IBM aiming to provide a programming model that can address the architectural challenge of multiples cores, hardware accelerators, clusters, and supercomputers in a manner that provides scalable performance in a productive manner [Cha+05]. X10 introduces a flexible treatment of concurrency, distribution, and locality, within an integrated type system. Locality is managed explicitly using places, computational units with local shared memory. A program runs over a set of places. Each place can host data or run activities, which are lightweight threads that can run on its place, or (explicitly or implicitly) asynchronously update memory, in other places. For synchronization, X10 uses "clocks", which are a generalization of barriers.

- *Fortress*: General-purpose, statically typed, component-based programming language developed by Oracle Lab (previously Sun Microsystems Inc.) designed for producing robust high-performance software with high programmability [Ste+11]. Fortress supports features such as transactions, specification of locality, and implicit parallel computation, as integral features built into the core of the language. It has a novel type system to integrate functional and object-oriented programming better. Thus, it supports mathematical notation, such as $\sum$, and static checking of properties, such as physical units and dimensions, static type checking of multidimensional arrays and matrices, and definitions of domain-specific language syntax in libraries.

Fortress has been officially discontinued in 2012, citing severe technical challenges with using Fortress's type system on existing virtual machines. In addition, at the time of writing this thesis, the development of the X10 language seems to have been at a standstill for several years. However, Chapel stands out as one of the most promising PGAS-based high-performance and high-productivity parallel programming language, benefiting from an active open-source code development and lots of community-oriented resources[4]. It is therefore considered for the implementation of the contributions of this thesis.

2.4 Related works

The range of applications for B&B software is broad, which naturally leads to a wide variety of problem types, user interfaces, parallelization methods, and machine configurations. As a result, the literature has seen the development of numerous specialized B&B algorithms tailored to specific problem classes or architectures. In the following, we review the most significant of these, including a particular emphasis on PGAS-based approaches.

2.4.1 Frameworks for parallel B&B

Many frameworks for parallel B&B algorithms have been proposed over the years, such as ALPS [Xu+05], Bob++ [Dje+06], MALLBA [Alb+02], PEBBL [EHP15], PICO [EPH01], Symphony [RG05], and YewPar [Arc+19], to name a few. These frameworks aim to facilitate interaction between the user and the parallel machine by defining abstract types for search tree nodes and solutions. Users are responsible for providing concrete implementations of these types, along with the specific branching and bounding procedures, while the framework manages the broader aspects of parallel B&B execution. Frameworks differ in the types of B&B variants they support, the parallel models they implement, and the programming environments they are designed for. Often structured as multi-layered class libraries, they allow for the integration of additional features by building

[4] *e.g.*, annual forum for users and developers, blog articles, and social networks.

upon foundational layers. For example, PEBBL originated as the core component of the parallel MIP solver PICO, and has since been extended to offer more functionalities.

Symphony employs the Master-Worker (MW) paradigm for parallelizing the application, with centralized node management [RG05]. A single process, the master, oversees all aspects of the search, while the worker processes carry out the tasks assigned by the master, such as exploring one or more node(s). However, since the unit of work is the subtree rather than the subproblem, each worker is responsible for searching its assigned subtree. As a result, while the overall search is coordinated by the master, the search control is effectively distributed among the workers. This strategy performs well with a small number of processors, but it does not scale effectively. As the number of processors increases, the central pool becomes a computational and communication bottleneck. Alternatively, several frameworks adopt the Master-Hub-Worker paradigm to address the limitations of the MW approach [Xu+05; EPH01; EHP15; Dje+06]. In this model, a middle management layer is introduced between the master and worker processes. A "cluster" consists of a hub, which manages a fixed number of workers. As the number of processes grows, additional hubs and worker clusters are added. This decentralized approach preserves many of the benefits of global decision-making while reducing overhead and shifting some of the computational load from the master process to the hubs. Some libraries propose one or several distributed strategies, for instance based on MPI [Dje+06; Xu+05; Alb+02; EPH01] or the PGAS-based HPX [Arc+19].

Generally speaking, it is challenging for a framework to deliver optimal performance compared to a custom-developed algorithm, as the specific characteristics of a problem may not be accounted for by the framework. Therefore, problem-specific or architecture-specific B&B algorithms are often proposed in the literature.

2.4.2 B&B for GPU

Jenkins *et al.* [Jen+11] offer a thorough overview of the challenges in implementing parallel backtracking on GPUs, with many of their findings on GPU-based backtracking remaining applicable to B&B algorithms that employ a DFS strategy. To leverage the GPU's massive parallel processing capabilities, a fine-grained parallelization of search space exploration and/or node evaluation is essential, depending heavily on the problem's nature and the chosen parallelization model. Additional key factors include latency hiding through coalescence, saturation, and shared memory utilization. In general, the algorithmic properties of B&B, including the irregularity of the search space, control flow, and memory access patterns, conflict with the GPU programming model. Moreover, the memory requirements for backtracking and B&B algorithms are often challenging to estimate and can exceed the memory capacity of GPUs. Various approaches for GPU-accelerated B&B algorithms have been proposed, each corresponding to different parallelization models, with their design typically driven by the specific characteristics of the problem being addressed. Typically, one can cluster the approaches according to the granularity—fine, medium, and coarse—of the bounding function.

For fine-grained problems, the parallel tree exploration model is commonly implemented on the GPU [Car+11; Fei+10; Li+15; RS10; ZSW11]. In works such as [Fei+10;

Li+15; ZSW11], the node evaluation function for the N-Queens problem is executed on the GPU, utilizing minimal registers or memory and performing a few bit-operations. A typical approach in these algorithms involves first conducting a sequential (or weakly parallel) search on the CPU, followed by a parallel search on the GPU. The primary goal of the CPU search is to generate a sufficiently large active set of subproblems. Each subproblem is then assigned to a GPU thread and serves as the root for an independent search. The main challenge in this fine-grained approach lies in the careful tuning of parameters such as the cutoff depth or the active set size, as these factors significantly affect thread granularity. Although varying thread granularities can lead to load imbalance, none of these works implement dynamic load balancing on the GPU. They assume that the static work distribution, established after the CPU search, is sufficiently regular to avoid significant issues.

Medium-grained problems are defined in [Gmy17] as problems involving a bounding operator that is very high compared to the rest of the algorithm, but where the cost of evaluating one node is sufficiently small to be efficiently performed by a single GPU-thread. An example of such problems is the PFSP, where it has been shown that the bounding function consumes up to 99% of the sequential execution time [MCB14]. Most of the approaches in the literature tackle those problems by bounding several subproblems in parallel on the GPU, while managing the tree exploration on the CPU [Cha+13b; LE12]. Significant efforts have been dedicated to transferring larger portions of the algorithm to the GPU and minimizing the overheads associated with data transfers between the CPU and GPU. In [Gmy+16], a fully GPU-based B&B algorithm is proposed, utilizing the compact IVM data structure. Various lock-free WS strategies are explored to address workload irregularities, allowing explorers to exchange work units during a dedicated WS phase incorporated into the synchronous parallel exploration process.

Finally, for coarser-grained problems, where the bounding function is particularly heavy, accelerating the function itself often provides the best results [Dab+16; MCA13]. For instance, a GPU-accelerated B&B algorithm dealing with the blocking jobshop scheduling problem has been proposed [Dab+16]. The approach offloads subproblems to the GPU but uses a block-based parallelization for each node evaluation. As a result, fewer subproblems need to be offloaded to fully saturate the GPU compared to medium-grained problems.

2.4.3 Hybrid and distributed parallel B&B

Few works investigated the parallelization of B&B using multiple CPUs and GPUs in distributed heterogeneous systems. In [VD16], the authors examine the design of parallel B&B algorithms for large-scale heterogeneous computing environments with shared memory cores, distributed CPUs, and GPUs. Through extensive experimentation, the study highlights the importance of adaptive and hybrid load balancing to achieve linear speed-ups and reduce idle times, showing that intra-node parallelism and decentralized load balancing are key to handling locking issues and scaling distributed resources effectively. In [Cha+13a], the authors introduce a refined approach where branching, bounding, and pruning are parallelized on the GPU to reduce CPU-GPU communication

latency. An experimental comparison between concurrent and cooperative approaches reveals that the cooperative method enhances performance over a GPU-only strategy, while the concurrent approach provides no advantage. The authors highlight reducing CPU-GPU communication overhead as a key challenge and address this by introducing overlapping communication schemes and auto-tuning the size of offloaded task pools.

Computational grids, offering a vast pool of computational power, have also led to the formulation of some parallel B&B algorithms [MMT07; BMT12; Ans+02; Dro+12]. A compact encoding of work units is utilized in [MMT07] to minimize communication overhead in distributed B&B, along with a checkpoint-based fault-tolerance mechanism. Applied to the `Ta056` flowshop problem instance, this approach achieved an optimal solution within 25 days using 1,900 processors across 9 clusters. In fact, some of the largest known exact resolutions of COPs have been performed using grid computing technologies; another example being the resolution of hard quadratic assignment problem instances in [Ans+02], using the MW paradigm. In [BMT12], the proposed algorithm addresses the scalability limitations of traditional MW-based B&B algorithms by introducing a hierarchical MW paradigm, where inner nodes (masters) handle branching and leaf nodes (workers) explore sub-trees. Applied to the Flowshop scheduling problem, the authors demonstrates improved scalability and efficiency in large-scale computational grid environments. In [BMT14], the authors build on their previous approach by introducing an enhanced fault-tolerance mechanism.

2.4.4 PGAS-based parallel B&B

Generally, the literature on parallel PGAS-based optimization is very scarce. In [MAD13] and [Mun+14], the authors investigate the use of the Global address space Programming Interface (GPI) PGAS Application Programming Interface (API) and X10, respectively, for parallel local search metaheuristics (approximate optimization). The reported results show that good speed-ups could be obtained on some basic problem instances, but no comparison against MPI+X approaches is provided. In the context of parallel B&B algorithms, a PGAS-based productivity-aware design and implementation of distributed B&B for solving large COPs on multi-core systems is proposed in [Car+20]. The approach is based on Chapel and combines a sequential initial search with Chapel's parallel and distributed iterators for load balancing. Experiments on the PFSP demonstrate the expressiveness and productivity of Chapel compared to an MPI +Pthreads counterpart implementation, in addition to equivalent performance for the best results on 1,024 CPU cores.

PGAS programming models were originally designed to facilitate productive parallel programming at both the intra-node and inter-node levels in homogeneous parallel machines. However, facing the growing need to support accelerators, especially GPU accelerators, PGAS models have been extended to accommodate heterogeneous systems. Past approaches focus on extending existing compilers, for example to compile and optimize high-level data-parallel constructs for GPU execution [Sid+12; CBS11; Che+11]. It is also not rare that the user eventually writes a fully external GPU program that includes the host part (*i.e.*, GPU memory (de)allocation, host-device/device-host data

transfer) and the device part (*i.e.*, GPU kernels) from their primary language. In [HPS19; HPS23], the authors introduce the Chapel's `GPUIterator` and `GPUAPI` modules to facilitate the invocation of user-written low-level GPU programs and choose an appropriate abstraction level depending on the tuning scenarios, respectively. Finally, GPU-native support has recently become available in languages like Chapel, marking a significant advancement in easing GPU programming within the PGAS model [MWA24].

To the best of our knowledge, the only work investigating PGAS-based parallel B&B for heterogeneous systems is based on Chapel and the `GPUIterator` module [Car+21]. It combines a partial search strategy with pre-compiled CUDA kernels for more efficient exploitation of the intra-node parallelism. The combination of Chapel with CUDA aimed to address Chapel's lack of GPU support from a few years ago. Since then, the language has been expanded to include a native vendor-neutral GPU support, thus unifying the different levels of parallelism within a single language. Moreover, whereas the approach proposed by [Car+21] tackled permutation-based problems using a bitsets data structure, this thesis focuses on generic approaches with respect to the problem being solved.

2.5 Benchmark problems

This section provides a description of the benchmark problems considered in this thesis. Appendix A summarizes the instances solved, along with some execution statistics.

2.5.1 Permutation Flowshop Scheduling Problem

Because of many economic and industrial applications, the Permutation Flowshop Scheduling Problem (PFSP) is widely used in the literature, specifically the formulation with *makespan* criterion [HS05]. It consists in finding an optimal processing order (a permutation) for n jobs $\{J_1, \ldots, J_n\}$ on m machines $\{M_1, \ldots, M_m\}$, such that the completion time of the last job on the last machine (makespan) is minimized. Obeying a chain production principle, the processing of a job J_j on machine M_k can only start if processing of J_j is completed on all upstream machines $M_1, \ldots, M_{k-1}$. Processing job J_j on machine M_k takes a given indivisible amount of time p_{jk} and all jobs are to be processed in the same order on all machines. Figure 2.8 illustrates a PFSP instance consisting of $n = 3$ jobs and $m = 4$ machines. For $m \geq 3$, the problem is shown to be NP-hard [GJS76].

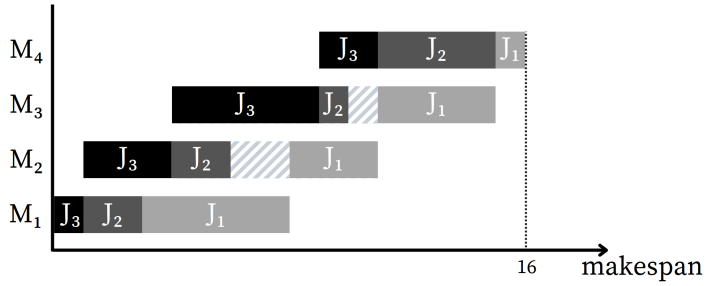
The following two lower bound functions are considered in this thesis:

- LB1: A variant of the so-called *one-machine bound* that can be computed in $\mathcal{O}(m)$ steps per subproblem [LLR78];

- LB2: The so-called *two-machine bound* that is known for its good results and has complexity of $\mathcal{O}(m^2 n \log(n))$ steps per subproblem. It is mainly based on Johnson's theorem [Joh54] which provides polynomial time procedure for finding an optimal solution for solving the 2-machine problem.

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

(a) Processing times.



(b) Optimal solution.

Figure 2.8: Illustration of a PFSP instance consisting of $n = 3$ jobs and $m = 4$ machines. The table shows the processing times of the jobs on each machine. The Gantt diagram shows the optimal solution.

The LB1 bound results in larger trees with a more fine-grained workload, while LB2 results in a more coarse-grained but also more irregular workload, due to the improved efficiency of the pruning operator.

Different branching strategies are also considered. The static *forward* branching scheme that constructs solutions from left to right, and the dynamic *minBranch* branching scheme proposed in [Gmy+20b]. Using a dynamic branching strategy, the algorithm must choose between the sets of subproblems constructed from left to right and right to left, respectively. Assuming that the lower bounds corresponding to these subproblems have been computed, the *minBranch* branching scheme locally minimizes the branching factor (the average number of children per node) by keeping the set where more nodes can be pruned. This strategy is known to produce the smallest tree size in most cases when combined with LB1, compared to other branching strategies.

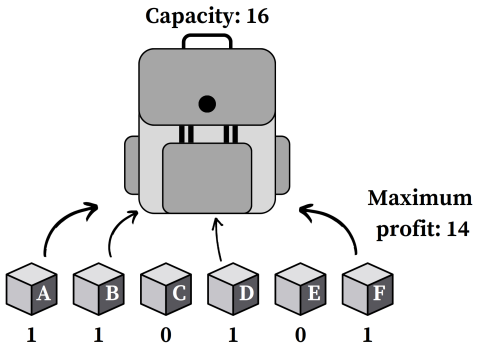
The most used benchmark instances considered in the literature are the ones defined by Taillard [Tai93]. The latter are indexed from `ta001` to `ta120` and are divided into 12 groups of 10 instances according to their size ($n \times m$): 20×5, 20×10, 20×20, 50×5, 50×10, 50×20, 100×5, 100×10, 100×20, 200×10, 200×20, and 500×20. When $m \leq 10$, the instances can be solved in few seconds using a sequential B&B. However, instances where $m = 20$ and $n \geq 50$ are very hard to solve. For example, proving the optimality of the 50×20 `ta058` required over 13 hours of processing on 256 NVIDIA V100 GPUs, and 339×10^{12} node decompositions [Gmy22]. Some instances remain unsolved, such as `ta051`, `ta054`, and `ta060`, more than 30 years after Taillard's benchmark release.

2.5.2 0/1-Knapsack problem

The NP-hard 0/1-Knapsack problem is one of the most intensively studied problems, as it appears in many practical situations [Lag96]. Given a finite set S of k items, a positive weight and profit for each item, along with a knapsack capacity W, the problem consists in finding a subset $S' \subseteq S$ such that the total profit is maximized while fulfilling the

	A	B	C	D	E	F
Profit	3	5	7	2	1	4
Weight	3	4	12	4	7	2

(a) Instance data.



(b) Optimal solution.

Figure 2.9: Illustration of a 0/1-Knapsack instance with 6 items. The table shows the profits and weights associated to each item, denoted from A to F. The array of binary variables on the right shows the optimal solution.

capacity constraint[5]. A solution is represented as an array of binary variables, where 1 means that the item is selected, and 0 means that it is not selected. Figure 2.9 illustrates a 0/1-Knapsack instance with 6 items.

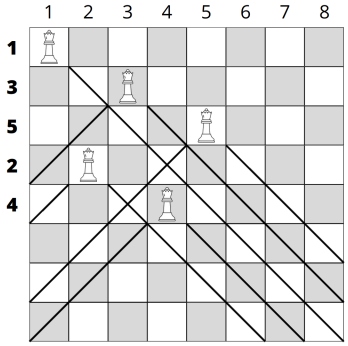
The upper bound function proposed by Dantzig is considered [Dan57]. It is an efficient method to determine in $\mathcal{O}(k)$ a solution to the continuous relaxation of the problem, hence providing an upper bound on the optimal solution. Solutions are constructed from left to right.

The instances proposed by Pisinger are used in this thesis [Pis05]. They are composed of multiple groups of randomly generated instances, in which the weights are uniformly distributed in a given interval with data range 1,000 and 10,000 and the number of items varies from 20 to 10,000. These instances are known to be hard, as there are numerous instances for which all currently known upper bounds perform badly, and for which the running times of the algorithms are close to the worst-case time-bound. The instances addressed in this thesis contain either 50 or 100 items and require anywhere from a few seconds to several hours to solve sequentially.

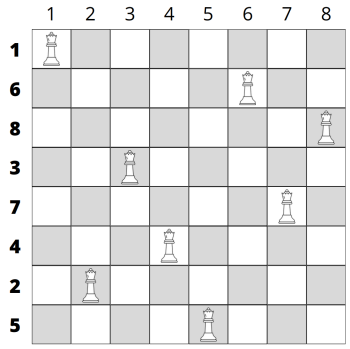
2.5.3 N-Queens problem

The N-Queens problem has been studied for over 170 years and is still widely considered in Computer Science today, particularly as a benchmark problem for constraint programming algorithms [EST92]. It consists in placing N queens on a N by N chessboard, one queen on each square, so that no queen captures any other, that is, the board configuration in which there exists at most one queen on the same row, column and diagonals. Several variants of the problem exist, but the one considered in this thesis is to enumerate *all* valid solutions.

[5]Without a loss of generality, we assume in this thesis that profits, weights, and W are positive integers.



(a) Partial solution that cannot be extended to achieve feasibility.



(b) Valid solution.

Figure 2.10: Illustration of the N-Queens problem for $N = 8$. The permutation representing the board configuration is displayed on the left side of the board.

The N-Queens problem can be modeled as a permutation problem, in which the value $x_j \in \{1, \ldots, N\}$ at index $j \in \{1, \ldots, N\}$ indicates that a queen is placed in column x_j of row j. As an example, Figure 2.10 illustrates an unfeasible partial solution along with a valid solution for $N = 8$. 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. Therefore, to evaluate the feasibility of a (partial) solution, it is enough to check for diagonal conflicts among the queens already placed.

Formally, N-Queens is not an optimization problem, but a constraint satisfaction problem. However, B&B can easily be adapted to solve such problems. 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. However, the size of the explored tree grows exponentially with N, so we consider the instances for $N = 15$ to 19. The largest known solution count to date is for the $N = 27$ instance that contains approximately 235×10^{15} solutions [PE17]. Permutations are only constructed from left to right, as the tree size cannot be reduced by constructing solutions from right to left due to symmetries.

2.5.4 Unbalanced Tree Search benchmark

The Unbalanced Tree Search benchmark (UTS) has been designed to evaluate the performance for parallel applications requiring dynamic load balancing [Oli+07]. It consists in counting the number of nodes in an implicitly constructed tree that is parameterized

in shape, depth, size, and imbalance. The tree is generated on the fly using a Random Number Generator (RNG) that allows the random stream to be processed in parallel while still producing a deterministic tree.

In these thesis, two representative tree shapes are considered:

- *Binomial trees*: Each node has q children with probability p and no children with probability $1 - p$. When $pq < 1$, this process generates a finite tree with expected size $\frac{1}{1-pq}$. The variation in subtree sizes increases dramatically as pq approaches 1, causing the tree to become imbalanced. A binomial tree serves as an optimal adversary for load balancing strategies because choosing to move one node over another provides no advantage for load balancing; the expected work at all nodes is identical. The root-specific branching factor b_0 can be set sufficiently high to generate a variety of subtree sizes below the root.

