SCALABLE AND FAULT TOLERANT HIERARCHICAL B&B ALGORITHMS FOR COMPUTATIONAL GRIDS

Ahcène Bendjoudi

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université A.MIRA-BEJAIA
Faculté des Sciences Exactes
Département Informatique

THÈSE

Présentée par

BENDJOUDI Ahcène

Pour l'obtention du grade de

DOCTEUR EN SCIENCES

Filière : Informatique
Option : Réseaux et Systèmes Distribués

Thème

SCALABLE AND FAULT TOLERANT HIERARCHICAL B&B ALGORITHMS FOR COMPUTATIONAL GRIDS

Soutenue le : . . / . . / 2012

Devant le Jury composé de :

Mr DAHMANI Abdelnasser	Professeur	Université de Béjaïa	Président
Mr TALBI El-Ghazali	Professeur	Université Lille1	Rapporteur
Mr MELAB Nouredine	Professeur	Université Lille1	Rapporteur
Mr TARI Abdelkamel	Maître de Conférence A	Université de Béjaia	Examinateur
Mr BADACHE Nadjib	Professeur	USTHB	Examinateur
Mr KOUDIL Mouloud	Professeur	ESI	Examinateur

Année Universitaire : 2011/2012

Abstract:

Solving to optimality large instances of combinatorial optimization problems using Branch and Bound (B&B) algorithms requires a huge amount of computing resources. Nowadays, such power is provided by large scale environments such as computational grids. However, grids induce new challenges: scalability, heterogeneity, and fault tolerance. Most of existing grid-based B&Bs are developed using the Master-Worker paradigm, their scalability is therefore limited. Moreover fault tolerance is rarely addressed in these works. In this thesis, we propose three main contributions to deal with these issues: *P2P-B&B*, *H-B&B*, and *FTH-B&B*. P2P-B&B is a MW-based B&B framework which deals with scalability by reducing the task request frequency and enabling direct communication between workers. H-B&B also deals with scalability. Unlike the state-of-the-art approaches, H-B&B is fully dynamic and adaptive, meaning it takes into account the dynamic acquisition of new computing resources. FTH-B&B is based on new fault tolerant mechanisms enabling efficient building of the hierarchy and maintaining its balancing, and minimizing of work redundancy when storing and recovering tasks. The proposed approaches have been implemented using ProActive grid-middleware and applied to the Flow-Shop scheduling Problem (FSP). The large scale experiments performed on Grid'5000 proved the efficiency of the proposed approaches.

Keys Words : *Parallel B&B, Master-Worker, Hierarchical Master-Worker, Fault Tolerance, Grid Computing, Large Scale Experiment, FSP, ProActive, Grid'5000.*

Résumé:

La résolution exacte de problèmes d'optimisation combinatoire avec les algorithmes Branch and Bound (*B&B*) nécessite un nombre exorbitant de ressources de calcul. Actuellement, cette puissance est offerte par les environnements large échelle comme les grilles de calcul. Cependant, les grilles présentent de nouveaux challenges : le passage à l'échelle, l'hétérogénéité et la tolérance aux pannes. La majorité des algorithmes B&B revisités pour les grilles de calcul sont basés sur le paradigme Master-Worker, ce qui limite leur passage à l'échelle. De plus, la tolérance aux pannes est rarement adressée dans ces travaux. Dans cette thèse, nous proposons trois principales contributions : *P2P-B&B*, *H-B&B* et *FTH-B&B*. P2P-B&B est un famework basé sur le paradigme Master-Worker traite le passage à l'échelle par la réduction de la fréquence de requêtes de tâches et en permettant les communications directes entre les workers. H-B&B traite aussi le passage à l'échelle. Contrairement aux approches proposées dans la littérature, H-B&B est complètement dynamique et adaptatif i.e. prenant en compte l'acquisition dynamique des ressources de calcul. FTH-B&B est basé sur de nouveaux méchanismes de tolérance aux pannes permettant de construire et maintenir la hiérarchie équilibrée, et de minimiser la redondance de travail quand les tâches sont sauvegardées et restaurées. Les approches proposées ont été implémentées avec la plateforme pour grille ProActive et ont été appliquées au problème d'ordonnancement de type Flow-Shop. Les expérimentations large échelle effectuées sur la grille Grid'5000 ont prouvé l'éfficacité des approches proposées.

Mots clés : *B&B Parallèle, Master-Worker, Master-Worker Hiérarchique, Tolérance aux Pannes, Grilles, FSP, ProActive, Grid'5000.*

Contents

List of Tables

List of Figures

Acknowledgements

I will take this opportunity to thank gratefully my thesis advisors, Pr El Ghazali Talbi and Pr Nouredine Melab for their availability and continuous support during my thesis preparation. Many thanks to Pr El-Ghazali Talbi for his valuable remarks and advices. Special thanks also to Pr Nouredine Melab for being so deeply involved in my work, for the lots of fruitful e-mail exchanges, and also for his efficient way to review the papers related to this work and thesis.