- *Geometric trees*: Each node has a branching factor that follows a geometric distribution with an expected value that is specified by the parameter $b_0 > 1$. Since the geometric distribution has a long tail, some nodes will have significantly more than b_0 children, yielding unbalanced trees. The parameter d specifies the maximum, beyond which the tree is not allowed to grow. Unlike binomial trees, the expected size of the subtree rooted at a node increases with proximity to the root.

Multiple instances of a tree type can be generated by varying the seed r for RNG state at root.

Chapter 3

PGAS-based Parallel B&B for CPU-based Clusters

Contents

This chapter investigates the design and implementation of a PGAS-based parallel B&B for CPU-based clusters, leveraging PGAS's ability to efficiently manage memory access and enhance parallelism in distributed computing environments. Several challenges have to be addressed, such as ensuring memory consistency, achieving effective load balancing, and minimizing communication overhead associated with shared data structures. Additionally, scalability becomes a concern as the number of nodes increases, necessitating efficient management of inter-node communication and synchronization.

Section 3.1 contributes a PGAS-based data structure, called `distBag_DFS`, implementing a highly parallel segmented multi-pool specialized for unbalanced DFS. This data structure integrates a dynamic load-balancing mechanism based on large-scale WS,

operating at both intra- and inter-node levels. This mechanism, which required sophisticated synchronization, promotes locality in WS, enabling scalability.

Then, Section 3.2 presents the `P3D-DFS` algorithm, a generic `distBag_DFS`-based parallel B&B algorithm for CPU-based clusters. Crucial aspects of the algorithm, such as termination detection, are discussed.

Both data structure and parallel B&B algorithm are then thoroughly tested in Section 3.3, which includes evaluations of memory consumption, the dynamic load balancing mechanism, the strong scaling efficiency considering both intra- and inter-node levels, comparison with an MPI+X counterpart implementation, and large-scale experiments conducted on a petascale system. Finally, Section 3.4 draws the conclusion of this chapter.

3.1 The PGAS-based `distBag_DFS` data structure

This section provides a comprehensive description of the `distBag_DFS` data structure. Each of its internal components plays a crucial role in exploiting intra- and inter-node parallelism of the target multi-core architecture and managing data efficiently. In addition, the dynamic load balancing mechanism based on WS is forwarded.

3.1.1 Origins

The `distBag_DFS` data structure presented in this chapter is a revisit of an existing PGAS data structure, called `distBag` [JFK17]. The latter is a parallel-safe data structure that can be used across multiple threads across multiple nodes, and support basic operations that any data structure needs, such as insertion/removal of an arbitrary element and iteration over all elements. More specifically, `distBag` implements a generic unordered distributed multi-set and employs a load balancing mechanism based on WS to balance work across nodes.

Internally, `distBag` maintains multiple *bag*, one per compute node, themselves composed of multiple sets, called *segments*, implemented as unrolled linked-lists. The parallel-safety of segments is guaranteed by a spin-lock on each segment. In the following, we refer to *pool* and *segment* without distinction. By default, there are as many segments per node as threads per node. However, pools are not explicitly mapped onto threads. Actually, multiple pools are maintained in parallel to reduce lock contention, but threads remove and insert elements from any (not necessarily the same) unlocked pool. In practice, this "unordered" characteristic of `distBag` may cause memory issue, as in the context of tree exploration. For instance, when child nodes are inserted into a different pool than the one from which the parent node was taken, no exploration strategy can be followed. As a direct consequence, large number of nodes may be generated and stored in the data structure, leading to large memory requirements. This will be experimentally confirmed in Section 3.3.1.

The parallel distributed nature of the PGAS `distBag` data structure, as well as the underlying load balancing mechanism make it particularly attractive in the context of

parallel productivity-aware tree-search. However, its segments' scheduling policy does not allow us to use it for parallel DFS B&B, nor to have any control over the order of insertion/retrieval of elements. This motivates our revisit of the data structure into `distBag_DFS`, a variant specialized for parallel DFS.

3.1.2 Hierarchical structure and core components

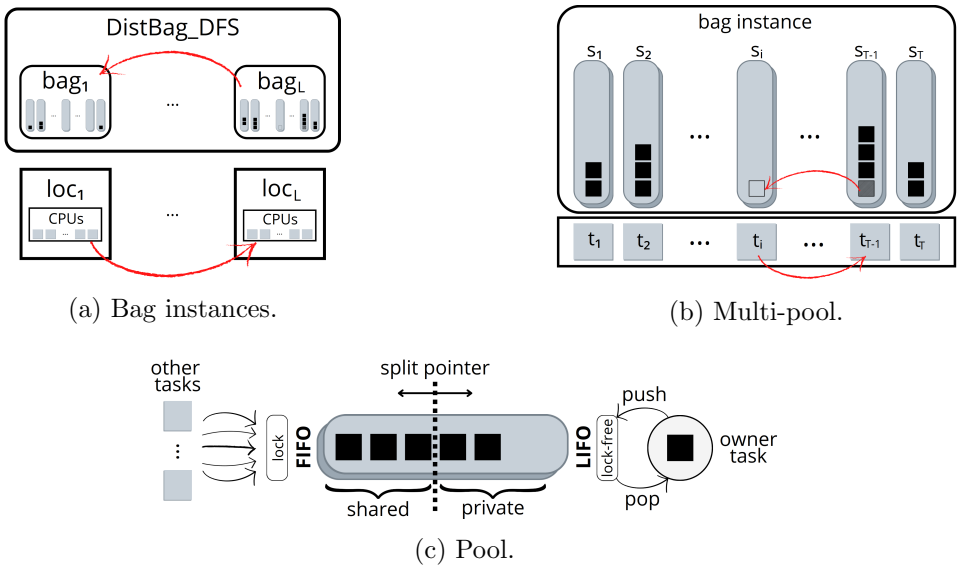
Figure 3.1 illustrates the hierarchical organization of the data structure. In order to account for the Non Uniform Memory Access (NUMA) characteristics of large-scale systems with multi-core compute nodes, `distBag_DFS` maintains a separate *bag instance* (multi-pool) for each locality, as depicted in Figure 3.1a. A locality represents a subset of the target architecture, used to manage and optimize affinity for improved performance and scalability. Typically, a locality corresponds to a single compute node so that it can access its local memory with a comparably uniform and minimal cost. Accessing data from different localities, however, comes at a higher cost. Although the data structure supports distributed use, it allows for the creation of privatized bag instances to maximize performance. Each locality thus operates on its own privatized instance.

As shown in Figure 3.1b, each instance of a bag contains multiple pools, hereafter called *segments*. Specifically, one segment is assigned to each parallel thread (denoted as T in the figure). Each thread is uniquely identified by indexes from 1 to T, which are used to map threads to their corresponding segments. During the insertion and retrieval procedures of `distBag_DFS`, the segment is specified to indicate where a tree node is inserted into or retrieved from. This specification is crucial for DFS because it ensures that when a node is evaluated, its entire subtree is explored before processing another sibling node. If child nodes are inserted into a different segment than their parent, the DFS condition cannot be guaranteed. It is however important to note that while each segment ensures a local DFS order, this order is not guaranteed across the multi-pools.

Segments are implemented using non-blocking split deques, as described in [DP14]. Each segment is logically divided into "shared" and "private" regions using an atomic *split pointer*, illustrated in Figure 3.1c. This design enables lock-free access to the private section of the deque locally, and facilitates transfer of work between the shared and private regions without copying. Work transfer occurs by adjusting the split pointer bidirectionally using appropriate operators. Thread access the shared region for load balancing, synchronizing themselves using an atomic lock. Segments are initially sized with a capacity of 1,024 elements, and when a segment reaches full capacity, it expands its size exponentially by a power of two. To prevent uncontrolled growth and excessive memory usage, `distBag_DFS` sets a maximum capacity limit for each segment. However, this limit is rarely reached due to the DFS exploration order, which ensures limited memory consumption over time.

3.1.3 Locality-aware dynamic load balancing

Load balancing ensures that work is evenly distributed among processing units, minimizing idle time and maximizing parallel exploration benefits. By effectively distributing the



(a) Bag instances.

(b) Multi-pool.

(c) Pool.

Figure 3.1: Illustration of the `distBag_DFS` components: (a) bag instances, (b) multi-pool, and (c) pool based on non-blocking split deque. Red arrows represent potential work stealing operations.

workload, the overall performance of parallel B&B algorithms is significantly enhanced, resulting in a substantial reduction in solution time.

`distBag_DFS` is equipped with a locality-aware dynamic load balancing mechanism based on WS, as shown in Figure 3.1 by the red arrows. Algorithm 3.1 describes the pseudo-code of the WS mechanism.

- *Intra-node work stealing*: The algorithm starts by iterating over potential victim threads (line 1), according to a victim selection iterator forwarded later. For each victim thread, the algorithm retrieves the corresponding target segment from the multi-pool (line 2). It then acquires the atomic lock on this segment to ensure that no other thief thread can modify it concurrently, which prevents race conditions (line 3). Once the segment is locked, the algorithm checks if the shared part of the latter contains at least *minElts* elements. The *minElts* parameter prevents stealing from a pool that already contains few elements. If this condition is met, the thief thread steals an element before releasing the lock (lines 5-6). This element is finally returned along with a boolean flag—SUCCESS or FAIL—showing the status of the operation (line 7). If the shared segment does not have enough elements (*i.e.*, segment's size $<$ *minElts*), the algorithm checks if the private segment has at least one element. If so, it sends a request to the victim thread asking to expand the shared portion of the segment for future stealing attempts (line 9). Finally, the local WS attempts fail if all the victim threads have been visited and no element was stolen. In distributed setting, the triggers the inter-node WS attempts.

- *Inter-node work stealing*: In contrast to the intra-node approach, the algorithm starts by iterating over localities (line 14). For each victim locality, the algorithm executes the subsequent block of code on the remote locality, meaning that the following is performed on the remote bag instance (line 15). Then, similarly to the intra-node WS approach, the algorithm iterates over threads within the remote locality (line 16). One key difference is the amount of tree nodes that are stolen at once. In this approach, half of the private region of the segments is stolen, aiming at reducing the total number of inter-node WS operations (line 20). In addition, all threads are visited once. However, if at least one steal has been successful, the algorithm breaks out the loop iterating over victim localities (lines 26-27). This avoids unnecessary attempts to steal work once sufficient work has already been stolen. Finally, the stolen nodes are inserted back to the thief threads's segment, except one that is returned (lines 29-30). The inter-node WS attempts fail if all the threadss from all the localities have been visited and no element was stolen. In that case, nothing is returned along with a FAIL flag.

The design choices regarding the victim selection policy and the granularity policies are discussed in the following.

3.1.3.1 Victim selection policy

The victim selection strategy defines how a thief thread selects its target(s). One of the most commonly used and provably efficient policies in the literature is the random selection policy [BL99]. It assumes that a victim thread is selected uniformly at random when a thread initiates a WS operation. Algorithm 3.2 describes the approach used by `distBag_DFS`. Assuming that T threads are executing per locality and that the WS operation is initiated by thread $threadId \in \{0, \ldots, T-1\}$, the algorithm iterates over the randomly permuted list of thread indices, yielding each index except the one matching $threadId$. The algorithm is generic and is employed to select victim threads and/or localities, as shown in lines 1, 14, and 16 of Algorithm 3.1.

3.1.3.2 Granularity policies

The granularity policy determines how many work items are stolen per WS operation. Its implementation is guided by two factors: (1) the depth-first search ordering of `distBag_DFS`, and (2) the locality-awareness of the dynamic WS mechanism.

The depth-first ordering guarantees that the tree nodes at the end of the segment, which are shared among the threads, are the shallowest in the tree. For a wide variety of COPs, these tree nodes also tend to have a large branching factor. Indeed, when solving COPs, there is generally a high degree of freedom in how the tree nodes are branched initially, leading to a drastic increase in the number of possible sub-configurations. As one delves deeper into the tree, constraints accumulate, reducing the number of partially feasible solutions and thus the size of the subtrees. This is particularly true for permutation problems. For instance, in the absence of pruning, solving a permutation problem

Algorithm 3.1: Pseudo-code of `distBag_DFS`'s work stealing mechanism.

```
   input : threadId - index of the calling thread
           T - number of threads
           LocaleId - index of the calling locality
           L - number of localities

   // INTRA-NODE WORK STEALING ATTEMPTS
 1 for victimThreadId from victim_iterator(threadId, T) do    /* see Alg. 3.2
    */
 2     targetSegment ← segments[victimThreadId];
 3     targetSegment.acquireLock();
 4     if (targetSegment.shared.size ≥ minElts) then
 5         elt ← targetSegment.stealElement();
 6         targetSegment.releaseLock();
 7         return (elt, SUCCESS);
 8     else if (targetSegment.private.size ≥ 1) then
 9         targetSegment.splitRequest();
10     targetSegment.releaseLock();
11 if (numLocales > 1) then
       // INTER-NODE WORK STEALING ATTEMPTS
12     steal ← false;
13     stolenElts ← [ ];
14     for victimLocaleId from victim_iterator(LocaleId, L) do
15         go on Locales[victimLocaleId];
16             for victimThreadId from victim_iterator(threadId, T) do
17                 targetSegment ← segments[victimThreadId];
18                 targetSegment.acquireLock();
19                 if (targetSegment.shared.size ≥ minElts) then
20                     elts ← targetSegment.stealElements();
21                     stolenElts.insert(elts);
22                     steal ← true;
23                 else if (targetSegment.private.size ≥ 1) then
24                     targetSegment.splitRequest();
25                 targetSegment.releaseLock();
26         if (steal = true) then
27             break;
28     if (steal = true) then
29         segments[threadId].insert(stolenElts);
30         return (segments[threadId].get(), SUCCESS);
31 return (NULL, FAIL);
```

Algorithm 3.2: Pseudo-code of the random victim selection policy iterator.

```
input : myId - index of the calling thread/locality
        N - number of threads/localities
1 id ← 0;
2 victims ← permuteRand([0, ..., N − 1]);
3 while (id < N) do
4     if (victims[id] ≠ myId) then
5         yield victims[id];
6     id ← id + 1;
```

generates a tree in which each tree node produces $s - d \geq 0$ children, where $d \in [0, s]$ is the depth of the node and s is the size of the permutation. Given this observation, the granularity of our intra-node WS mechanism is fine, involving the stealing of a single node, as it is assumed to contain a significant amount of work. However, at the inter-node level, half of the available nodes in the shared region of the victim thread's pool are stolen, as taking only one node is insufficient to fully occupy the thief thread and the other threads sharing its locality.

3.2 `distBag_DFS`-based parallel B&B (`P3D-DFS`)

This section presents the overall design of the proposed `P3D-DFS` PGAS-based parallel B&B algorithm based on the `distBag_DFS` data structure.

3.2.1 Overall design of `P3D-DFS`

The `distBag_DFS` data structure is the cornerstone of the proposed algorithm, providing the implementation of multiple parallel-safe pools and encapsulating a locality-aware dynamic load balancing mechanism, transparently to the B&B algorithm. This multi-pool configuration allows to design efficiently a parallel tree exploration model, whose main advantage is the potential high degree of parallelism. By leveraging the `distBag_DFS` structure, the algorithm can effectively manage and distribute computational tasks among multiple CPU threads, reducing contention and enhancing throughput. This results in a more efficient exploration of the search space, as threads can operate independently while still benefiting from localized data access.

Aside from the potentially high degree of parallelism of the parallel tree exploration model used, the genericity is also a major advantage. First, `P3D-DFS` is independent from the problem solved, as the parallel design does not affect the bounding operator. Then, the `distBag_DFS` data structure is itself generic with regards to the element type contained, which means that for the same problem, different solution encodings can also be considered. The extensibility of `P3D-DFS` to different optimization problems will be discussed more in details in Chapter 5.

The implementation complexity of parallel B&B algorithms, which is usually concentrated on the distributed aspects, notably the load-balancing mechanism and the management of a data structure, are now handled transparently by the `distBag_DFS` data structure. This results in an algorithm with a high level of abstraction, where the last major design challenge is the detection of global termination.

3.2.2 Detecting global termination

In asynchronous environments, where B&B processes operate independently and at different speeds, detecting global termination becomes particularly challenging. This refers to the process of determining when all B&B processes have completed their tasks and no further subproblems are pending. Several strategies exist for termination detection, including the use of tokens, or consensus algorithms that help identify completion across distributed processes [MC98]. These methods often involve maintaining a record of active threads and their states, enabling the system to determine if any ongoing computations are still in progress.

Figure 3.2 shows a flowchart of the global termination detection algorithm implemented in `P3D-DFS`. It can be considered as a "wave algorithm", as the initiator makes a decision, the decision is preceded by an event in each process, and the wave terminate [MC98]. It assumes that each threads maintains a state variable in the global address space, either set to `BUSY` or `IDLE`. A state variable is also assigned to each locale, and is set to `BUSY` if at least one of its threads is busy; `IDLE` otherwise. The algorithm is initiated when a thread becomes idle during execution, meaning that its work pool is empty, and all WS attempts fail. In that case, the initiator first checks the states of the other threads from its locality (step 1), after setting its own state to `IDLE`. If at least one thread is busy, then the termination detection ends and the initiator continues the B&B algorithm. Otherwise, the locality state is set to `IDLE` and the states of the other localites is checked (step 2). If all localities are found to be idle, the algorithm triggers the global termination of the B&B algorithm. In this PGAS-based global termination detection algorithm, the initiator read the state of each other thread from the global address space, in a one-sided manner. Since some of the state variables are located in remote memory areas, we implement the global termination detection mechanism in a locality-aware manner. The first step allows to check the state of each thread from the same locality at a relatively low cost, performing Uniform Memory Access (UMA) at the intra-node level. Then the second step, involving remote memory accesses at the inter-node level, is triggered only if the first step does not allowed the initiator to make a decision. This method aims to minimize the number of remote communication, hence reducing the associated overhead.

To ensure that the state read by the initiator is consistent with the actual state of the thread, we implement states as atomic variables. More precisely, each locality maintains an array of atomic variables (one per thread), and each thread updates its state during execution. In other words, the arrays store each thread's state in contiguous memory to facilitates the finding of the global-state, while each state is only modifies by its assigned thread. In practice, we observed that this implementation generates *false sharing* [BS93],

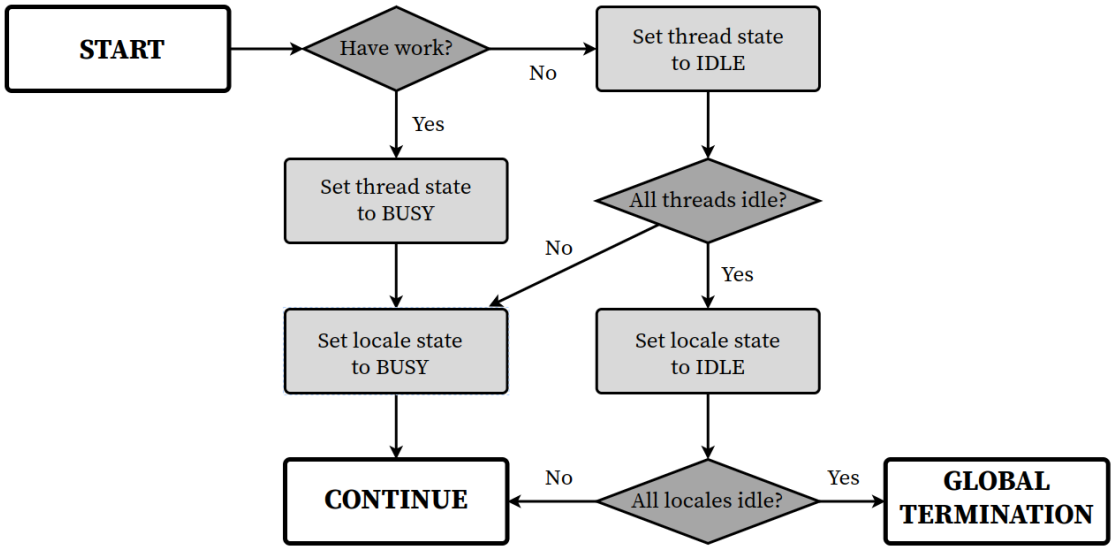


Figure 3.2: Flowchart of the global termination detection.

as illustrated in Figure 3.3. The figure assumes that two CPU cores, each with their own L1 and L2 caches, share an L3 cache. Variable x is loaded by core 1 (left) into its L1 and L2 caches, and variable y (located in the same cache line as x) is loaded by core 2 (right). Since x and y are in the same cache line, the modification of x by core 1 (blue arrows) invalidates the corresponding cache line in core 2's caches (red arrows). If core 2 then accesses variable y, it finds that its cache line has been invalidated, and have to reload the cache line. Reloading can happen from L3 cache if the cache line modified by core 1 is still present in the cache, or RAM if the cache line has been evicted from L3 (for example, by other memory accesses). Even though core 1 and core 2 are working on different variables (x and y), they continue to invalidate each other's cache lines due to the proximity of the variables in memory, creating "false sharing". Usually, two different ways exist to reduce such phenomenon: make sure that unrelated data are stored in different cache lines, and use local data for intermediate calculation and then access shared memory when necessary. In the implementation of the global termination detection algorithm in `P3D-DFS`, we adopted the second approach and introduced local data *threadState* for intermediate checking, as shown in Algorithm 3.3.

3.3 Experiments

In this section, we report the experimental results for the data structure and algorithm presented in this chapter. Unless explicitly mentioned, all the experiments are performed on a cluster, where each compute node is equipped with two 64-core AMD EPYC Rome 7H12 @ 2.60 GHz CPUs, and 256 GB of RAM. Compute nodes are connected through an InfiniBand HDR 100 Gb/s network, configured over a Fat-Tree topology. The code is compiled and executed using Chapel 2.1.0, along with the gcc 10.2.0 back-end compiler.

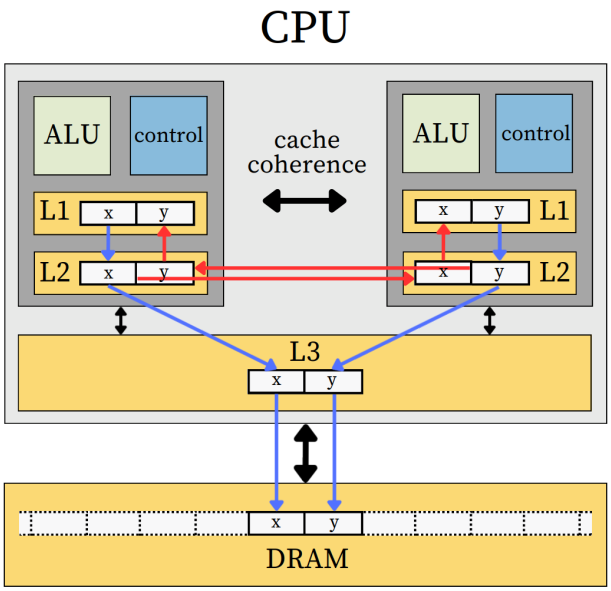


Figure 3.3: Illustration of false sharing in shared-memory systems.

Algorithm 3.3: Reducing false sharing in global termination detection.

input : *threadId* - index of the calling thread
threadStates - array of atomic thread states
threadState - private copy of *threadStates*[*threadId*]

```
1 if (threadState = IDLE) then
2     threadState ← BUSY;
3     threadStates[threadId].write(BUSY);
```

Appendix B contains the environment configurations used to build and execute Chapel code.

3.3.1 Comparison with other data structures

This section aims to demonstrate experimentally the limitations of the original `distBag` data structure in terms of memory usage. As a test-case, the N-Queens instances are solved with $N = 15$ and 16 using the parallel B&B algorithm presented in this chapter, based on `distBag` and our revisited version `distBag_DFS`. The number of threads is set to 4 and the bag size according to the execution time is shown in Figure 3.4. The bag size is expressed in terms of subproblems, which is an indicator of the overall memory usage of the program.

First of all, we can observe that the `distBag`-based parallel B&B stores up to 3.2×10^7 pending subproblems at once for $N = 15$, and up to 5×10^7 for $N = 16$. The latter experiment reached the limit we had set ourselves so as not to risk damaging the system's memory. As each subproblem contains at least 224 bits of data, this represents $224 \times 5 \times 10^7 = 11.2 \times 10^9$ bits, or 1.4 GB of memory. The bell curve observed solving the 15-Queens instance can be explained by the fact that in the upper parts of the search tree the average branching factor is higher than in the bottom parts. The shallowest nodes in the tree have few queens placed and therefore few constraints, which results in a high number of child nodes. However, as we go deeper into the tree, constraints accumulate, and pruning occurs more frequently, leading to the drain of the bag.

The same experiments using the `distBag_DFS`-based parallel B&B revealed that, for both instances, the size of the bag, and thus the memory consumption, remains bounded over time. Theoretically, in a DFS B&B algorithm applied to a permutation problem of size n, the maximum size that a pool can contain is $\sum_{i=1}^{n-1} i$. Indeed, by definition, the subtree of a node must be completely explored before branching another node of the same depth. Therefore, for each depth l, at most $n - l$ nodes are kept in memory (the generated but not yet evaluated subproblems). Since here we have 4 threads, thus 4 pools in parallel, this means that the number of pending subproblems that cannot be exceeded is 420 for $N = 15$ and 480 for $N = 16$. This rule is satisfied, as the bag size never exceeds 220 at a time.

3.3.2 Dynamic load balancing mechanism

In this section, the goal is to evaluate the `distBag_DFS`'s dynamic load balancing mechanism. For that purpose, `P3D-DFS` is instantiated on the UTS benchmark and two synthetic trees with different types—binomial and geometric—are solved using up to 128 CPU cores. In order to allow a fair comparison between both instances, we made sure that the sequential times are approximately the same.

Figure 3.5 shows that the best performance is achieved with the `UTS-geo` instance, achieving 59% of the ideal speed-up when using 128 processing cores. In contrast, the `UTS-bin` instance achieves only 26% of the ideal speed-up. Table 3.1 provides some execution statistics of the solved instances. For both instances, the percentage of WS

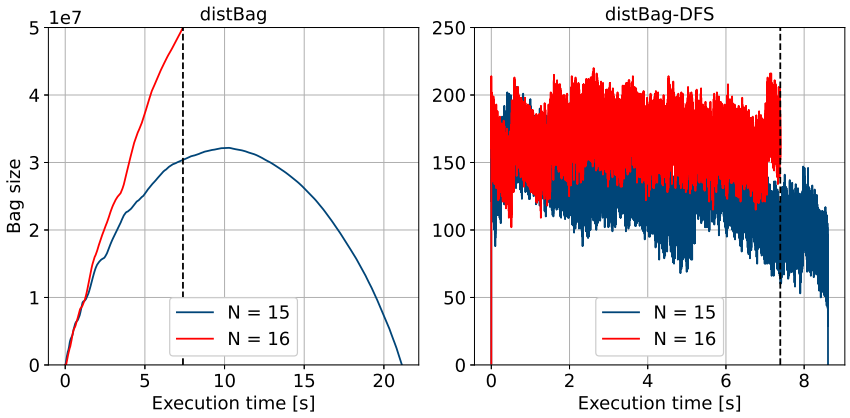


Figure 3.4: Bag size (in subproblems) according to the processing time when solving the 15-Queens and 16-Queens instances.

Table 3.1: Summary of the UTS instances solved, along with some execution statistics.

Instance	# nodes (10^6)	Time (s)	kn/s	# WS attempts (% success)
`UTS-geo`	91.4	36.06	2,534.6	48,433 (99.0%)
`UTS-bin`	131.7	36.30	3,628.1	1,473,048 (96.8%)

attempts failed is less than 4%, which demonstrates the efficiency of the WS. However, we observe that the number of WS attempts is 30 times greater when solving the `UTS-bin` instance compared to the `UTS-geo` instance. This can be explained by the tree shape, and more precisely the branching factor. In a geometric tree, the expected size of the subtree rooted at a node increases with proximity to the root, meaning that shallowest nodes have a higher branching factor than the others, leading to the generation of many subproblems. This perfectly fits `distBag_DFS`'s load balancing mechanism as shallowest nodes are stolen first. However, in a binary tree the expected work at all nodes is identical, *i.e.*, at most two child nodes per branching operation. Therefore, stealing the shallowest nodes does not provide any benefits and only one subproblem is not sufficient to allow the generation of new subproblems, leading to many WS attempts. This causes contention on the data structure, hence degrading overall parallel efficiency.

Figure 3.6 shows the percentage of explored tree nodes per processing core solving the `UTS-bin` instance, hence providing the workload distribution of the overall B&B tree. Even in this challenging scenario, the total workload is almost evenly balanced among all the processing cores, meaning that all the allocated resources are fully exploited.

3.3.3 Strong scaling efficiency

Figure 3.7 shows the strong scaling efficiency of `P3D-DFS` instantiated on three different problems—PFSP, 0/1-Knapsack, and N-Queens—considering both intra- and inter-node parallel levels. For each problem, a set of instances of different sizes have been selected.

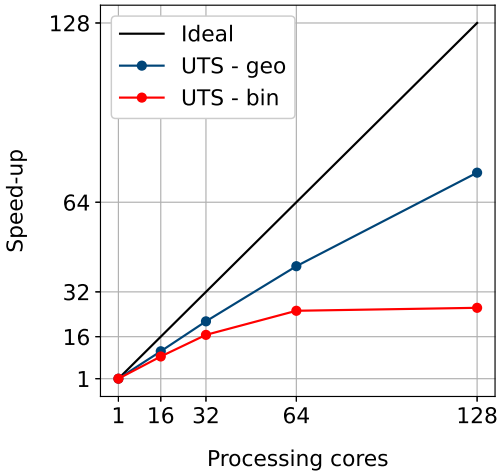


Figure 3.5: Speed-up achieved solving geometrical and binomial synthetic UTS trees, compared to a sequential version.

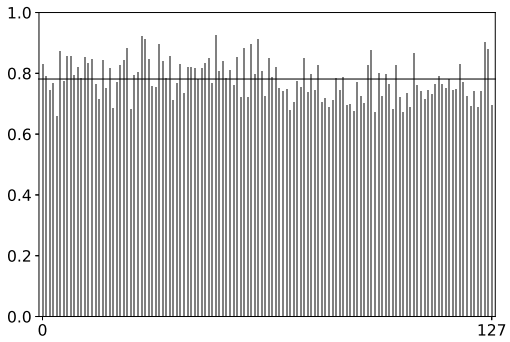
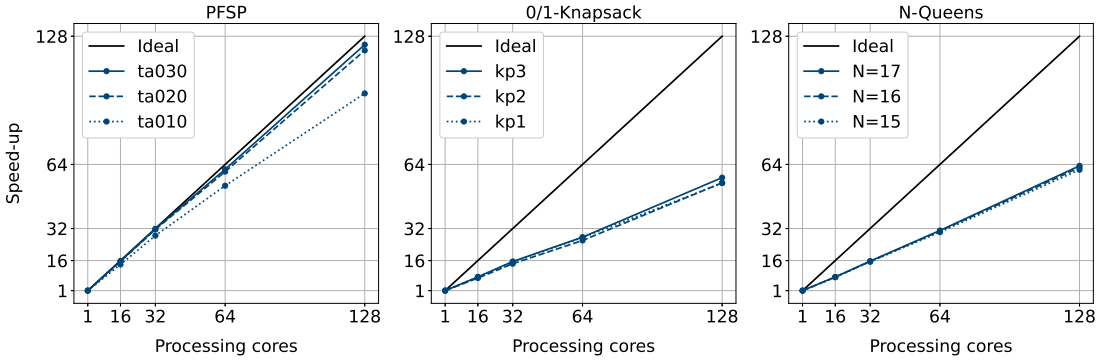


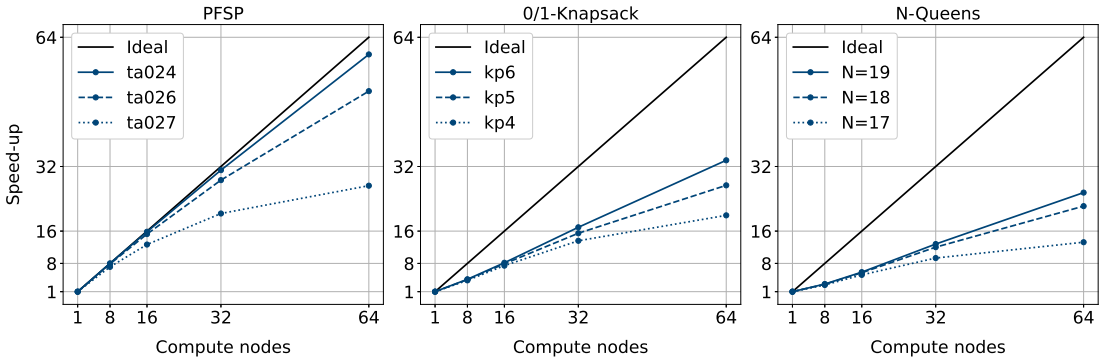
Figure 3.6: Percentage of explored nodes per tasks solving the `UTS-bin` instance. The black lines represent the ideal percentage, *i.e.*, $100/nTasks$.

For the PFSP, the LB2 lower bound function is used along with the static forward branching rule. At the intra-node (resp. inter-node) parallel level, the speed-up is computed as the ratio of the sequential (resp. single-node) algorithm over the multi-core (resp. multi-node) variant.

At the intra-node parallel level, the strong scaling efficiency ranges from 43% to 97% of the ideal speed-up using 128 CPU cores. The best results are achieved solving the PFSP instances, where near-linear speed-ups are observed on the largest instances. On the 0/1-Knapsack problem, the performance of `P3D-DFS` is quite limited, where at most 45% of strong scaling efficiency is reached using 128 cores. These results are directly related to the previous section, where we shown the limitation of `distBag_DFS`'s load balancing mechanism exploring binary trees. Solving the N-Queens instances, the performance is also hindered, achieving only 50% of the ideal speed-up. The poor results highlight the challenges posed by the fine-grained node evaluation function. Table 3.2 shows execution statistics of the largest instance solved for each problem using 128 CPU cores. Solving the PFSP `ta030` instance, 98% of the total execution time is dedicated to the decomposition of nodes, which allows the high-performance of our parallel tree-exploration model. In contrast, the fine-granularity of the 17-Queens instance enables to decompose much more nodes per second, but also adds more contention on the data structure, as 28% is dedicated to remove/insert nodes from/to the data structure. Specifically, the frequent synchronization and communication required in such a fine-grained scenario lead to significant overhead, which detracts from the advantages of parallel execution. Solving the 0/1-Knapsack `kp3` instance, we can see that the algorithm spends almost as much time decomposing nodes as it does retrieving them from the data structure (including the WS operation).



(a) Intra-node parallel level.



(b) Inter-node parallel level.

Figure 3.7: Strong scaling efficiency of `P3D-DFS` instantiated on three different problems, at both intra- and inter-node parallel levels. Three instances of different sizes are solved for each problem.

At the inter-node parallel level, similar observations are made. The strong scaling efficiency ranges from 20% to 94% of the ideal speed-up using 64 compute nodes, or 8,192 CPU cores. As the instance size increases, speed-up improves because larger instances include a greater number of independent tasks that can be processed simultaneously. This allows a more efficient distribution of the workload across the available compute nodes.

3.3.4 Comparison against an MPI+X approach

This evaluation compares `P3D-DFS` to a low-level state-of-the-art implementation, hereafter called `MPI-PBB` [Gmy17]. This latter is a hierarchical MW B&B written in C++ and MPI+PThreads, compiled and executed using OpenMPI 4.0.5. An interval-based encoding of work units and the IVM data structure [Gmy+17] are used for the implementation of DFS. Each MPI worker can be configured to use multiple worker threads performing local WS operations for load balancing on the intra-node level. A dedicated

Table 3.2: Execution statistics of the largest instance solved for each problem using 128 CPU cores.

Instance	kn/s	Percentage of total execution time			
		Remove	Decompose	Insert	Termination
`ta030`	2,204.7	$< 1\%$	98%	$< 1\%$	$< 1\%$
17-Queens	269,751.7	18%	62%	10%	8%
`kp3`	161,505.5	42%	47%	5%	4%

thread is used for communication with the master process, allowing to overlap work progress and communication efficiently. To overlap computation and communication, redundant exploration of some parts of the search space is tolerated. Inter-node workload balancing is performed by the intermediate of the centralized coordinator process. In the following experiments, 16 threads (plus one additional communication thread) are used per MPI process, for a total of 8 MPI processes per compute node. Node 0 runs the master process using the same configuration, meaning that it hosts at most 7 worker processes (112 threads). It is worth to mention that `P3D-DFS` and `MPI-PBB` follow the DFS search strategy as selection rule, and implement the same bounding functions introduced in Section 2.5.1. However, they divergence in some aspects, for example the data structure used to store pending subproblems. For a fair comparison, both implementations enumerate equivalent search spaces for proving the optimality of a solution.

Figure 3.8 reports the execution time of `P3D-DFS` for solving the PFSP instances `ta021-ta030` to optimality, relatively to the `MPI-PBB` baseline. Results are shown for 1 (128 cores) to 64 compute nodes (8,192 cores) and instances are sorted by the number of nodes. In almost all configurations (except the biggest instances on one node), `P3D-DFS` is faster than `MPI-PBB`. The most significant results are obtained solving the smallest instances (`ta029`, `ta030`, and `ta022`), where `P3D-DFS` is up to 65% faster than the baseline. The poor performance of `MPI-PBB` compared to `P3D-DFS` can be explained by at least two factors. First, the presence of a master process centralizing the work-pool has the advantage of making termination detection trivial, but becomes a substantial bottleneck as the number of nodes increases. Then, in general, "the results show that fine-grained problems with computationally inexpensive node evaluation functions can benefit most from using IVM" [Gmy17]. As a result, using the specialized IVM structure to solve PFSP instances with the coarse-grained LB2 bounding function offers little or no advantage over `distBag_DFS`. As instance size increases, we can notice that the performance gap narrows, to the point where `P3D-DFS` becomes slightly slower than `MPI-PBB` on one node.

Figure 3.9 shows the speed-ups achieved by `P3D-DFS` and `MPI-PBB` on 2 to 64 compute nodes. Results ranging from 13% (`ta030`) to 95% (`ta021`) of the linear speed-up on 64 nodes are observed for `P3D-DFS`. For the smaller instances (`ta029`, `ta030`, `ta022`, and `ta027`), severe speed-up decreases are observed when comparing the results on 8 to the ones on 64 locales. For this subset of smaller instances, the size of the explored tree is

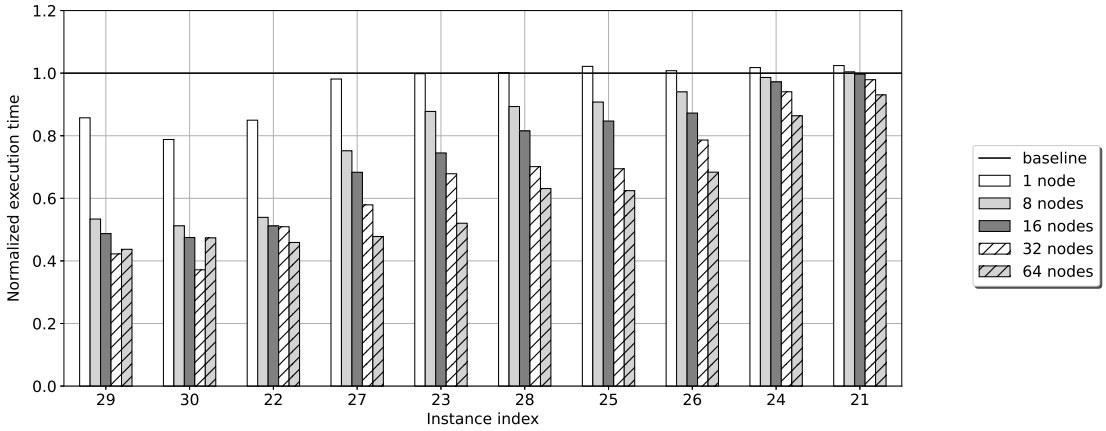


Figure 3.8: Execution time of `P3D-DFS` for solving instances `ta021-ta030` to optimality. Results shown are for 1 (128 cores) to 64 compute nodes (8,192 cores). For each configuration, the execution time is given relative to the `MPI-PBB` baseline. Instances are sorted by the number of nodes.

not large enough to fully utilize the computational resources available on 64 nodes. As a result, there is insufficient workload to distribute across all nodes, leading to suboptimal parallel efficiency. In turn, we can see that `MPI-PBB` is outperformed in all configurations. The scalability is up to 50% smaller on 64 nodes for the set of smaller instances, while only 10% solving the largest `ta021` instance. This shows that the use of the `distBag_DFS` data structure and its load-balancing mechanism are perfectly suited to solving coarse-grain PFSP instances using multiple compute nodes, whereas the approach used by `MPI-PBB` suffers from the presence of a centralized coordinator process.

3.3.5 Large-scale experiments

This section investigates the performance of `P3D-DFS` at scale on the MeluXina petascale system[6]. All resources are deployed to solve hard PFSP instances and confirm the optimality of solutions given in the literature. For these experiments, the dynamic *minBranch* branching rule is used along with the LB1 bounding function. Up to 400 compute nodes are used, each composed of two 64-core AMD EPYC Rome 7H12 @ 2.60 GHz CPUs, for a total of 128 cores, and 512 GB of RAM. The compute nodes are interconnected using the InfiniBand HDR 200 Gb/s high-speed fabric and operate under Rocky Linux 8.7. Chapel 1.31.0 is used in a fine-tuned configuration environment, along with the gcc 11.3.0 back-end compiler.

Figure 3.10 shows the strong scaling efficiency reached solving the 50×20 `ta056` PFSP instance up to 400 nodes. The latter instance exhibits 173×10^9 nodes and requires 4.0 CPU-hours. The experimental results revealed that up to 70% of the ideal speed-up can be achieved using up to 128 computer nodes, and around 50% using 400 nodes. In the

[6]Cluster module, ranked 460^{th} in the TOP500 release of June 2024.