I'm also pleased to thank Pr Abdelnasser Dahmani, Pr Nadjib Badache, Pr Mouloud Koudil, Dr Abdelkamel Tari, and again Pr El-Ghazali Talbi and Pr Nouredine Melab who honor me by their presence as examining committee in my thesis defense.

I would like to greatly thank again the director of the CERIST center of research Pr Nadjib Badache, the chief of DTISI division Pr Omar Nouali, and my first line responsible Dr Nadia Nouali, for their support and unbounded trust on me and also for the means they managed to secure for me to fully accomplish my thesis preparation in an efficient way. Without forgetting to acknowledge my dearest colleagues in Pervasive computing team, Nadir Bouchama, Abdelaziz Babakhouya, Yacine Belhoul, Saïd Yahiaoui, Malika Mehdi, Hocine Saadi, and Abderezak Seba, for the whole time we spent together working, debating, brainstorming, sharing ideas and more.

I'm obliged to the responsible of the doctoral school ReSyD Dr Abdelkamel Tari for his help and all the administration facilities to the achievement of this thesis.

My greetings to all members of Dolphin team and specially to Yacine Kessaci, Mostepha Redouane Khouadjia, Moustapha Diaby, Mathieu Djamai, and Thé Van Luong with whom I exchanged valuable ideas during my visit to LIFL and helped me to move forward efficiently in my work preparation.

Last but not least, thanks to Abderrahmane Sider, Saïd Gharout, Mohamed Abdelghani Bouaissa, and Ali Benssam for their help, and their comments on my thesis.

Introduction

MANY real world economic problems can be modeled as Combinatorial Optimization Problems (*COPs*). Their solving consists in finding one or several best solution(s) among a very large but finite set of possible configurations. Each configuration constitutes a solution to the considered problem and belongs to a space called *search space*. Solving to optimality COPs consists in finding the optimal solution among the large set of feasible solutions. In practice, COP instances are often large in size and CPU time-intensive. They require a huge amount of computing resources to be solved optimally. These problems are known to be NP-hard [GAR79] and their sequential exact resolution is impractical. The most used exact methods are *Branch and X (B&X)* algorithms which refer to three variants of methods: *Branch and Cut (B&C)*, *Branch and Price (B&P)*, and *Branch and Bound (B&B)*. These algorithms perform an implicit enumeration of the search space instead of exhaustive one. Based on a pruning technique, they reduce considerably the computation time required to explore the whole search space. However, such a technique is not sufficient when very large problem instances are to be dealt with. High performance computing such as grid-based parallel computing is required.

Nowadays, computational grids provide a huge amount of computing resources that can offer the power required by parallel B&B algorithms to solve large COP instances. A computational grid, as defined by Foster *et al.* in [FK98], is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. These last years, computational Grids have been widely deployed around the world to provide high performance computing tools for research and industry. Grids gather large amount of heterogeneous resources across geographically distributed sites to a single virtual organization. Resources are usually volatile, heterogenous, autonomous, and organized into clusters managed by different administrative domains. These characteristics make arising new challenges to the classical parallel B&B algorithms.

The traditional parallel algorithms must be rethought to meet the characteristics of grids, particularly their large scale and the volatility and heterogeneity of their resources. Scalability is a major issue to be tackled as B&Bs executed in large scale

environments spawn a huge number of work units which must be executed and/or communicated. Consequently, bottlenecks can be created on some parts of the network or some computational resources. The presence of node failures in the computational grid requires dealing with the fault tolerance issue to correctly perform the computation. The heterogeneous and the dynamic nature of grids require to balance the work load in order to maximize the use of the grid resources during the exploration process. Using communications between distributed processes may increase the computation speedup. However, grid environments are not adapted for communication because of the high latency between sites.

Several B&B algorithms and frameworks have been developed and adapted to large scale environments using the Master-Worker paradigm (*MW*) [ACK+02][GLY00]. The MW paradigm consists in defining two entities: a single master and a pool of workers. The master decomposes an initial task into multiple smaller ones and distributes them among the workers. The workers, on their side, perform the execution of the different tasks. After a worker ends its calculation, it sends back the result to the master and asks for a new task. This simple mechanism makes the MW paradigm widely studied and successfully used for many parallel applications. Consequently, many of sequential applications can be easily brought to the MW paradigm since the whole algorithm control is done by the master. Indeed, users only have to find a way to suitably decompose the problem to be solved, to distribute tasks, to gather results and to terminate the calculation.

However, the MW paradigm is strongly limited regarding scalability in large scale environments such as computational grids [ANF03][AFO06]. Indeed, the central master process is subject to bottlenecks caused by the *many-to-one* requests submitted by the different workers. This slows down the master in serving the workers and these later in executing their tasks, thus degrading the global performance of the exploration process. In the literature, many techniques have been explored to overcome this drawback [MMT07a, MMT07b, EPH00, ANF03, AFO06, GSDB09, BDLP08, DVC+09]. These techniques fall into two categories. The first category includes techniques maintaining the use of the MW scheme and modifying the used resolution methods [MMT07a, MMT07b, EPH00] adapting them to the large scale environment. In the second category, modifications are brought to the main scheme of the MW paradigm [ANF03, AFO06, GSDB09, BDLP08, DVC+09] based on the Hierarchical Master-Worker paradigm (*HMW*). In this paradigm, there are multiple masters, each of them supervises a set of workers. Therefore, the hierarchical organization allows pushing the limits of the MW to bear more worker processes.