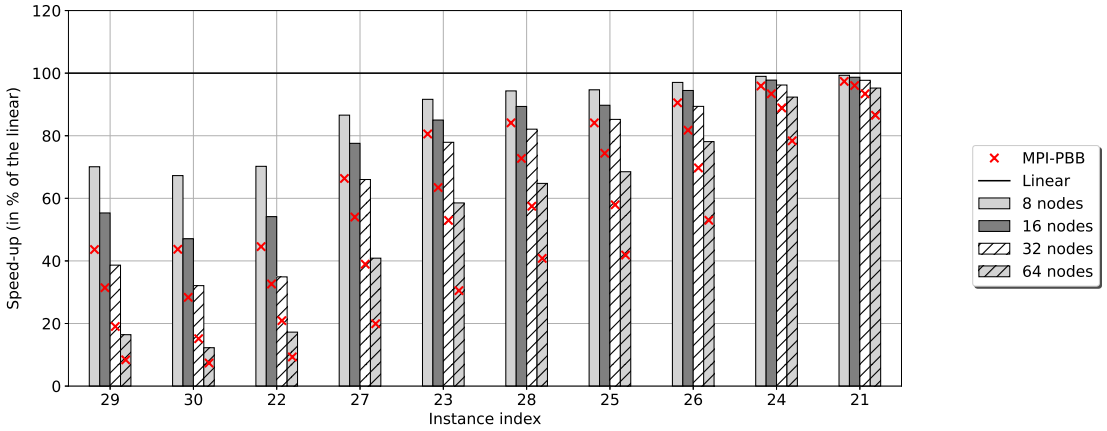


Figure 3.9: Speed-up achieved by `P3D-DFS` and `MPI-PBB` on 2 (256 cores) to 64 compute nodes (8,192 cores) compared to the execution on one node. Values are given in percent of the linear speed-up. For a given configuration, the speed-up achieved by `MPI-PBB` is presented through an $\times$ red mark. Instances are sorted by the number of nodes.

Table 3.3: Summary of execution statistics solving hard benchmark instances (50×20).

Instance	# CPU	Time (s)	CPU-hour	# nodes (10^9)	Optimum
`ta056`	800	18.1	4.0	173.3	3,679
`ta052`	128	7,960.5	283.0	17,117.8	3,699
`ta057`	800	2,017.6	448.3	28,340.7	3,704
`ta053`	128	43,605.5	1,550.4	94,885.1	3,640

latter configuration, 51,200 processing cores are used, and therefore as many segments are maintained in parallel by the `distBag_DFS` data structure. This leads to a large number of potentially remote communications, due to dynamic load balancing, which can explain the limit in performance scalability.

In this second series of experiments, we confirm the optimality of the solutions reported in [Gmy22] for some of the very hard 50×20 PFSP instances. Table 3.3 summarizes the obtained execution statistics. For the biggest instance (`ta053`), more than 12 hours of computation are required to prove the optimality of the given solution using 128 CPUs. This represents more than 11 years of computation using a serial execution on a single processing core. In contrast, the above reference used 128 GPUs during 8 hours and also exhibited a tree composed of 95×10^{12} nodes. This means that our CPU-based approach is able to cope with large-scale and to solve hard PFSP instances, which were previously solved using hundreds of GPU accelerators.

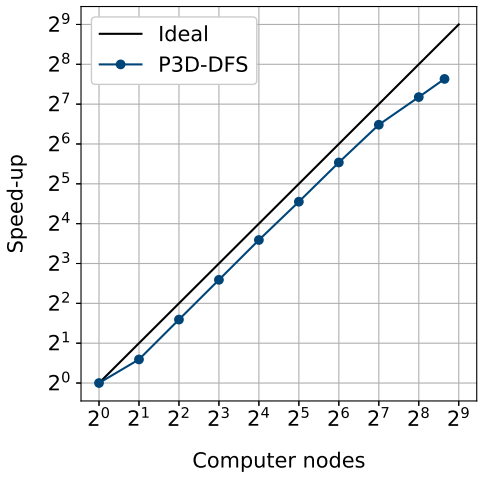


Figure 3.10: Speed-up achieved solving `ta056`, compared to a multi-core version.

3.4 Conclusion

In this chapter, we have revisited the design and implementation of PGAS-based parallel B&B algorithms for CPU-based clusters. We have presented a parallel B&B algorithm (`P3D-DFS`) based on an innovative PGAS-based data structure, called `distBag_DFS`.

`distBag_DFS` is a parallel-safe multi-pool data structure dedicated to depth-first exploration of large, irregular trees. It integrates a dynamic load-balancing mechanism based on large-scale WS, operating at both intra- and inter-node levels. This mechanism, which required sophisticated synchronization based on non-blocking split deques, promotes locality in WS, enabling scalability. The second contribution of this chapter is the design and implementation of `P3D-DFS`, a PGAS-based B&B algorithms for CPU-based clusters. The algorithm is generic with regards to the tackled optimization problem, and has been tested on three different challenging problems: PFSP, 0/1-Knapsack, and N-Queens. The PGAS-based design of `P3D-DFS` has been compared to a state-of-the-art MPI+X counterpart implementation from the literature, in terms of performance. Large-scale experimentation on a petascale supercomputer are also reported.

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

- The DFS scheduling policy of `distBag_DFS`'s segments allows to effectively control memory usage despite its parallel and unpredictable nature. This results in a bounded memory consumption on the tested permutation-based problem instances, in contrast to the existing PGAS-based `distBag` data structure.

- The `distBag_DFS`'s dynamic load balancing mechanism achieved good performance, as well as a fair workload distribution between all the allocated resources, solving fine-grained UTS instances. The mechanism takes advantage of the fact that the shallowest nodes are stolen first, and that those nodes generally have a higher

branching factor than the others. Nevertheless, it is observed that the performance may be impacted when it is not the case, like exploring binary trees.

- `P3D-DFS` achieves strong scaling efficiencies of up to 97% at the intra-node level and 94% at the inter-node level, with the best performance observed for the PFSP problem. However, the 0/1-Knapsack and N-Queens problems show limited efficiencies, highlighting the challenges posed by load balancing and fine-grained computations that introduce significant overhead.

- The PGAS-based `P3D-DFS` algorithm outperforms a state-of-the-art MPI+X counterpart implementation (`MPI-PBB`) across most configurations, achieving up to 65% faster execution for smaller PFSP instances, while maintaining better scalability with speed-ups of up to 95% on 64 nodes. In contrast, `MPI-PBB` suffers from bottlenecks due to its centralized master process, especially as instance sizes increase and workload distribution becomes less efficient.

- We demonstrated the ability of `P3D-DFS` to cope with large-scale solving hard PFSP instances on a petascale supercomputer. Using up to 400 compute nodes, or 51,200 CPU cores, we confirmed in few hours the optimality of solutions for some large instances, which would have taken several years in sequential.

Chapter 4

PGAS-based Parallel B&B for GPU-powered Clusters

Contents

In this chapter, we extend the design and implementation of the PGAS-based parallel B&B algorithm to deal with GPU-powered heterogeneous architectures. In this context, a key challenge is the dynamic handling of irregular workloads on GPUs having a SIMD architecture, as well as ensuring portability across different platforms, which is often constrained by vendor-specific APIs (*e.g.*, CUDA for NVIDIA). Additionally, GPU computing within the PGAS paradigm is still in its early stages, necessitating further exploration and development.

Section 4.1 presents the approach used to leverage GPU accelerators. The goal is to evaluate the bounds in parallel on the GPU, which is particularly compute-intensive, while the tree exploration is performed on the CPU. Then, Section 4.2 outlines the design and implementation of the PGAS-based GPU-accelerated parallel B&B algorithm. It specifically describes how the CPU and GPU interact with each other and discusses aspects related to dynamic load balancing.

Section 4.3 contains the experimental evaluations, investigating code performance and portability along with strong scaling efficiency considering both intra- and inter-node levels. Finally, Section 4.4 draws the conclusion of this chapter.

4.1 GPU-acceleration of the bounding operator

It is now acknowledged that the most compute-intensive part in a B&B algorithm is often the bounding operation. For instance, it has been shown that 98% of the total execution time is dedicated to the bounding operation in solving large PFSP instances [MCB14]. Therefore, a common way to accelerate a B&B algorithm is to offload the bounding operation to one or several GPU (s). This approach holds promise, provided that the node evaluation aligns with the GPU's execution model. Since offloading involves data preparation and transfers, the bounding operation must be significantly accelerated to offset the associated overhead. Following this strategy, two parallel models are possible: the parallel evaluation of a bound and the parallel evaluation of bounds. In this thesis, the latter one is investigated for its genericity.

Figure 4.1 illustrates the overall parallel model of our GPU-accelerated B&B algorithm. It combines the parallel tree exploration on CPU with the parallel evaluation of bounds on GPU. More specifically, several compute nodes are involved in parallel, each equipped of multi-core CPUs and many-core GPUs. For a sake of simplicity, the figure assumes that all compute nodes have the same amount of processing cores. Different interactions between the CPU and GPU occur during execution. First, the CPU is responsible for preparing and offloading the subproblems to be evaluated on the GPU. The amount of subproblems to be sent will be discussed in the next section. Indeed, in order to accelerate the bounding operation effectively, the number of offloaded subproblems must be sufficiently high to guarantee full utilization of the GPU, but not too high to avoid saturation and significant communication overhead. Then, the GPU launches kernel functions in parallel and returns the results to the CPU. The CPU can then proceed with the branching, selection, and pruning operators. In this scheme, the CPU thread needs to wait for the completion of the GPU kernel since it produces lower bounds which are needed for the pruning operation. Additionally, the CPU is responsible for managing the load balance among available threads.

4.2 PGAS-based GPU-accelerated parallel B&B

4.2.1 Overall design

Figure 4.2 shows the design of our PGAS-based GPU-accelerated parallel B&B algorithm. It is assumed that compute nodes are homogeneous, meaning that they all of them have the same amount and power of computing resources.

At the intra-node level, the algorithm maintains a multi-pool of tree nodes. More precisely, each GPU device is mapped to a CPU core host managing a pool of tree nodes. In that configuration, the CPU manages all the algorithm's dynamic aspects, such as

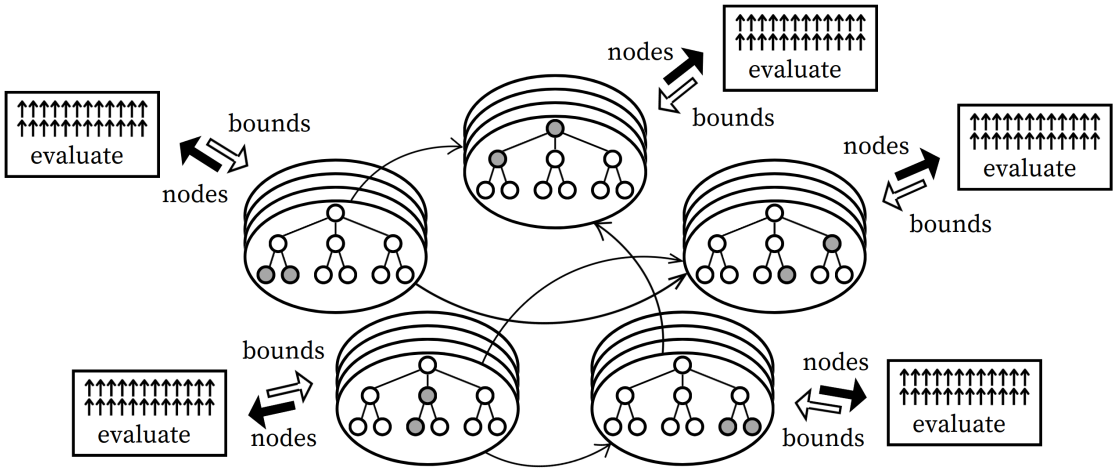


Figure 4.1: Parallel model of the GPU-accelerated multi-core B&B.

data structure management and irregular workload. In addition, GPU devices are kept independent from each other, with potential communication and collective operations performed by the hosts. Based on these pools, the tree exploration takes place in parallel on the hosts: (1) a tree node is first get from the pool; (2) this tree node is then evaluated using the problem-specific objective function; (3) depending on the evaluation it is either branched or pruned; and (4) the previous steps are repeated until the global termination of the algorithm is triggered. In this algorithm, the global termination detection algorithm follows the approach implemented in `P3D-DFS`. In practice, the pool size Q increases as the iterations progress, leading to CPU over-occupancy. At this point, GPU devices are used to accelerate the evaluation of pending tree nodes, performing this massively in parallel. Specifically, when a pool contains at least m tree nodes, a chunk of $q = \max(Q, M)$ tree nodes is transferred from the host to the device. Then, the device performs the evaluation of all tree nodes in parallel and sends back the results to the host. The latter is now able to proceed with the tree exploration, applying branching or pruning. The m and M parameters are used to determine the minimum number of tree nodes required for efficient GPU processing and to set the maximum chunk size for data transfer, respectively. This ensures optimal use of GPU resources and minimizes communication overhead. Good values for m and M should be determined experimentally.

At the inter-node parallel level, we assume that compute nodes are interconnected though a high-performance network. The latter enables remote data exchange, which is essential for the design of efficient load balancing mechanisms. Indeed, the performance of such a multi-pool algorithm targeting irregular applications lies in the load balancing between local and remote pools.

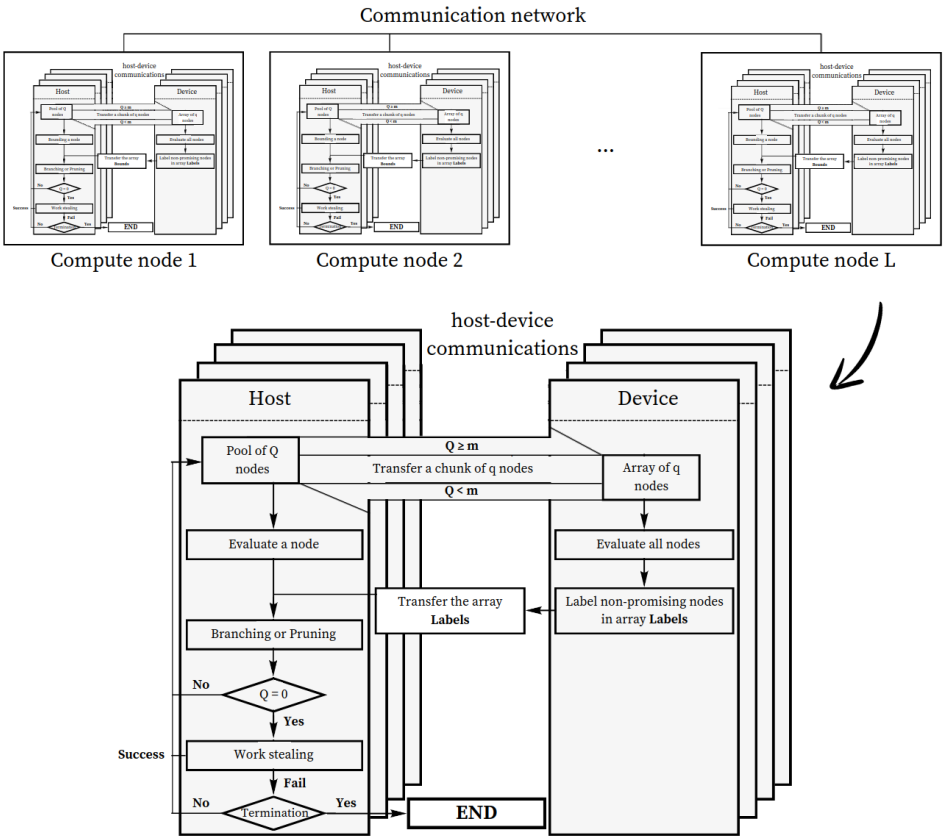


Figure 4.2: Flowchart of the GPU-accelerated multi-core B&B algorithm.

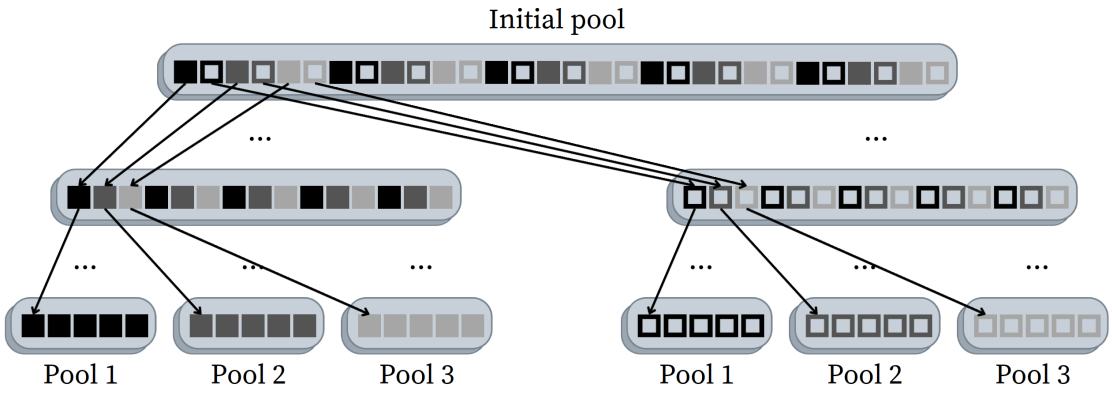


Figure 4.3: Illustration of the static workload distribution, assuming that 2 compute nodes are used, each with 3 GPU devices, and $m = 5$.

4.2.2 Load balancing mechanisms

Two complementary load balancing mechanisms are proposed to guarantee the full utilization of available resources: (1) a static workload distribution that occurs before the parallel execution and after an initial partial search, and (2) a locality-aware WS mechanism dynamically triggered during parallel execution, based on the one implemented in `distBag_DFS`.

4.2.2.1 Initial search and static workload distribution

Preliminary to the parallel tree exploration, an initial partial exploration of the tree is performed sequentially on the CPU, aiming at providing a sufficiently large number of tree nodes to each pool to prevent GPU starvation at the beginning of the parallel search. Indeed, GPU devices are composed of thousands of cores, which require a substantial number of tree nodes to maintain high utilization and efficiency. The initial search is performed in a BFS manner, allowing for a broad and uniform distribution of tree nodes, and ends when $m \times numGPUs \times numComputeNodes$ pending tree nodes are in the pool.

At the end of the initial search, the workload is evenly spread among pools in a cyclic fashion, as shown in Figure 4.3. More precisely, two cyclic distributions occur successively. The first one distributes all the workload evenly among all the available localities (compute nodes), while the second one distributes all the workload on each locality evenly among the available processing threads. Although the mapping is static, it is preferable in practice to perform two successive distributions. Indeed, on a distributed architecture with NUMA, having a single master thread communicate with each other thread of every compute node to transfer elements would generate higher communication overheads.

4.2.2.2 Dynamic Work Stealing

A WS mechanism is dynamically triggered during parallel execution when a pool becomes empty. This mechanism is composed of three main components: (1) a victim selection strategy, (2) a work sharing policy, and (3) a synchronization mechanism. Each of these is discussed in the following.

- *Victim selection strategy*: The victim selection strategy determines how victim pools are chosen among all available ones. The latter mechanism is based on the random victim selection policy iterator introduced in previous chapter. It assumes that a victim thread is selected at random when a thread initiates a WS operation. As most of distributed systems have a NUMA architecture, threads are not selected uniformly as the cost of accessing a remote thread is higher compared to a local one. Therefore, the victim selection strategy exploits locality-awareness. When a thread becomes empty during execution and triggers a WS operation, it first selects uniformly a victim thread from its own locality. Indeed, at the intra-node level, memory access can be considered as uniform. If all WS attempts fail, the calling thread searches at the inter-node level from another locality, also selected randomly.

- *Work sharing policy*: The work sharing policy controls how many tree nodes are stolen from the victim pool. It is an important aspect for performance as stealing only few tree nodes may not be sufficient to balance the workload, while stealing too many tree nodes may create load imbalance implicitly. In addition, any WS operation implies a potentially remote data exchange, and it is often not worth to exchange only few tree nodes regarding the communication overhead. In this algorithm, the thief thread steals half of the available tree nodes, only if the victim pool size is larger than $2 \times m$. This work sharing policy guarantees that at least m nodes remain in both thief and victim pools after a stealing operation, allowing threads to proceed with the exploration.

- *Synchronization mechanism*: The synchronization mechanism determines how thief and victim threads coordinate themselves to ensure the parallel-safety of pools. In this algorithm, we adopt a spin-lock synchronization mechanism. When a thread tries to acquire a spin-lock, it continuously checks (or "spins") to see if the lock is available, rather than yielding control or putting itself to sleep. This is achieved by repeatedly polling a flag or memory location until the lock becomes free. Spin-locks are typically lightweight and provide low-latency access to locks, making them suitable for short critical sections where the lock is held briefly. However, they can cause high CPU usage if contention is significant, as waiting threads consume processor cycles while spinning. Spin-locks are often used in real-time or low-latency systems and are generally more efficient on multiprocessor systems, where other threads can make progress without blocking the processor.

4.3 Experiments

This section evaluates the performance and portability of our algorithms. Section 4.3.1 describes the experimental protocol and testbed. Section 4.3.2 provides an extensive experimentation on code performance and portability considering six different GPU architectures, including different generations and vendors. Section 4.3.3 contains preliminary experiments for the parameter calibration. Finally, Section 4.3.4 presents the strong scaling efficiency of our algorithm, considering both intra- and inter-node parallel levels.

4.3.1 Experimental protocol and testbed

The proposed PGAS-based algorithm has been developed in three distinct implementations, each optimized for execution on a single-GPU, single-node multi-GPU, and multi-node multi-GPU configurations, respectively. In addition, we also implemented low-level counterpart implementations based on CUDA, following as close as possible the same design. This substantial effort seeks to address the lack of optimized low-level open-source implementations of GPU-accelerated B&B algorithms in the literature. These baselines will serve as a reference for evaluating the parallel efficiency of our PGAS-based approach.

The experiments are conducted on the HPE Cray EX LUMI European pre-exascale supercomputer[7]. The compute nodes are equipped with a single 64-core AMD EPYC 7A53 "Trento" CPU, four AMD Instinct MI250X GPUs, and are connected to a HPE Cray Slingshot-11 200 Gb/s network interconnect. A MI250X is a multi-chip module with two GPU dies, each featuring 110 compute units (CU) and 64 GB slice of HBM2e memory for a total of 220 CUs and 128 GB memory per MI250X module. From a software perspective, one MI250X module is considered as two GPUs, meaning that nodes can be considered as 8-GPU compute nodes. The portability of CUDA code to AMD GPU architectures is handled by the `hipify-perl` tool, which automatically translates CUDA code into portable HIP code.

In the following, the algorithms are tested on the PFSP and N-Queens problems. Instances ranging from $N = 15$ to 19 are considered for the latter problem, while the class of 20×20 instances is considered for the PFSP.

4.3.2 Code performance and portability

GPU accelerators vary significantly across architectures and vendors in terms of performance, memory capacity, and parallelism capabilities. Consequently, in this section we evaluate the performance and portability of our PGAS-based GPU-accelerated algorithm. Specifically, in addition to the AMD MI250X accelerators, the following GPU architectures are also considered:

- *NVIDIA P100*: 12-core Intel Xeon Gold 6126 (Skylake-SP) @ 2.60GHz CPU, equipped with an NVIDIA Tesla P100 PCIE 16GB GPU (3,584 cores, released in

[7]LUMI is ranked 5^{th} in June 2024 TOP500 ranking.

June 2016);

- *NVIDIA V100*: 12-core Intel Xeon Gold 6126 (Skylake-SP) @ 2.60 GHz CPU, equipped with an NVIDIA Tesla V100 PCIE 32GB GPU (5,120 cores, released in March 2018);
- *NVIDIA A100*: 32-core AMD EPYC 7513 (Zen 3) @ 2.60 GHz CPU, equipped with an NVIDIA A100 SXM4 40GB GPU (6,912 cores, released in May 2020);
- *AMD MI50*: 48-core AMD EPYC 7642 (Zen 2) @ 2.40 GHz CPU, equipped with an AMD Radeon Instinct MI50 32GB GPU (3,840 cores, released in November 2018);
- *AMD MI300X*: 64-core AMD EPYC 7A53 "Trento" (Zen 3) @ 2.0 GHz CPU, equipped with an AMD Instinct MI300X GPU (19,456 cores, released in June 2023).

Figure 4.4 shows the normalized execution time of the Chapel algorithm compared to the CUDA/HIP baseline, solving instances of the PFSP and N-Queens problems. Single-GPU experiments are conducted to isolate and evaluate the core performance on individual GPUs without the added complexity of multi-GPU coordination or communication overhead. It is important to observe that no direct comparison can be made between the performance and capabilities of the different GPU architectures.

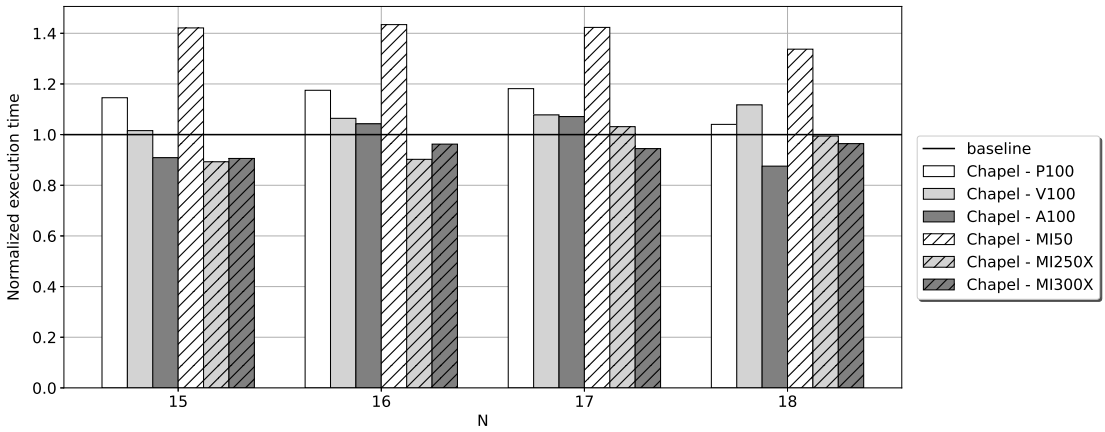
Solving the N-Queens instances, one can see that the Chapel algorithm is between 87% faster to 43% slower than the baselines. Actually, it can be observed that the performance gap strongly depends on the GPU architecture. Considering NVIDIA GPUs, the performance improves with more modern architectures. Specifically, the Chapel-based algorithm is on average 13% slower on the NVIDIA P100 (released in 2016), 7% slower on the NVIDIA V100 (released in 2018), and 3% faster on the NVIDIA A100 (released in 2020). This is largely due to the fact that modern architectures offer greater capabilities in terms of core count, memory storage and bandwidth, *etc.* Furthermore, optimizing compiler performance on older GPU architectures, such as the NVIDIA P100, is of limited interest to language developers targeting exascale, like Chapel, as these architectures are no longer (or only minimally) used in modern systems [TOP24]. Similar observations are done considering the AMD GPUs, where the best average performance are achieved using the AMD MI300X GPU.

The experiments on PFSP instances reveal similar behaviors, but the performance gaps between the PGAS approach and the low-level baselines are now much more pronounced. For example, the Chapel approach is on average 62% slower than the baseline on the AMD MI250X GPU, whereas it was 4% faster on the N-Queens problem. A key difference between the two problems lies in the bounding function of the B&B algorithm. For the N-Queens problem, the bounding function is relatively simple, relying only on the depth of the tree node and the permutation that represents the associated subproblem to determine satisfiability. In contrast, the PFSP requires a more complex bounding function, as it must account for cumulative processing times across multiple machines

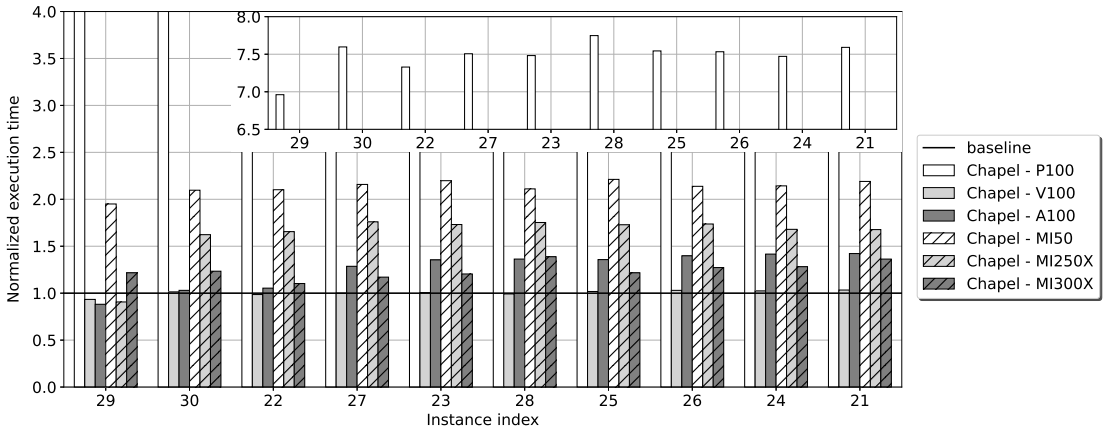
and jobs to compute the lower bound. Storing data, such as constant matrices, on the GPU device reduces memory transfer overhead and provides faster access during computation. However, this may increase the demands on memory usage and management, especially on GPUs with limited memory capacity, leading to a severe degradation in performance.

In investigating the performance gap between Chapel and baseline codes, we detected that Loop-Invariant Code Motion (LICM) may be thrown off by the Chapel compiler in a few specific configurations. LICM is a compiler optimization that performs automatically the movement of statements or expressions that can be moved outside the body of a loop without affecting the semantics of the program. In programs implemented using high-level languages, such as Chapel, LICM typically occurs when arrays, which are not merely simple arrays but also embed additional data or metadata, are used within loops. For instance, in contrast to C arrays that are essentially contiguous blocks of memory that store elements of a specific data type, Chapel arrays also include metadata such as their size, domain (index space), distribution, *etc.* As a result, accessing redundant metadata repeatedly within the loop may result in performance degradation in the absence of LICM, compared to simpler baseline codes where such operations are either absent or easily optimized away. Algorithm 4.1 illustrates the LICM optimization that is usually performed by compilers to deal with this issue. It consists in isolating array pointers towards contiguous blocks of memory in dedicated variables (here *Aptr* and *Bptr*) and using the latter inside the loop, thus avoiding any metadata access. In practice, we observed that the Chapel compiler may fail LICM in certain scenarios, thus degrading performance especially solving PFSP instances, where the problem data is stored in constant arrays on the device. Although it is not possible to identify the root cause of the non-triggering of the optimization in the Chapel compiler without a low-level analysis of it (which goes beyond the scope of this thesis), doing LICM manually in inner-most parts of our code allows to reduce the execution time from 10% on the P100 GPU to 26% on the MI50 GPU.

LICM is not the only factor that can explain the difference in performance between Chapel and baseline counterparts. Related works investigating the performance portability of Chapel in a different context also reported that the Chapel compiler may fail to take advantage of register coalescing optimizations provided by LLVM, which are available to the GPU back-end code generation [MWA24]. Register coalescing is another compiler optimization technique aimed at improving the efficiency of register usage by reducing the number of move instructions between registers. This optimization identifies situations where multiple registers hold values that could be stored in a single register without affecting the program's correctness, and then "coalesces" (merges) them into one register, eliminating unnecessary data transfers between registers. Failing this optimization may result in a higher register pressure and, thus, lower warp occupancy, reducing parallel execution.



(a) N-Queens problem instances.



(b) PFSP instances.

Figure 4.4: Normalized execution time of the single-GPU Chapel code solving different problem instances on different NVIDIA and AMD GPU architectures, compared to CUDA/HIP-based counterpart implementations.

Algorithm 4.1: Example of LICM optimization in high-level languages.

input : n: number of iterations;
A: array of constant data;
B: array of constant data;

```
1 Aptr ← constPtr(A[0]);
2 Bptr ← constPtr(B[0]);
3 for i from 0 to n do
      // arithmetic operations, accessing Aptr and Bptr
```

Table 4.1: Calibration of (m, M) parameters on AMD MI250X, executing the Chapel single-GPU B&B algorithm for different values of (m, M) and solving the `ta030` PFSP instance. The execution times are in second; whiter is better.

m \ M	50,000	100,000	200,000	300,000	400,000	500,000	600,000
10	9.78	9.55	9.41	9.42	9.36	9.36	9.42
20	9.78	9.52	9.4	9.39	9.38	9.36	9.38
30	9.78	9.56	9.41	9.4	9.36	9.36	9.38
40	9.78	9.56	9.4	9.38	9.37	9.37	9.37
50	9.78	9.55	9.41	9.38	9.37	9.36	9.36
60	9.79	9.56	9.4	9.38	9.37	9.36	9.36
70	9.78	9.56	9.42	9.4	9.36	9.36	9.37
80	9.78	9.55	9.41	9.38	9.36	9.37	9.38
90	9.79	9.56	9.41	9.37	9.37	9.36	9.38
100	9.8	9.56	9.41	9.39	9.36	9.36	9.39

4.3.3 Parameter calibration

The experiments presented in this section aim to determine the couple of parameters (m, M) that optimize the execution of the proposed B&B algorithm on AMD MI250X. m determines the minimal workload necessary for efficient GPU processing. If set too low, the overhead of data transfer can negate the advantages of parallel computation, leading to sub-optimal performance. Conversely, an appropriately chosen m allows the GPU to handle a sufficient volume of data, maximizing throughput and reducing idle time. On the other hand, M influences the efficiency of data transfers. A chunk size that is too large may exceed memory limits or result in long transfer times, while a size that is too small can create excessive overhead from frequent transfers. Carefully tuning both m and M is needed to balance between efficient resource utilization and processing speed, enhancing overall execution times and performance on the GPU.

Table 4.1 shows the execution time of the single-GPU B&B algorithm solving the `ta030` PFSP instance on AMD MI250X, and varying the parameters m and M. One can first see that for almost all values of m, setting $M = 500,000$ allows to minimize the execution time. Moreover, in the range of the optimal value for M (*i.e.*, $M \in [400,000; 600,000]$) setting $m \in [50; 60]$ seems to provide the best results. We can also note that the worst-case time obtained on this instance is about 5% longer than the optimal one. The calibration of (m, M) has been repeated for several PFSP and N-Queens instances. Most of the time, setting m and M to 50 and 500,000, respectively, provides good results. These values are therefore considered in the following.

4.3.4 Strong scaling efficiency

Table 4.2 shows the absolute strong scaling efficiency achieved by our PGAS-based GPU-accelerated B&B algorithms considering both intra- and inter-node parallel levels. At the intra-node (resp. inter-node) level, the single-node (resp. multi-node) multi-GPU

B&B algorithm is tested, and compared to the optimized single-GPU (resp. single-node multi-GPU) version.

At the intra-node level, we achieve on average 87% of the ideal speed-up solving the N-Queens instances using 2 GPUs, and 40% using 8 GPUs. As the number of GPUs increases, the coordination and data exchange associated with intra-node load balancing produce overhead that becomes progressively more significant. Since the N-Queens problem is fine-grained, the computation cost per subproblem is insufficient to offset these overheads, which limits scalability. Solving the PFSP instances, the observed speed-up is very low, reflecting the fact that the single-GPU Chapel version of our algorithm is, on average, 62% slower than the HIP-based baseline, as shown in Figure 4.4b. However, it is theoretically possible to determine the relative speed-up of our algorithm. In general, if we assume that $t_{chpl} = x \times t_{hip}$ with $x \geq 1$, then the following relationship between absolute and relative speed-up holds: $s_{relative} = x \times s_{absolute}$. This means that, on average, our Chapel approach achieves a relative speed-up of $1.62 \times 0.89 = 1.44$ using 2 GPUs, and $1.62 \times 2.39 = 3.87$ with 8 GPUs. For the best relative results, 73% of the ideal speed-up is achieved solving the `ta024` instance using 8 GPUs.

At the inter-node level, a similar trend appears. Scalability for solving large N-Queens instances is limited as the number of compute nodes increases, achieving only 25% of the ideal speed-up on average with 128 compute nodes, corresponding to roughly 3 billion nodes processed per second. For the PFSP, a relative speed-up of 44% is observed with 32 compute nodes, but this drops to less than 19% with 128 compute nodes. This limitation arises from the synchronization approach of the dynamic WS mechanism, which relies on spin-locks. When hundreds of GPUs are used, this approach maintains numerous pools and threads, leading to substantial overhead and performance degradation at scale. However, this effect is less severe for the N-Queens problem, as the problem generates relatively regular trees, allowing the initial static load-balancing mechanism to distribute the workload more effectively among nodes. This reduces WS and, consequently, synchronization requirements.

4.4 Conclusion

In this chapter, we have extended the design and implementation of our PGAS-based parallel B&B algorithm to deal with GPU-powered heterogeneous architectures.

The algorithm combines the parallel tree exploration on CPU with the parallel evaluation of bounds on GPU to accelerate the compute-intense bounding operator of the B&B algorithm. The workload irregularity is managed by a multi-level dynamic load-balancing mechanism, inspired by the one of `distBag_DFS`, adapted to the context of GPU. From the implementation aspect, the use of Chapel allows high-level abstractions that seamlessly integrate multiple levels of parallelism—CPU, GPU, and inter-node—within a unified programming language. In addition, the portability challenge is addressed by the vendor-neutral GPU-support of the language.

The algorithm is generic with regards to the tackled optimization problem, and has been tested on two different challenging problems: PFSP and N-Queens. For compari-

Table 4.2: Strong scaling efficiency achieved by the GPU-accelerated B&B considering both intra- and inter-node levels. Instances are sorted by the number of nodes.

(a) Intra-node level (single-node multi-GPU configuration).

Instance	GPU×1	GPU×2		GPU×4		GPU×8	
	kn/s	*kn/s*	speed-up	*kn/s*	speed-up	*kn/s*	speed-up
15-Queens	34,744.0	57,970.7	1.67	82,106.1	2.36	88,475.6	2.54
16-Queens	34,884.4	60,163.2	1.72	95,487.7	2.74	119,120.2	3.41
17-Queens	34,940.7	64,458.7	1.84	93,966.3	2.69	124,720.3	3.57
AVG	**34,856.4**	**60,864.2**	**1.74**	**90,553.4**	**2.60**	**110,768.8**	**3.17**
ta029	2,036.3	1,546.3	0.76	2,222.7	1.09	2,602.0	1.28
ta030	2,134.5	1,733.6	0.81	2,356.0	1.10	2,641.1	1.24
ta022	2,126.4	1,680.6	0.79	2,435.4	1.15	2,699.0	1.27
ta027	2,217.6	1,980.8	0.89	3,319.3	1.50	4,790.8	2.16
ta023	2,256.0	2,068.4	0.92	3,503.1	1.55	5,817.1	2.58
ta028	2,216.0	2,092.4	0.94	3,569.3	1.61	5,993.0	2.70
ta025	2,278.9	2,102.2	0.92	3,673.6	1.61	6,468.7	2.84
ta026	2,228.3	2,166.3	0.97	3,814.3	1.71	6,745.3	3.03
ta024	2,233.3	2,167.5	0.97	3,973.0	1.78	8,086.0	3.62
ta021	2,288.4	2,175.4	0.95	3,992.2	1.74	7,170.0	3.13
AVG	**2,201.6**	**1,971.4**	**0.89**	**3,285.9**	**1.48**	**5,301.3**	**2.39**

(b) Inter-node level (multi-node multi-GPU configuration).

Instance	node×1	node×8		node×16		node×32		node×64		node×128	
	kn/s	*kn/s*	speed-up	*kn/s*	speed-up	*kn/s*	speed-up	*kn/s*	speed-up	*kn/s*	speed-up
17-Queens	124,720.3	241,356.8	1.94	408,788.8	3.28	708,250.0	5.68	1,288,628.5	10.33	2,362,483.6	18.94
18-Queens	80,571.4	284,731.0	3.53	526,908.5	6.54	914,021.0	11.34	1,520,257.5	18.87	2,913,535.2	36.16
19-Queens	79,308.8	286,321.1	3.61	631,282.6	7.96	1,205,458.2	15.20	2,029,609.9	25.59	3,210,848.4	40.49
AVG	**94,866.8**	**270,803.0**	**3.03**	**522,326.6**	**5.93**	**942,576.4**	**10.74**	**1,612,832.0**	**18.26**	**2,828,955.7**	**31.86**
ta028	5,993.0	15,812.0	2.64	26,172.8	4.37	38,768.2	6.47	44,693.5	7.46	40,969.0	6.84
ta025	6,468.7	17,583.4	2.79	30,628.0	4.86	42,100.1	6.69	58,218.3	9.25	63,865.0	10.15
ta026	6,745.3	18,642.2	2.76	33,022.3	4.89	48,921.9	7.25	64,821.5	9.60	93,878.2	13.92
ta024	8,086.0	22,668.0	2.81	40,293.7	5.01	68,731.1	8.54	100,030.1	12.44	141,005.3	17.54
ta021	7,170.0	20,394.6	2.84	37,642.5	5.25	67,637.6	9.43	93,249.8	13.00	131,768.6	18.37
AVG	**6,892.6**	**19,020.0**	**2.77**	**33,551.8**	**4.88**	**53,231.7**	**7.68**	**72,202.6**	**10.35**	**94,297.2**	**13.36**

son purpose, CUDA-based counterpart implementation has also been implemented. We investigated the performance and portability of the algorithms on several GPU architecture, and also performed large-scale experimentation on a pre-exascale supercomputer.

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

- Chapel's performance varies significantly across problems and GPU architectures. For instance, it outperforms the baseline by 3% on the NVIDIA A100 for the N-Queens problem but lags 13% behind on the older P100. For the PFSP, the performance gap is even more pronounced, primarily due to the complex bounding function that increases memory demands and limits performance.
- The experimental results show that missed compiler optimizations in Chapel, including LICM and register coalescing, contribute to performance gaps. Manual LICM improves execution by up to 26% on certain GPUs.
- The experiments demonstrated that the tuning of the m and M parameters in our B&B algorithm may have an impact on overall performance. On the AMD MI250X, $m \in [50, 60]$ and $M = 500,000$ consistently provide good results across various N-Queens and PFSP instances.
- The strong scaling efficiency analysis shows that our approach achieves up to 40% speed-up on the N-Queens problem with 8 GPUs at the intra-node level, but only 25% at the inter-node level with 128 nodes. Deploying up to 1,024 GPUs in parallel proves limited for fine-grained problems, as the node evaluation cost fails to offset synchronization overhead.
- For the coarse-grained PFSP problem, relative speed-ups of up to 73% were achieved at the intra-node level using 8 GPUs for the largest instances. However, at the inter-node level, large-scale performance is constrained by the overhead from spin-lock-based thread synchronization.

Chapter 5

A Chapel Software Platform for PGAS-based Parallel B&B

Contents

The algorithms presented in this thesis are characterized by their genericity with regards to the problem being solved. This chapter introduces the resulting Chapel-based software platform for PGAS-based parallel B&B algorithms, referred to as `pBB-chpl`, enabling users to assess the feasibility of reusing the platform to solve new problems.

Section 5.1 motivates the need for software platform in the context of scientific research and raises the challenges to achieve a scalable design. The design scalability is defined as the ability of a computer program to be modified, to be expanded and to cope with increased use [RXX11]. To rise to these challenges, existing tools are underlined for the convenient features they provide regarding code abstraction, utility, extensibility and accessibility. The Object-Oriented Programming (OOP) paradigm and the Chapel programming language are notably put forward.

Section 5.2 presents the `DistributedBag` module, implemented within the context of `pBB-chpl`, encapsulating the `distBag_DFS` data structure presented in Chapter 3. This module has been integrated into the Chapel language, enabling users to easily

import the data structure into their own code for various applications, independently from `pBB-chpl`. Aspects relative to the data structure capabilities and tuning options are discussed.

Then, Section 5.3 details the `pBB-chpl` software platform, which contains the skeletons of the parallel B&B algorithms presented in Chapter 3 and Chapter 4. The modular and flexible code structure of the platform is presented, along with a description of the skeletons and their target architectures, including multi-core desktops and laptops, commodity clusters, in addition to the high-end supercomputers. Finally, Section 5.4 draws the conclusion of this chapter.

The implementation and documentation of the `DistributedBag` module are provided in the open-source code repository and online documentation of the Chapel programming language, available at `https://github.com/chapel-lang/chapel` and `https://chapel-lang.org/docs/`, respectively. In addition, the parallel B&B skeletons targeting CPU-based and GPU-powered clusters are hosted on the open-source code repositories `https://github.com/Guillaume-Helbecque/P3D-DFS` and `https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel`, respectively, and archived on Zenodo (DOIs 10.5281/zenodo.7328540 and 10.5281/zenodo.10786275, respectively).

5.1 Scalable code development

This section first presents the different aspects motivating the adoption of a scalable code architecture. Then, the main conceptual objectives of our platform are defined, and the tools used to achieve these objectives are discussed.

5.1.1 Motivations

Reducing the costs of software development
One of the primary objectives of this thesis is to promote software productivity at various levels. While we have designed and implemented PGAS-based parallel B&B algorithms that unify the different levels of parallelism through high-level abstractions, a second objective is to make our algorithms available to the community and enable their extensibility to many other problems. Indeed, the costs associated with software development and maintenance should not be entirely overshadowed by computational costs. As argued in [RXX11], the time and resources required for software development often exceed those of computation. The authors recommend structuring 80% of the code to facilitate the software development, while focusing on optimizing the remaining 20% that bears the greatest computational load to reduce runtime.

Targeting as wider community as possible
The dissemination of parallel B&B requires software tools that are intuitive, extensible, and customizable, allowing a wide scientific community to leverage them effectively. Ensuring compatibility across computational environments, from desktops to supercomputers, reduces barriers for non-expert users and promotes widespread adoption and

collaboration. The platform should provide ready-to-use methods and templates, empowering researchers across disciplines to benchmark and solve complex, real-world optimization problems.

Scalable design
Since the late 20th century, designing scalable code architectures has been recognized as a task of paramount importance, on par with the development of supercomputers themselves [DNS97]. In the context of software architecture, design scalability is defined as the ability of a computer program to be modified, expanded, and to handle increased usage [RXX11]. An architecture that employs modular decomposition is expected to reduce dependencies between algorithmic components, thereby enhancing flexibility. A key condition for achieving a scalable design is also the availability of comprehensive and well-illustrated documentation, which promotes accessibility and helps mitigate the complexity associated with code evolution.

5.1.2 Conceptual objectives

Code reusability
One of the key objectives of design scalability is ensuring code reusability. Reusability is defined by [Al-+10] as "the ability to use part or the whole system [...] to reduce the effort, cost, and time to develop a new system". It have also been identified three main approaches to software re-usability: *no reuse*, *code reuse*, and *code and design reuse* [Mel05]. The motivation behind the no reuse approach is the apparent simplicity of the implemented algorithms, which encourages the developer to implement them independently. However, beyond the difficulty of maintaining the code, such an approach requires significant time and effort and is prone to errors.

The code reuse approach involves reusing standalone programs or libraries. Reusing standalone programs requires a detailed examination of the code and the rewriting of specific sections related to the problem at hand. This task is time-consuming, tedious, and prone to errors. For this reason, libraries provide a more efficient and reliable solution for reusing existing code. Additionally, libraries are often tested and documented, making their maintenance much easier. However, libraries only allow for code reuse and are not designed to facilitate the reuse of design.

Finally, the objective of the code and design reuse approach is to overcome the limitations of the previous approach by enabling both code and design reuse. In other words, it aims to minimize the amount of code developed (and thus the development effort) whenever a new optimization problem is addressed. In the context of combinatorial optimization, this approach is based on separating the invariant parts of the solution methods from the parts specific to the problems being solved. The invariant part, independent of the problems, forms a set of components that can serve as design models, and are often called "skeletons".

Utility and adaptation
The aim of the invariant part of the software platform is twofold. Firstly, it should at-

tract a wide user community by saving them time, enabling them to focus more on their specific problems. This is achieved by providing a diverse set of ready-to-use algorithmic tools (*e.g.*, problem-solving methods) that have already demonstrated their value for `pBB-chpl`. Secondly, the skeletons should provide access points for easily integrating specific code. The invariant code must be sufficiently robust to prevent significant disruptions while still offering flexibility for adaptation.

Accessibility
Accessibility is primarily ensured through the ease of use of the software platform. The process of obtaining the code, installing the necessary dependencies, and running an initial example should be simple and quick. The ease of extension is the second key aspect. It is closely tied to a clear understanding of the code structure, from the overall platform layout to the finer details at the class level. Access points for inserting custom code should be clearly identified. Finally, portability across different operating systems and hardware infrastructures is crucial to achieve accessibility. The heterogeneity of modern computational systems, particularly in terms of hierarchical memory organization, must be addressed to maximize the parallel use of computational resources.

5.1.3 Tools for scalable code architecture

Object-Oriented Programming
OOP is a paradigm that structures code around "objects", encapsulating attributes and methods within classes to promote modular, re-usable, and scalable design [Weg90]. Key principles—encapsulation, inheritance, and polymorphism—enable a clear separation of concerns and reduce complexity as the code grows. Encapsulation is a technique that combines attributes and the methods that operate on these attributes within a single unit, called an object. By making an object's internal state private and exposing only selected methods for interacting with it, encapsulation restricts direct access to sensitive information and maintains data integrity. Inheritance is a mechanism that allows a new class (subclass) to acquire properties and behaviors (attributes and methods) from an existing class (superclass), establishing a hierarchical relationship between classes. The subclass can inherit the methods and attributes of the superclass, and it can also override or extend them to provide specialized behavior. Inheritance enables the creation of more specific classes based on general ones, facilitating a structured and organized codebase while reducing redundancy. Finally, polymorphism allows objects of different classes to be treated as objects of a common superclass. Through method overriding (runtime polymorphism) or method overloading (compile-time polymorphism), polymorphism allows a method to behave differently based on the object it is called on. This promotes flexibility and extensibility, as new classes can be added without modifying existing code. Put together, these principles support modularity, and are ideal for scalable systems, where components can be independently developed, improved, and deployed, ultimately making large-scale systems more manageable, robust, and easier to extend.

Chapel programming language
Choosing one programming language over another for the implementation of a software framework is generally not a trivial decision. In the context of PGAS programming, we have opted for the Chapel programming language for multiple reasons. First of all, Chapel is a free, open-source and general-purpose language, benefiting from an active development and community, notably with an annual forum for Chapel users and developers. In addition, Chapel already powers several academic research projects, ranging from a multi-physics software oriented toward fluid dynamics (CHAMPS) to simulation of the dynamics of ultra-light dark matter for astrophysics (chplUltra)[8]. Even though Chapel is a new base language that requires some learning before it can be used effectively, it is designed to be easy to learn for users of Python, C, C++, Fortran, Java, Matlab, and the like. Indeed, Chapel allows avoiding the use of pointers, supports type inference, and high-level abstractions, allowing developers to focus on expressing their algorithms and parallelism without getting bogged down in low-level implementation details. In addition, Chapel includes OOP features for modularity, code portability through the compiler, code indentation favoring readability and vectorization alleviating the amount of code. Additionally, Chapel allows C, Fortran, and Python interoperability, enabling the re-use of existing libraries. All these assets make the Chapel programming language a good candidate for the implementation of a scalable code architecture.

Documentation and distribution
Documentation and distribution are essential aspects of software development, ensuring that the code is clear, well-organized, and easily accessible to both users and developers. Clear and thorough documentation serves as a guide to understanding the system, outlining its architecture, functionality, and user interaction. Effective distribution, meanwhile, guarantees that the software can be deployed, updated, and used effortlessly by the target audience. This includes packaging the software to simplify its installation and configuration, ensuring compatibility across various environments, and maintaining version control for managing updates and fixes. To this end, `pBB-chpl` is hosted on GitHub, a popular platform for open-source software, and is also archived on Zenodo, an archiving service that facilitates long-term preservation. Together, documentation and distribution foster seamless collaboration and usability, allowing both end-users and developers to interact with the software efficiently and confidently.

5.2 The Chapel's `DistributedBag` module

This section presents an initial effort towards designing a scalable software interface, based on the integration of the `distBag_DFS` data structure into the Chapel programming language.

[8]Projects powered by Chapel: `https://chapel-lang.org/poweredby.html`.

5.2.1 Integration of `distBag_DFS` into Chapel

Chapel provides a rich collection of modules that enable users to import the libraries they require. These modules are categorized into two groups: standard modules and package modules. Standard modules describe features that are considered part of the Chapel Standard Library, while package modules consist of libraries that exist outside the Chapel Standard Library. This separation is typically due to these libraries being either not fundamental enough or not yet sufficiently mature for inclusion in the standard library.

The `distBag_DFS` data structure introduced in Chapter 3 has been integrated into the `DistributedBag` package module since Chapel 2.0 release. This integration offers several key benefits:

- *Re-usability*: By packaging `distBag_DFS` as a module, the data structure can be reused across multiple projects and applications with minimal effort. This promotes software productivity by enabling developers to leverage existing functionalities without needing to re-implement them for each project. Furthermore, this modular approach reduces code duplication, making it easier to maintain consistent and reliable solutions across different contexts. Additionally, being part of an open-source ecosystem, developers can freely access, modify, and extend the module, fostering innovation and collaboration across a wide range of projects.

- *Maintainability*: All notable features of Chapel, including package modules, are extensively tested to ensure stability and reliability as the language evolves. This provides two major advantages. First, it ensures that the functionality of the `distBag_DFS` data structure remains consistent, even as new language features are introduced or the Chapel compiler is updated. Second, it helps to early detect performance regressions, allowing developers to address potential issues before they become widespread. Additionally, encapsulating the data structure in a module makes it easier to update or improve the implementation without introducing unintended side effects in other parts of the codebase. Any necessary changes can be made in a single location, streamlining the maintenance process.

- *Modularity*: Modularity is a core principle of the package module system. By organizing `distBag_DFS` within its own module, the structure is isolated and well-defined. This promotes clean and well-structured code, allowing developers to focus on specific components without affecting unrelated functionality.

- *Visibility*: The integration of `distBag_DFS` in a package module increases its visibility within the Chapel community, and by extension the parallel computing community. By packaging it in a module, the data structure becomes accessible to a wider audience of developers who may not be familiar with its specifics but can now easily incorporate it into their own projects. Documentation generated for the module, as illustrated in Figure 5.1, further enhances its accessibility. This visibility encourages adoption, feedback, collaboration, and contributions from the community, which in turn helps refine and improve the data structure over time.

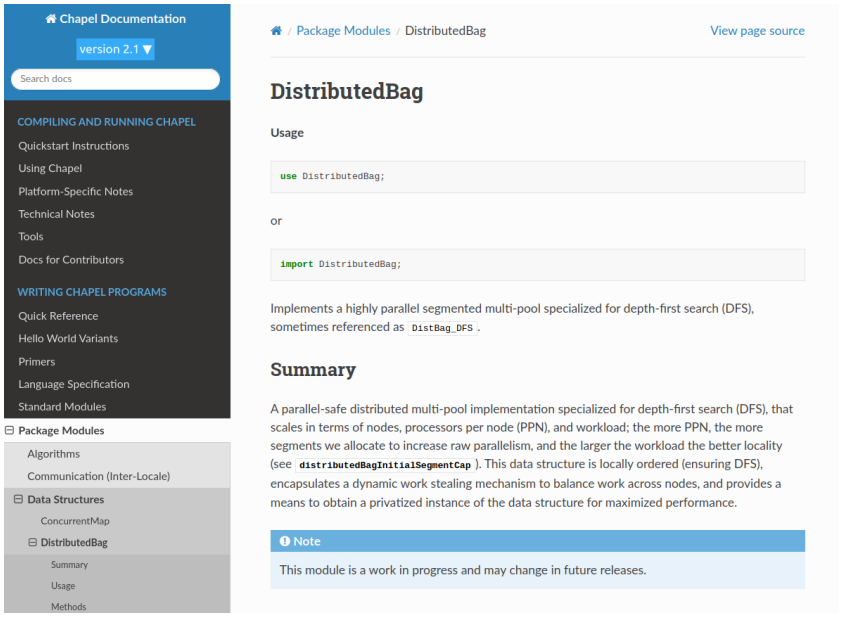


Figure 5.1: Chapel's official documentation of the `DistributedBag` package module.

In the following, we present an overview of the features of the data structure, enabling the implementation of various applications, independently from the `pBB-chpl` framework.

5.2.2 Local and global operations

`distBag_DFS` provides several operations, categorized into two types: local operations, which apply to a specific segment of an individual bag instance, and global operations, which affect the entire `distBag_DFS`.

Local operations include `add`, `addBulk`, and `remove`, and allow to insert an element, insert elements in bulk, and remove an element from the segment specified by $taskId$, respectively. Each of these operations applies to the bag instance of the locale it is called from. One of the specific features of `distBag_DFS` is that the index of the task to operate on must be explicitly given when calling the operation, ensuring DFS ordering. Alternative implementations could investigate an automated way to deal with this index, but PGAS-based languages, such as Chapel, often intentionally avoid supporting a standard language-level way to query a task's id. On the other hand, some internal opaque type exist, referring to the task id that the runtime uses. However, their usage can lead to portability issues since different runtime tasking options may exist, and the support is not guaranteed to continue across future versions of the languages. Another specific feature of the data structure is that the work-stealing mechanism is managed transparently to the user by the `remove` operation.

In contrast to local operations, global operations apply to the whole `distBag_DFS`, requiring all segments from all bag instances to be visited. These operations include

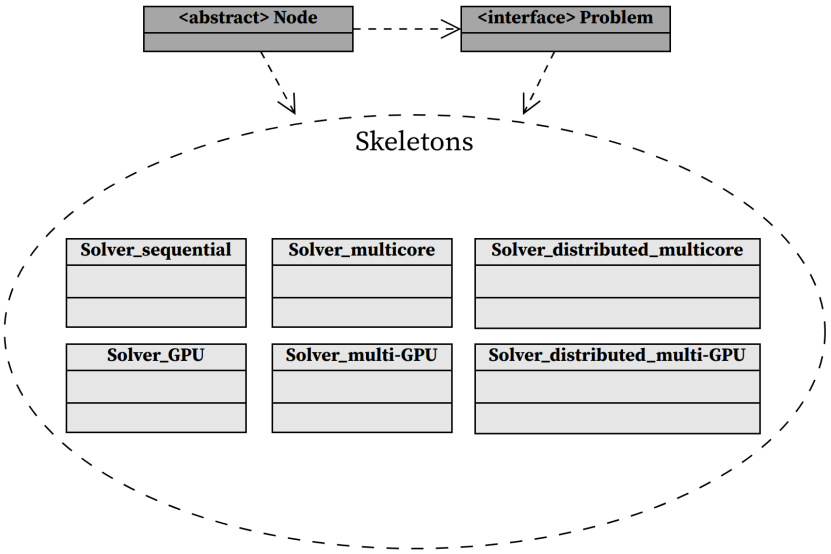


Figure 5.2: UML diagram of the `pBB-chpl` software platform.

`clear`, `contains`, `getSize`, and `these`, and allow to clear the `distBag_DFS`, search for a specific element in it, get its global size, and iterate over its elements, respectively. The first three operations are implemented in a best-effort manner to enhance memory and processing efficiency. This approach sacrifices strict consistency for improved performance and resource usage, making it suitable for operations where occasional inaccuracies are acceptable. Differently, a snapshot approach is preferred for the `these` operation as it provides a stable state of the data structure, avoiding concurrency issues. Indeed, the method creates a consistent view of the data structure at a particular point in time by making a full-copy. Although this increases memory consumption, it also enhances parallelism, allowing other concurrent, even mutating, operations to proceed without interference. However, this approach can result in iterating over duplicated or missing elements due to concurrent modifications. While locking approaches provide strong consistency, they often lead to contention and reduced concurrency especially when many tasks are used, making the snapshot approach a balanced choice for maintaining performance while iterating.

5.3 Skeletons for PGAS-based parallel B&B (`pBB-chpl`)

Figure 5.2 shows the overall structure of the `pBB-chpl` Chapel software platform for PGAS-based parallel B&B algorithms developed in this thesis. In the following, the different levels of abstraction and B&B skeletons are presented, allowing the user to gauge the implications of extending the software platform to another problem. In addition, the various configuration options and target systems are discussed.

5.3.1 Multi-level abstraction

The `pBB-chpl` platform introduces several levels of abstraction for implementing parallel B&B algorithms. The first of these is the `Node` type, which typically contains the data required to compute the bound for a subproblem and also holds the partial solution associated with that subproblem. Algorithm 5.1 illustrates the `Node` types used for the problems tested in this thesis: PFSP, 0/1-Knapsack, N-Queens, and UTS. For a sake of simplicity, initializers are omitted. Common attributes across these types are usually the node's depth and the encoding for the partial solution, which are commonly used for determining the position of the node within the overall search tree and compute bounds. While other platforms may offer a separate abstraction for the solution type, which typically includes both the solution value and the solution itself, `pBB-chpl` uses the same `Node` type to represent the solution. This design choice is based on the perspective that solutions can be seen as special cases of subproblems, where all the decision variables are assigned (*i.e.*, typically when the node's depth equals the problem size).

Algorithm 5.1: Example of `Node` types currently supported in `pBB-chpl`.

```
1  record Node_PFSP
2      var depth: int;
3      var prmu: [ ]*int;
4      var limit1: int;
5      var limit2: int;

6  record Node_Knapsack
7      var depth: int;
8      var items: [ ]*int;
9      var weight: int;
10     var profit: int;

11 record Node_NQueens
12     var depth: int;
13     var board: [ ]*int;

14 record Node_UTS
15     var depth: int;
16     var dist: int;
17     var numChildren: int;
18     var rngState: state_t;
```

A second level of abstraction is obviously dedicated to the definition of the problem, including a function to bound/evaluate a subproblems, which, based on the depth of the node, calculates either the evaluation of the partial solution or the cost of a complete solution; a branching function, which defines the method for dividing a subproblem into smaller subproblems; a function to generate the root problem; and a function to create the initial solution. All these methods are defined in the `Problem` interface, as illustrated in Algorithm 5.2. An interface defines a "contract" outlining specific methods that a class must implement if it chooses to adopt that interface. The latter consist of three "core" methods: `decompose`, which embeds the branching and bounding functions; `getInitSolution`, which determines how the initial solution is initialized; and `copy`, a copy-initializer used to specify how a `Problem` class instance has to be copied to each compute node in distributed settings. In a scenario where a problem requires data and

pre-processing on it before the exploration begins[9], the `copy` initializer can be used to handle the replication of this data across the locales, eliminating the need to repeat the data reading and pre-processing phase on each locale. The interface also includes four additional "utility" methods that allow the user to specify the output format, for example. These methods are typically used for extracting execution statistics or for debugging purposes.

Algorithm 5.2: The `Problem` interface.

```
1 class Problem
    // CORE PROCEDURES
2   proc copy() {}
3   proc decompose(/*args*/) {}
4   proc getInitSolution(): int {}
    // UTILITY PROCEDURES
5   proc print_settings(): void {}
6   proc print_results(/*args*/): void {}
7   proc output_filepath(): string {}
8   proc help_message(): void {}
```

5.3.2 Parallel B&B skeletons and target systems

Once the user has provided the concrete node and problem objects, he/she obtains different parallel solvers. Solvers allow to target several architectures, including multi-core desktops and laptops, commodity clusters, in addition to the high-end supercomputers for which `pBB-chpl` was designed. More precisely, the platform supports the following parallel execution modes:

- *sequential*: Optimized version designed for single-core execution, without any parallel feature. This version primarily serves as a baseline to evaluate the performance of parallel variants and is only applicable to solving small-size problems, where sequential execution is feasible. As an example, Algorithm 5.3 presents the associated Chapel-based B&B skeleton, highlighting a straightforward structure for initialization, tree exploration, and result output. For simplicity, logic related to counting statistics and time measurement has been omitted.

- *single-node multi-core*: Optimized version designed for single-node, multi-core execution, leveraging parallelism within a single machine. It can be used to either provide a reference for evaluating the performance of distributed implementations, or in a standalone manner to solve problems of small to moderate size that can benefit from shared-memory parallelism on a single compute node.

[9] such as for some Knapsack problems, where the list of items has to be ordered according to the weight and profit of each item.

- *multi-node multi-core*: This version, optimized for multi-node, multi-core execution, utilizes distributed parallelism across multiple compute nodes. It is mostly used as a standalone solution for tackling large-scale problems that require the combined resources of multiple compute nodes and benefit from both distributed and shared-memory parallelism. Currently, the skeleton assumes that all compute nodes have an identical number of computing threads. Configurations with heterogeneous compute nodes—such as variations in core count across compute nodes—have not been tested and may not perform as expected.

- *single-node single-GPU*: Optimized for single-GPU execution. It can be used either as a benchmark for evaluating multi-GPU implementations or as a standalone solution for solving moderate-size problems that benefit from GPU parallelism.

- *single-node multi-GPU*: Optimized for single-node, multi-GPU execution, leveraging parallelism across multiple GPUs within a single compute node. This version is suitable for solving larger problems that can benefit from multiple GPUs, providing enhanced computational power for tasks that demand high throughput. It can serve as a benchmark for evaluating distributed multi-GPU implementations or as a standalone solution for problems of medium to large size.

- *multi-node multi-GPU*: Optimized for multi-node, multi-GPU execution, utilizing both distributed parallelism and multiple GPUs across different compute nodes. This version is designed for solving very large-scale problems that require the combined power of multiple GPUs spread across several nodes. It is primarily used in HPC environments where the problem size and computational demands exceed the capabilities of a single-node, multi-GPU configuration.

The portability of the GPU-accelerated implementations across NVIDIA and AMD GPU architectures is ensured by the Chapel compiler, which offers vendor-neutral support for GPU. Specifically, Chapel supports compilation from a single source file to multiple target architectures, including CPUs and GPUs. This is achieved through the LLVM compiler framework, which allows Chapel code generation to target a diverse range of back-ends, such as PTX for NVIDIA GPUs and AMDGCN for AMD GPUs. Other GPU architectures (*e.g.*, Intel GPUs) are not currently supported, but their integration with these platforms is part of the future development plans for the Chapel compiler.

5.4 Conclusion

In this chapter, we have presented the `pBB-chpl` Chapel software platform for PGAS-based parallel B&B algorithms, with a particular focus on its scalable code architecture. The `distBag_DFS` data structure, developed within this context, has been encapsulated in the `DistributedBag` module and integrated into the Chapel programming language. This exposes the data structure to a large community of users interested in parallel

Algorithm 5.3: Chapel-based B&B skeleton for sequential execution.

```
1  use List;
2  use Node, Problem;
3  proc search_sequential(Node, problem)
4      var best = problem.getInitSolution();
5      problem.print_settings();
       // INITIALIZATION
6      var pool: list(Node);
7      var root = new Node (problem);
8      pool.pushBack(root);
       // TREE EXPLORATION
9      while !pool.isEmpty() do
10         var parent: Node = pool.popBack();
11         var children = problem.decompose(Node, parent, best);
12         pool.pushBack(children);
       // OUTPUT
13     problem.print_results(best);
```

computing, and aims to promote its use as well as stimulate scientific collaboration. The `pBB-chpl` platform, on the other hand, provides multiple parallel B&B skeletons allowing to target several architectures, including multi-core desktops and laptops, commodity clusters, in addition to the high-end supercomputers for which `pBB-chpl` was designed. The software platform and associated documentation are open-source and publicly available online through Github repositories.

Chapter 6

Conclusions and Perspectives

Contents

Section 6.1 and Section 6.2 present the general conclusions of this thesis and outline several perspectives, respectively. Lastly, Section 6.3 provides a summary of all scientific production made throughout this PhD research, including the publications and software developments.

6.1 Conclusions

The June 2022 edition of the TOP500 marked a historic turning-point with the beginning of the exascale *era*. Modern systems evolve to tackle unprecedented computational demands, integrating thousands of hybrid compute nodes, each integrating multi-core processors coupled with GPU accelerators. In the near future, exascale supercomputers will have a profound impact on everyday life, quickly analyzing massive volumes of data and more realistically simulating the complex processes and relationships behind many of the fundamental forces of the universe. This leap forward, however, introduces significant challenges for supercomputer programmers, particularly in managing the hierarchical organization, providing multi-level parallelism. In this context, the design and implementation of efficient algorithms for those computing environments is challenging, and the development of new computational paradigms and software to achieve exascale becomes a necessity.

In this thesis the focus is put on exact combinatorial optimization using tree search algorithms. Based on the PGAS paradigm, providing an alternative to the traditional

shared-memory and message-passing models, we have revisited the design and implementation of parallel B&B algorithms on large-scale systems combining multi-core processors and many-core GPU. While PGAS is known to facilitate parallel programming with shared-memory semantics across distributed systems, its use in the optimization context is still in its infancy. Moreover, the irregularity of B&B makes its implementation particularly challenging in computing environments where performance relies on SIMD processing and regular memory access patterns. The proposed algorithms exploit a pool-based design to allow generic implementation with regards to the problem solved. As test-cases, three well-known benchmarks problems are used: the PFSP, the 0/1-Knapsack problem and the N-Queens problem.

The first contribution consists in the design and implementation of the PGAS-based `distBag_DFS` multi-pool data structure dedicated to depth-first exploration of large, irregular trees. It is intrinsically highly parallel and integrates a dynamic load-balancing mechanism based on large-scale work-stealing, operating at both intra- and inter-node levels. This mechanism, which required sophisticated synchronization based on non-blocking double-ended queues, promotes locality in work-stealing, enabling scalability. As a second contribution, we have proposed the design and implementation of a PGAS-based B&B algorithm for CPU-based clusters, based on the `distBag_DFS` data structure. The algorithm exploits genericity of the pool-based data structure to allow a generic design with regards to the tackled optimization problem.

The third major contribution is the extension of the design and implementation of the PGAS-based parallel B&B algorithm to deal with GPU-powered heterogeneous architectures. The algorithm combines the parallel tree exploration on CPU with the parallel evaluation of bounds on GPU to accelerate the compute-intense bounding operator of the B&B algorithm. The workload irregularity is managed by a multi-level dynamic load-balancing mechanism, inspired by the one of `distBag_DFS`, adapted to the context of GPU. From the implementation aspect, the use of Chapel allows high-level abstractions that seamlessly integrate multiple levels of parallelism—CPU, GPU, and inter-node—within a unified programming language. In addition, the portability challenge is addressed by the vendor-neutral GPU-support of the language. For comparison purpose, optimized CUDA-based baseline implementations have also been provided.

The fourth and last contribution consists in the `pBB-chpl` Chapel software platform developed during this thesis for the dissemination of the included PGAS-based parallel B&B skeletons. The software demonstrates extensible and flexible skeletons, allowing to target several architectures, including multi-core desktops and laptops, commodity clusters, in addition to the high-end supercomputers for which `pBB-chpl` was designed. The accessibility is guaranteed by the free and open-source license. Finally, independently from the software platform, the `distBag_DFS` data structure has been integrated into the Chapel programming language (HPE/Cray) as the `DistributedBag` package module.

One of the main objectives of this thesis was to investigate whether the PGAS paradigm can be used to build more efficient and more programmable parallel B&B

algorithms for modern ultra-scale computing platforms. The experiments, conducted on the petascale MeluXina and LUMI supercomputers (460^{th} and 5^{th} in TOP500 in June 2024, respectively), indicate that the answer to this question is highly dependent on both the input and the target platform. On CPU-based clusters, the results show that our `distBag_DFS`-based parallel B&B can achieve near-linear strong scaling efficiencies on hard PFSP instances, while outperforming a state-of-the-art baseline implementation based on MPI+X. More significantly, our algorithm is able to scale up to 400 compute nodes, or 51,200 CPU cores, solving very hard PFSP instances. These good results highlight the suitability of our approach to tackle coarse-grained problems. However, on the 0/1-Knapsack and N-Queens problems, the experiments show limited efficiencies, highlighting the challenges posed by load balancing and fine-grained computations. On GPU-powered clusters, on the other hand, the results show that the PGAS approach allows portable execution on several GPU architectures, while providing faster execution time compared to baselines on modern architectures solving N-Queens instances. However, the complexity of the bounding function for the PFSP problem leads to a performance degradation, attributable to missed optimizations by the Chapel compiler. The strong scaling analysis indicates that our approach achieves reasonable intra-node speed-ups—up to 40% for the fine-grained N-Queens and 73% for the coarse-grained PFSP with 8 GPUs. However, inter-node performance is significantly limited by synchronization overhead, with only 25% efficiency for N-Queens and reduced scalability for PFSP at large scales due to spin-lock-based thread synchronization.

6.2 Perspectives

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

- *Extend/improve proposed approaches*: The experimental results indicated that the granularity of dynamic load balancing in `distBag_DFS` produces an excessive number of WS attempts, particularly in cases where stealing shallow nodes (such as in binary trees) offers no advantage. Looking forward, an adaptive approach to load balancing granularity tailored to the specific characteristics of each problem is proposed. In addition, a comprehensive analysis of the performance limitations encountered in GPU configurations is planned. This direction could further strengthen our collaboration with the Chapel development team (HPE/Cray), potentially leading to recommendations for compiler optimization. Furthermore, investigating other PGAS-based languages, as well as high-level programming languages like Julia [Bez+17], is of interest. While studies comparing high-level languages in parallel metaheuristics exist [Gmy+20a], to the best of our knowledge, no such analysis has been conducted for parallel B&B.

- *Design fault-tolerance mechanisms*: Fault tolerance represents another major challenge that remains to be investigated [Cap09; Sni+14]. Failures can arise from various sources, including hardware failures due to component degradation, software

bugs, network issues like congestion or disconnections, *etc.* It is also established that as supercomputers increase in size to millions of processing cores, their Mean-Time Between Failures tends to become shorter [Sha+19]. In the optimization context, failures lead to the loss of work unit(s) being processed by some thread(s) during the resolution process. Therefore, a major issue, which is particularly critical in exact optimization, is how to recover the failed work units to ensure a reliable execution. One of the mainly used approach in the literature is checkpoint-and-restart method [Sha+13], which involves periodically saving the state of a running application or process (checkpoint) to a stable storage medium, such as a disk. This saved state includes essential data, such as variable values and the program counter, allowing the application to be restored (restart) in the event of a failure. However, this approach raises several questions, mainly: which critical information defines the state of the work units and allows to resume properly their execution? When, where and how (using which data structures) to store it efficiently? How to deal with scalability and heterogeneity?

- *Hybridize B&B with metaheuristics*: It is now widely recognized that hybrid exact-approximate methods outperform traditional optimization approaches when used separately [Meh11]. However, designing such hybrid methods requires considerable effort at the algorithmic level. In fact, there are numerous hybridization schemes, and selecting the most suitable one for a specific application is not always straightforward [Tal09]. Among these, low-level hybrids directly integrate metaheuristics into the B&B process to improve tree exploration and bounding of subproblems. For instance, the choice of the branching operator is known to have a strong impact of the explored tree [Gmy+20b; Cer+17], and metaheuristics can be used to select the one that optimizes the algorithm on a given problem. In contrast, high-level teamwork hybrids combine B&B and metaheuristics at a strategic level, allowing each method to alternate in guiding the search or refining the solution. Another promising application of metaheuristics is their use in tuning the parameters of the B&B algorithm itself.

- *Solve open instances of hard COPs*: Many COPs still have open benchmark instances and their solving to the optimality represents a major challenge. For instance, some of the PFSP Taillard's instances remain open, more than 30 years after the benchmark's release. In a recent attempt to solve them to the optimality, it has been shown that despite best-known UBs seems likely to be optimal, proofs of optimality are very hard to obtain for some 50×20 instances (`ta051`, `ta054`, `ta055`, `ta059` and `ta060`) [Gmy22]. Solution attempts using 3-5,000 GPU-hours per instance failed that objective. Moreover, for the open 100×20 instances, the exploration could not be completed despite using 2-10,000 GPU-hours of computation per instance.

6.3 Dissemination

Below are listed all scientific contributions made throughout this PhD research, including international publications and open-source software.

6.3.1 International peer-reviewed publications

Journals

- **G. Helbecque**, J. Gmys, N. Melab, T. Carneiro, and P. Bouvry. Parallel distributed productivity-aware tree-search using Chapel. *Concurrency Computat Pract Exper* (CCPE). 35(27):e7874, 2023. DOI: 10.1002/cpe.7874.

- [selected for special issue] **G. Helbecque**, E. Krishnasamy, T. Carneiro, N. Melab, and P. Bouvry. Massively Parallel PGAS-based Branch-and-Bound for GPU-powered Clusters. *Concurrency Computat Pract Exper* (CCPE). 2024.

Conferences and workshops with proceedings

- **G. Helbecque**, J. Gmys, T. Carneiro, N. Melab, and P. Bouvry. A performance-oriented comparative study of the Chapel high-productivity language to conventional programming environments. In: *Proceedings of the Thirteenth International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM). pp. 21–29, 2022. DOI: 10.1145/3528425.3529104.

- **G. Helbecque**, E. Krishnasamy, N. Melab, and P. Bouvry. GPU-Accelerated Tree-Search in Chapel versus CUDA and HIP. In: *14th IEEE Workshop Parallel / Distributed Combinatorics and Optimization* (PDCO). 2024. DOI: 10.1109/IPDPSW63119.2024.00156.

- **G. Helbecque**, T. Carneiro, N. Melab, J. Gmys, and P. Bouvry. PGAS Data Structure for Unbalanced Tree-Based Algorithms at Scale. In: *Computational Science – ICCS 2024* (ICCS). vol 14834, 2024. DOI: 10.1007/978-3-031-63759-9_13.

- T. Carneiro, E. Kayraklioglu, **G. Helbecque**, N. Melab. Investigating Portability in Chapel for Tree-based Optimization on GPU-powered Clusters. In: *Euro-Par 2024: Parallel Processing* (EuroPar). LNCS, vol. 14803, pp. 386-399, 2024. DOI: 10.1007/978-3-031-69583-4_27.

- **G. Helbecque**, E. Krishnasamy, T. Carneiro, N. Melab, and P. Bouvry. A Chapel-based Multi-GPU Branch-and-Bound Algorithm: Application to the Flowshop Scheduling Problem. In: *22nd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms* (HeteroPar). 2024.

Conferences and workshops without proceedings

- **G. Helbecque**, J. Gmys, N. Melab, T. Carneiro, and P. Bouvry. Productivity-aware Parallel Distributed Tree-Search for Exact Optimization. In: *6th International Conference on Optimization and Learning* (OLA). 2023.
- **G. Helbecque**, J. Gmys, T. Carneiro, N. Melab, and P. Bouvry. Towards a scalable load balancing for productivity-aware tree-search. In: *10th Annual Chapel Implementers and Users Workshop* (CHIUW). 2023.
- **G. Helbecque**, E. Krishnasamy, N. Melab, and P. Bouvry. GPU Computing in Chapel: Application to Tree-Search Algorithms. In: *7th International Conference on Optimization and Learning* (OLA). 2024.

6.3.2 Open-source software

- The Chapel programming language. The `DistributedBag` package module. Version 2.0 or later. URL: `https://github.com/chapel-lang/chapel`.
- **G. Helbecque**, J. Gmys, T. Carneiro, N. Melab, and P. Bouvry. Productivity- and Performance-aware Parallel Distributed Depth-First Search (P3D-DFS). 2023. DOI: 10.5281/zenodo.7674860, URL: `https://github.com/Guillaume-Helbecque/P3D-DFS`.
- **G. Helbecque**, I. Tagliaferro, T. Carneiro, E. Krishnasamy, N. Melab, and P. Bouvry. GPU-accelerated tree-search in Chapel. 2024. DOI: 10.5281/zenodo.10786276, URL: `https://github.com/Guillaume-Helbecque/GPU-accelerated-tree-search-Chapel`.

References

[ABP98] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. "Thread scheduling for multiprogrammed multiprocessors". In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. 1998, pp. 119–129. DOI: 10.1145/277651.277678.

[ACR13] U. A. Acar, A. Charguéraud, and M. Rainey. "Scheduling parallel programs by work stealing with private deques". In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2013, pp. 219–228. DOI: 10.1145/2442516.2442538.

[Al-+10] A. B. Al-Badareen et al. "Reusable software components framework". In: *Proceedings of the European Conference of Systems, and European Conference of Circuits Technology and Devices, and European Conference of Communications, and European Conference on Computer Science*. 2010, pp. 126–130. DOI: 10.5555/1961414.1961435.

[Alb+02] E. Alba et al. "MALLBA: A Library of Skeletons for Combinatorial Optimisation". In: *Euro-Par 2002 Parallel Processing*. 2002, pp. 927–932. DOI: 10.1007/3-540-45706-2_132.

[Alm11] G. Almasi. "PGAS (Partitioned Global Address Space) Languages". In: *Encyclopedia of Parallel Computing*. Springer US, 2011, pp. 1539–1545. DOI: 10.1007/978-0-387-09766-4_210.

[Ans+02] K. Anstreicher et al. "Solving large quadratic assignment problems on computational grids". In: *Mathematical Programming* 91.3 (2002), pp. 563–588. DOI: 10.1007/s101070100255.

[Arc+19] B. Archibald et al. "Implementing YewPar: A Framework for Parallel Tree Search". In: *Euro-Par 2019: Parallel Processing*. Ed. by Ramin Yahyapour. 2019, pp. 184–196. DOI: 10.1007/978-3-030-29400-7_14.

[Bar+98] C. Barnhart et al. "Branch-and-Price: Column Generation for Solving Huge Integer Programs". In: *Operations Research* 46.3 (1998), pp. 316–329. DOI: 10.1287/opre.46.3.316.

[Bez+17] J. Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[BL99] R. D. Blumofe and C. E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *J. ACM* 46.5 (1999), pp. 720–748. DOI: 10.1145/324133.324234.

[BMT12] A. Bendjoudi, N. Melab, and E.-G. Talbi. "Hierarchical branch and bound algorithm for computational grids". In: *Future Generation Computer Systems* 28.8 (2012), pp. 1168–1176. DOI: 10.1016/j.future.2012.03.001.

[BMT14] A. Bendjoudi, N. Melab, and E.-G. Talbi. "FTH-B&B: A Fault-Tolerant Hierarchical Branch and Bound for Large Scale Unreliable Environments". In: *IEEE Transactions on Computers* 63.9 (2014), pp. 2302–2315. DOI: 10.1109/TC.2013.40.

[BS93] W. J. Bolosky and M. L. Scott. "False sharing and its effect on shared memory performance". In: *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*. 1993.

[Cap09] F. Cappello. "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities". In: *The International Journal of High Performance Computing Applications* 23.3 (2009), pp. 212–226. DOI: 10.1177/1094342009106189.

[Car+11] T. Carneiro et al. "A New Parallel Schema for Branch-and-Bound Algorithms Using GPGPU". In: *2011 23rd International Symposium on Computer Architecture and High Performance Computing*. 2011, pp. 41–47. DOI: 10.1109/SBAC-PAD.2011.20.

[Car+20] T. Carneiro et al. "Towards ultra-scale Branch-and-Bound using a high-productivity language". In: *Future Generation Computer Systems* 105 (2020), pp. 196–209. DOI: 10.1016/j.future.2019.11.011.

[Car+21] T. Carneiro et al. "Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators". In: *HPCS 2020 - The 18th International Conference on High Performance Computing & Simulation*. 2021.

[CBS11] D. Cunningham, R. Bordawekar, and V. Saraswat. "GPU programming in a high level language: compiling X10 to CUDA". In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. 2011. DOI: 10.1145/2212736.2212744.

[CCZ04] D. Callahan, B. Chamberlain, and H. Zima. "The Cascade High Productivity Language". In: *Proceedings. Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*. 2004, pp. 52–60. DOI: 10.1109/HIPS.2004.10002.

[Cer+17] A. Cerqueus et al. "On branching heuristics for the bi-objective 0/1 unidimensional knapsack problem". In: *Journal of Heuristics* 23.5 (2017), pp. 285–319. DOI: 10.1007/s10732-017-9346-9.

[Cha+05] P. Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2005, pp. 519–538. DOI: 10.1145/1094811.1094852.

[Cha+13a] I. Chakroun et al. “Combining multi-core and GPU computing for solving combinatorial optimization problems”. In: *Journal of Parallel and Distributed Computing* 73.12 (2013), pp. 1563–1577. DOI: 10.1016/j.jpdc.2013.07.023.

[Cha+13b] I. Chakroun et al. “Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm”. In: *Concurrency and Computation: Practice and Experience* 25.8 (2013), pp. 1121–1136. DOI: 10.1002/cpe.2931.

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

[Che+11] L. Chen et al. “Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation”. In: *Languages and Compilers for Parallel Computing*. 2011, pp. 151–165. DOI: 10.1007/978-3-642-19595-2_11.

[CP99] J. Clausen and M. Perregaard. “On the best search strategy in parallel branch-and-bound: Best-First Search versus Lazy Depth-First Search”. In: *Annals of Operations Research* 90 (1999), pp. 1–17. DOI: 10.1023/A:1018952429396.

[CZ06] S. Climer and W. Zhang. “Cut-and-solve: An iterative search strategy for combinatorial optimization problems”. In: *Artificial Intelligence* 170.8 (2006), pp. 714–738. DOI: 10.1016/j.artint.2006.02.005.

[Dab+16] A. Dabah et al. “GPU-Based Two Level Parallel B&B for the Blocking Job Shop Scheduling Problem”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 747–755. DOI: 10.1109/IPDPSW.2016.14.

[Dan57] G. B. Dantzig. “Discrete-Variable Extremum Problems”. In: *Operations Research* 5.2 (1957), pp. 266–288. DOI: 10.1287/opre.5.2.266.

[Din+09] J. Dinan et al. “Scalable work stealing”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009. DOI: 10.1145/1654059.1654113.

[Dje+06] A. Djerrah et al. “Bob++: Framework for Solving Optimization Problems with Branch-and-Bound methods”. In: *2006 15th IEEE International Conference on High Performance Distributed Computing*. 2006, pp. 369–370. DOI: 10.1109/HPDC.2006.1652188.

[DMN12] J. Diaz, C. Muñoz-Caro, and A. Niño. "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era". In: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2012), pp. 1369–1386. DOI: 10.1109/TPDS.2011.308.

[DNS97] V. Decyk, C. Norton, and B. Szymanski. "High Performance Object-Oriented Scientific Programming in Fortran 90". In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. 1997.

[DP14] T. van Dijk and J. C. van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing". In: *Euro-Par 2014: Parallel Processing Workshops*. 2014, pp. 206–217. DOI: 10.1007/978-3-319-14313-2_18.

[Dro+12] M. Drozdowski et al. "Grid Branch-and-Bound for Permutation Flowshop". In: *Parallel Processing and Applied Mathematics*. 2012, pp. 21–30. DOI: 10.1007/978-3-642-31500-8_3.

[EHP15] J. Eckstein, W. E. Hart, and C. A. Phillips. "PEBBL: an object-oriented framework for scalable parallel branch and bound". In: *Mathematical Programming Computation* 7.4 (2015), pp. 429–469. DOI: 10.1007/s12532-015-0087-1.

[EPH01] J. Eckstein, C. A. Phillips, and W. E. Hart. "Pico: An Object-Oriented Framework for Parallel Branch and Bound". In: *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*. Vol. 8. Studies in Computational Mathematics. Elsevier, 2001, pp. 219–265. DOI: 10.1016/S1570-579X(01)80014-8.

[EST92] C. Erbas, S. Sarkeshik, and M. M. Tanik. "Different perspectives of the N-Queens problem". In: *Proceedings of the 1992 ACM Annual Conference on Communications*. 1992, pp. 99–108. DOI: 10.1145/131214.131227.

[Fei+10] F. Feinbube et al. "NQueens on CUDA: Optimization Issues". In: *2010 Ninth International Symposium on Parallel and Distributed Computing*. 2010, pp. 63–70. DOI: 10.1109/ISPDC.2010.22.

[GC94] B. Gendron and T. G. Crainic. "Parallel Branch-And-Bound Algorithms: Survey and Synthesis". In: *Operations Research* 42.6 (1994), pp. 1042–1066. DOI: 10.1287/opre.42.6.1042.

[GGS03] B. Goldengorin, D. Ghosh, and G. Sierksma. "Branch and peg algorithms for the simple plant location problem". In: *Computers & Operations Research* 30.7 (2003), pp. 967–981. DOI: 10.1016/S0305-0548(02)00049-7.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[GJS76] M. R. Garey, D. S. Johnson, and R. Sethi. "The Complexity of Flowshop and Jobshop Scheduling". In: *Mathematics of Operations Research* 1.2 (1976), pp. 117–129. DOI: 10.1287/moor.1.2.117.

[Gmy+16] J. Gmys et al. “IVM-Based Work Stealing for Parallel Branch-and-Bound on GPU”. In: *Parallel Processing and Applied Mathematics*. Cham: Springer International Publishing, 2016, pp. 548–558. DOI: 10.1007/978-3-319-32149-3_51.

[Gmy+17] J. Gmys et al. “IVM-based parallel branch-and-bound using hierarchical work stealing on multi-GPU systems”. In: *Concurrency and Computation: Practice and Experience* 29.9 (2017), e4019. DOI: 10.1002/cpe.4019.

[Gmy+20a] J. Gmys et al. “A comparative study of high-productivity high-performance programming languages for parallel metaheuristics”. In: *Swarm and Evolutionary Computation* 57 (2020), p. 100720. DOI: 10.1016/j.swevo.2020.100720.

[Gmy+20b] J. Gmys et al. “A computationally efficient Branch-and-Bound algorithm for the permutation flow-shop scheduling problem”. In: *European Journal of Operational Research* 284.3 (2020), pp. 814–833. DOI: 10.1016/j.ejor.2020.01.039.

[Gmy17] J. Gmys. “Heterogeneous cluster computing for many-task exact optimization - Application to permutation problems”. PhD thesis. Université de Mons and Université de Lille, 2017.

[Gmy22] J. Gmys. “Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers”. In: *INFORMS Journal on Computing* 34.5 (2022), pp. 2502–2522. DOI: 10.1287/ijoc.2022.1193.

[GR14] M. B. Giles and I. Reguly. “Trends in high-performance computing for engineering calculations”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372.2022 (2014), p. 20130319. DOI: 10.1098/rsta.2013.0319.

[HPS19] A. Hayashi, S. R. Paul, and V. Sarkar. “GPUIterator: bridging the gap between Chapel and GPU platforms”. In: *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*. 2019, pp. 2–11. DOI: 10.1145/3329722.3330142.

[HPS23] A. Hayashi, S. R. Paul, and V. Sarkar. “A Multi-Level Platform-Independent GPU API for High-Level Programming Models”. In: *High Performance Computing. ISC High Performance 2022 International Workshops*. 2023, pp. 90–107. DOI: 10.1007/978-3-031-23220-6_7.

[HS05] S. Reza Hejazi and S. Saghafian. “Flowshop-scheduling problems with makespan criterion: a review”. In: *International Journal of Production Research* 43.14 (2005), pp. 2895–2929. DOI: 10.1080/00207540500056417.

[Jen+11] J. Jenkins et al. “Lessons Learned from Exploring the Backtracking Paradigm on the GPU”. In: *Euro-Par 2011 Parallel Processing*. 2011, pp. 425–437. DOI: 10.1007/978-3-642-23397-5_42.

[JFK17] L. Jenkins, M. Ferguson, and E. Kayraklioglu. *Distributed Data Structures*. Google Summer of Code program. 2017. URL: https://summerofcode.withgoogle.com/archive/2017/projects/6530769430249472.

[Joh54] S. M. Johnson. "Optimal two- and three-stage production schedules with setup times included". In: *Naval Research Logistics Quarterly* 1.1 (1954), pp. 61–68. DOI: 10.1002/nav.3800010110.

[Kec+11] S. W. Keckler et al. "GPUs and the Future of Parallel Computing". In: *IEEE Micro* 31.5 (2011), pp. 7–17. DOI: 10.1109/MM.2011.89.

[KK84] V. Kumar and L. N. Kanal. "Parallel Branch-and-Bound Formulations for AND/OR Tree Search". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-6.6 (1984), pp. 768–778. DOI: 10.1109/TPAMI.1984.4767600.

[KKS04] K. Kennedy, C. Koelbel, and R. Schreiber. "Defining and Measuring the Productivity of Programming Languages". In: *The International Journal of High Performance Computing Applications* 18.4 (2004), pp. 441–448. DOI: 10.1177/1094342004048537.

[Lag96] M. G. Lagoudakis. "The 0–1 Knapsack Problem: An Introductory Survey". 1996.

[LE12] M. E. Lalami and D. El-Baz. "GPU Implementation of the Branch and Bound Method for Knapsack Problems". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 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 de Lille 1, 2015.

[Li+15] L. Li et al. "A Parallel Algorithm for Game Tree Search Using GPGPU". In: *IEEE Transactions on Parallel and Distributed Systems* 26.8 (2015), pp. 2114–2127. DOI: 10.1109/TPDS.2014.2345054.

[LLR78] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. "A General Bounding Scheme for the Permutation Flow-Shop Problem". In: *Operations Research* 26.1 (1978), pp. 53–67. DOI: 10.1287/opre.26.1.53.

[LY07] E. Lusk and K. Yelick. "Languages for High-Productivity Computing: The DARPA HPCS Language Project". In: *Parallel Processing Letters* 17.1 (2007), pp. 89–102. DOI: 10.1142/S0129626407002892.

[MAD13] R. Machado, S. Abreu, and D. Diaz. "Parallel Local Search: Experiments with a PGAS-based programming model". In: *Proceedings of CICLOPS 2012 12th International Colloquium on Implementation of Constraint and LOgic Programming Systems*. 2013. DOI: 10.48550/arXiv.1301.7699.

[MC98] J. Matocha and T. Camp. "A taxonomy of distributed termination detection algorithms". In: *Journal of Systems and Software* 43.3 (1998), pp. 207–221. DOI: 10.1016/S0164-1212(98)10034-1.

[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*. 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". In: *Concurrency and Computation: Practice and Experience* 26.16 (2014), pp. 2667–2683. DOI: 10.1002/cpe.3155.

[Meh11] M. Mehdi. "Parallel hybrid optimization methods for permutation based problems". PhD thesis. Université de Lille 1 and Université du Luxembourg, 2011.

[Mel05] N. Melab. "Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul". Thèse HDR. Université des Sciences et Technologies de Lille, 2005.

[Mez+14] M. Mezmaz et al. "A Multi-core Parallel Branch-and-Bound Algorithm Using Factorial Number System". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1203–1212. DOI: 10.1109/IPDPS.2014.124.

[Mit11] J. E. Mitchell. "Branch and Cut". In: *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Ltd, 2011. DOI: 10.1002/9780470400531.eorms0117.

[MMT07] 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*. 2007, pp. 1–9. DOI: 10.1109/IPDPS.2007.370217.

[Mun+14] D. Munera et al. "A Parametric Framework for Cooperative Parallel Local Search". In: *Evolutionary Computation in Combinatorial Optimisation*. 2014, pp. 13–24. DOI: 10.1007/978-3-662-44320-0_2.

[MWA24] J. Milthorpe, X. Wang, and A. Azizi. "Performance Portability of the Chapel Language on Heterogeneous Architectures". In: *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2024, pp. 6–13. DOI: 10.1109/IPDPSW63119.2024.00011.

[Oli+07] S. Olivier et al. "UTS: An Unbalanced Tree Search Benchmark". In: *Languages and Compilers for Parallel Computing*. 2007, pp. 235–250. DOI: 10.1007/978-3-540-72521-3_18.

[Par+19] L. A. Parnell et al. “Trends in High Performance Computing: Exascale Systems and Facilities Beyond the First Wave”. In: *2019 18th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*. 2019, pp. 167–176. DOI: 10.1109/ITHERM.2019.8757229.

[PC04] R. Pastor and A. Corominas. “Branch and win: OR tree search algorithms for solving combinatorial optimisation problems”. In: *Top* 12.1 (2004), pp. 169–191. DOI: 10.1007/BF02578930.

[PE17] T. B. Preußer and M. R. Engelhardt. “Putting Queens in Carry Chains, №27”. In: *J Sign Process Syst* 88 (2017), pp. 185–201. DOI: 10.1007/s11265-016-1176-8.

[Pis05] D. Pisinger. “Where are the hard knapsack problems?” In: *Computers & Operations Research* 32.9 (2005), pp. 2271–2284. DOI: 10.1016/j.cor.2004.03.002.

[RG05] T. K. Ralphs and M. Güzelsoy. “The Symphony Callable Library for Mixed Integer Programming”. In: *The Next Wave in Computing, Optimization, and Decision Technologies*. Boston, MA: Springer US, 2005, pp. 61–76. DOI: 10.1007/0-387-23529-9_5.

[Ric97] M. Richards. *Backtracking algorithms in MCPL using bit patterns and recursion*. Tech. rep. UCAM-CL-TR-433. University of Cambridge, Computer Laboratory, 1997. DOI: 10.48456/tr-433.

[Rou87] C. Roucairol. “A parallel branch and bound algorithm for the quadratic assignment problem”. In: *Discrete Applied Mathematics* 18.2 (1987), pp. 211–225. DOI: 10.1016/0166-218X(87)90022-9.

[RS10] K. Rocki and R. Suda. “Parallel Minimax Tree Searching on GPU”. In: *Parallel Processing and Applied Mathematics*. 2010, pp. 449–456. DOI: 10.1007/978-3-642-14390-8_47.

[RXX11] D. Rouson, J. Xia, and X. Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge University Press, 2011. DOI: 10.1017/CBO9780511977381.

[SB04] M. Snir and D. A. Bader. “A Framework for Measuring Supercomputer Productivity”. In: *The International Journal of High Performance Computing Applications* 18.4 (2004), pp. 417–432. DOI: 10.1177/1094342004048535.

[SD08] T. Sterling and C. Dekate. “Productivity in High-Performance Computing”. In: *Advances in COMPUTERS*. Vol. 72. Advances in Computers. Elsevier, 2008, pp. 101–134. DOI: 10.1016/S0065-2458(08)00002-8.

[Sha+13] F. Shahzad et al. “A survey of checkpoint/restart techniques on distributed memory systems”. In: *Parallel Processing Letters* 23.04 (2013), p. 1340011. DOI: 10.1142/S0129626413400112.

[Sha+19] F. Shahzad et al. “CRAFT: A Library for Easier Application-Level Checkpoint/Restart and Automatic Fault Tolerance”. In: *IEEE Transactions on Parallel and Distributed Systems* 30.3 (2019), pp. 501–514. DOI: 10.1109/TPDS.2018.2866794.

[Sha11] N. Shavit. “Data structures in the multicore age”. In: *Commun. ACM* 54.3 (2011), pp. 76–84. DOI: 10.1145/1897852.1897873.

[Sid+12] A. Sidelnik et al. “Performance Portability with the Chapel Language”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 582–594. DOI: 10.1109/IPDPS.2012.60.

[Sni+14] M. Snir et al. “Addressing failures in exascale computing”. In: *Int. J. High Perform. Comput. Appl.* 28.2 (2014), pp. 129–173. DOI: 10.1177/1094342014522573.

[SRR08] P. San Segundo, D. Rodriguez-Losada, and C. Rossi. “Recent Developments in Bit-Parallel Algorithms”. In: *Tools in Artificial Intelligence*. Rijeka: IntechOpen, 2008. Chap. 20. DOI: 10.5772/6076.

[Ste+11] G. L. Steele et al. “Fortress (Sun HPCS Language)”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 718–735. DOI: 10.1007/978-0-387-09766-4_190.

[Tai93] E. Taillard. “Benchmarks for basic scheduling problems”. In: *European Journal of Operational Research* 64.2 (1993), pp. 278–285. DOI: 10.1016/0377-2217(93)90182-M.

[Tal09] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009. ISBN: 978-0-470-27858-1.

[TOP24] TOP500.org. *TOP500 The List*. 2024. URL: https://www.top500.org/.

[VD16] T.-T. Vu and B. Derbel. “Parallel Branch-and-Bound in multi-core multi-CPU multi-GPU heterogeneous environments”. In: *Future Generation Computer Systems* 56 (2016), pp. 95–109. DOI: 10.1016/j.future.2015.10.009.

[Weg90] Peter Wegner. “Concepts and paradigms of object-oriented programming”. In: *SIGPLAN OOPS Mess.* 1.1 (1990), pp. 7–87. DOI: 10.1145/382192.383004.

[Xu+05] Y. Xu et al. “Alps: A Framework for Implementing Parallel Tree Search Algorithms”. In: *The Next Wave in Computing, Optimization, and Decision Technologies*. Boston, MA: Springer US, 2005, pp. 319–334. DOI: 10.1007/0-387-23529-9_21.

[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*. 2011, pp. 142–156. DOI: 10.5555/2042522.2042533.

Appendix A

Instances and Execution Statistics

Tables A.1 to A.4 summarize the PFSP, 0/1-Knapsack, N-Queens, and UTS instances solved in this thesis, respectively, including their parameters and execution statistics[10].

Table A.1: Summary of the PFSP instances solved in this thesis. For each size, instances are sorted by their number of decomposed nodes.

(a) LB2 bound with the forward branching rule.

Instance	size ($m \times n$)	# nodes	Optimum
ta010	5×20	8,122,579	1,108
ta020	10×20	4,870,386	1,591
ta029	20×20	9,499,307	2,237
ta030	20×20	13,228,600	2,178
ta022	20×20	14,561,974	2,099
ta027	20×20	54,588,346	2,273
ta023	20×20	115,511,993	2,326
ta028	20×20	196,916,205	2,200
ta025	20×20	234,988,695	2,291
ta026	20×20	514,453,278	2,226
ta024	20×20	2,173,092,255	2,223
ta021	20×20	3,944,527,267	2,297

(b) LB1 bound with the *minBranch* branching rule.

Instance	size ($m \times n$)	# nodes	Optimum
ta056	20×50	173,314,864,803	3,679
ta052	20×50	17,117,837,075,484	3,699
ta057	20×50	28,340,718,802,452	3,704
ta053	20×50	94,885,132,700,180	3,640

[10] kp1, kp2, kp3, kp4, kp5, and kp6 are aliases for knapPI_3_100_1000_83, knapPI_3_100_1000_15, knapPI_13_50_1000_45, knapPI_13_100_1000_44, knapPI_13_100_1000_4, and knapPI_13_100_1000_5, respectively.

Table A.2: Summary of the 0/1-Knapsack instances solved in this thesis. Instances are sorted by their number of decomposed nodes.

Instance	# items	Capacity	# nodes	Optimum
`kp1`	100	44,997	75,469,089	53,897
`kp2`	100	8,777	225,989,975	11,777
`kp3`	50	8,354	1,560,143,662	12,242
`kp4`	100	12,907	4,945,928,475	19,047
`kp5`	100	1,315	13,637,806,638	1,734
`kp6`	100	1,594	91,261,029,078	2,601

Table A.3: Summary of the N-Queens instances solved in this thesis.

N	# nodes	# solutions
15	171,129,071	2,279,184
16	1,141,190,302	14,772,512
17	8,017,021,931	95,815,104
18	59,365,844,490	666,090,624
19	461,939,618,823	4,968,057,848

Table A.4: Summary of the UTS instances solved in this thesis. Instances are sorted by their number of decomposed nodes.

Instance	Tree type	b_0	p	q	a	d	r	# nodes
`UTS-geo`	GEO	7	-	-	2	23	201	91,373,715
`UTS-bin`	BIN	2000	2	0.499995	-	-	167	131,739,000

Appendix B

Hardware and Software Configuration

B.1 Hardware

The following systems and architectures were used in this thesis:

- **Aion cluster (Université du Luxembourg):**
  - **AMD Rome 7H12:** two 64-core AMD EPYC Rome 7H12 @ 2.6 GHz CPUs, and 256 GB of RAM per node. Nodes are connected through an InfiniBand HDR 100 Gb/s network, configured over a Fat-Tree topology.
- **MeluXina - Cluster module (ranked 460^{th} in June 2024 TOP500):**
  - **AMD Rome 7H12:** two 64-core AMD EPYC Rome 7H12 @ 2.6 GHz CPUs, and 512 GB of RAM per node. Nodes are connected to an InfiniBand HDR 200 Gb/s high-speed fabric, configured over a DragonFly+ topology.
- **LUMI (ranked 5^{th} in June 2024 TOP500):**
  - **AMD MI250X:** single 64-core AMD EPYC 7A53 "Trento" (Zen 3) @ 2.0 GHz CPU, equipped with four AMD Instinct MI250X GPUs (14,080 cores, released in November 2021), and 512 GB of RAM per node. Nodes are connected to a HPE Cray Slingshot-11 200 Gb/s network interconnect, configured over a DragonFly topology.
- **Grid'5000 testbed**:
  - **NVIDIA A100:** single 32-core AMD EPYC 7513 (Zen 3) @ 2.60 GHz CPU, equipped with four NVIDIA A100 SXM4 40GB GPUs (6,912 cores, released in May 2020), and 512 GB of RAM.

- **NVIDIA V100:** two 12-core Intel Xeon Gold 6126 (Skylake-SP) @ 2.60 GHz CPUs, equipped with two NVIDIA Tesla V100 PCIE 32GB GPUs (5,120 cores, released in March 2018), and 192 GB of RAM.
- **NVIDIA P100:** two 12-core Intel Xeon Gold 6126 (Skylake-SP) @ 2.60 GHz CPUs, equipped with two NVIDIA Tesla P100 PCIE 16GB GPUs (3,584 cores, released in June 2016), and 192 GB of RAM.
- **AMD MI50:** single 48-core AMD EPYC 7642 (Zen 2) @ 2.40 GHz CPU, equipped with eight AMD Radeon Instinct MI50 32GB GPUs (3,840 cores, released in November 2018), and 512 GB of RAM.
- **AMD MI300X (early access):** 64-core AMD EPYC 7A53 "Trento" (Zen 3) @ 2.0 GHz CPU, equipped with a AMD Instinct MI300X GPU (19,456 cores, released in June 2023).

B.2 Software configuration

The Chapel compiler was manually built from source on each system described in the previous section. Table B.1 summarizes the environment configuration used for building the compiler and running the code. Only non-default options are listed; further details can be found at `https://chapel-lang.org/docs/usingchapel/chplenv.html`.

Table B.1: Chapel environment configuration for each target architecture.

	CHPL_RT_NUM_THREADS_PER_LOCALE	CHPL_COMM	CHPL_COMM_SUBSTRATE	CHPL_LAUNCHER	CHPL_IBV_SPAWNERS	CHPL_LLVM	CHPL_LOCALE_MODEL	CHPL_GPU	CHPL_GPU_ARCH	CHPL_RT_NUM_GPUS_PER_LOCALE
AMD Rome 7H12 (Aion)	128	gasnet	ibv	gasnetrun_ibv	ssh	none	-	-	-	-
AMD Rome 7H12 (MeluXina)	128	gasnet	ibv	gasnetrun_ibv	mpi	none	-	-	-	-
NVIDIA A100	32	-	-	-	-	bundled	gpu	nvidia	sm_80	4
NVIDIA V100	24	-	-	-	-	bundled	gpu	nvidia	sm_70	2
NVIDIA P100	24	-	-	-	-	bundled	gpu	nvidia	sm_60	2
AMD MI300X	64	-	-	-	-	system	gpu	amd	gfx942	8
AMD MI250X	64	ofi	-	slurm-srun	-	system	gpu	amd	gfx90a	8
AMD MI50	48	-	-	-	-	system	gpu	amd	gfx906	8