Although, a higher scalability can be achieved using hierarchical algorithms, designing such HMW-B&B algorithms is not straightforward since they must also deal with the unreliability of the computing resources. Fault Tolerance (*FT*) is the second major issue to deal with when developing grid-based B&Bs. The computing resources are highly unreliable and volatile. They are unforeseeable; they join and leave the system frequently. FT can be achieved at two levels: *application-level* or *middleware-level*. Several middlewares provide FT mechanisms to reliable execution of

applications: ProActive [PROA][BBC+06][CDCL06], XtremWeb [XTRE], and Condor [CON][GSDB09][PL96]. Nevertheless, they are costly in terms of CPU execution time slowing down the application. The second strategy is application-level and consists in introducing FT mechanism(s) within the algorithms [MMT07a, MMT07b, IAM00, FM87, GKLY00, GLY00, DVC+09]. Three major aspects must be taken into account when designing FT at application-level: First, one must ensure fault recovery to avoid the loss of work units and to gain in terms of execution time by minimizing the redundant work. Second, one must ensure to maintain the same topology (formed between the different processes of the B&B) during the lifetime of the algorithm. In addition, one must avoid orphan branches in order to guarantee a valid functioning and then a valid result of the algorithm. Third, one must ensure an efficient restart of a great number of failed processes.

However, most of the proposed HMW approaches are static and only composed of one level of masters each of them manages a set of workers. They still have a limited scalability in large scale environments of thousands of processors such as computational grids. In addition, they handle tasks of fixed granularity and do not evolve in time to deal with the dynamic nature of grids. Moreover, even proposed B&Bs deal with FT, they are limited in terms of work redundancy because they are based on checkpointing and rollback mechanisms. Indeed, when a process fails, the entire subproblem (assigned to it before) is processed again from scratch by another safe process inducing more redundant work.

In this thesis, we present three main contributions: First, we propose a P2P MW-based B&B framework (*P2P-B&B*) aiming at facilitating the development of grid-based B&Bs, hiding the complexity of the grid to the users, and developing complete grid-based B&Bs. In this framework the scalability is achieved by reducing the task request frequency and enabling direct communication between workers. The task request frequency is minimized by proposing a solution to handle coarse-grained tasks without losing performance by performing multiple executions of an atomic task rather than single execution of a compact coarse-grained task. Direct communications allow the workers to share their upper bounds and to perform other collaboration tasks alleviating the master process.

Second, we propose a new HMW-based B&B (*H-B&B*) aiming at improving the scalability of the conventional MW-based B&B paradigm by eliminating the bottlenecks created on the central master process. H-B&B is based on the *P2P-B&B* framework. Unlike the literature approaches, H-B&B is fully dynamic as it is composed of several levels of masters, and evolves over time according to the dynamic acquisition of new computing nodes. It includes several sub-MW-based B&B systems where each master manages a set of workers organized into groups. The processes of H-B&B form a hierarchy where the inner nodes host masters and the leaves host workers. Masters perform decentralized branching on subproblems using a new exploration strategy and workers perform a complete exploration of the received subproblems. The components of a same sub-B&B system perform in a collaborative way the same task assigned to them, share the result of that task, and return it back to their master. The different components of H-B&B handle tasks of different grain sizes according to their roles (master or worker)

and to their position in the hierarchy. Accordingly, bottlenecks likely to be created at centralized points in the hierarchy can be controlled.

Finally, we propose a Fault Tolerant Hierarchical B&B (*FTH-B&B*) designed for large scale unreliable environments such as computational grids. FTH-B&B, also based on *P2P-B&B*, is an application-level distributed FT mechanism composed of several FT MW-B&Bs organized hierarchically into groups. A fault recovery mechanism is introduced to avoid the loss of work units and to improve efficiency in terms of execution time. Indeed, work units are stored so that one can recover at each instant the subproblems assigned to failed processes using a *3-phase communication protocol*. Moreover, our approach ensures to maintain a balanced and safe hierarchy during the lifetime of the algorithm in order to guarantee a valid functioning. Finally, an efficient restart of the application is ensured by a distributed checkpointing mechanism in case of failure.

All the proposed contributions are implemented using the ProActive grid-middleware [PROA]. These solutions have then been applied to solve exactly Flow-Shop scheduling problem (*FSP*). In most existing works, the grid-based performance evaluation is performed through simulation. The performance evaluation of our contributions is performed on a real experimental grid: the Grid'5000 French nation-wide grid [GRIDa] where up to 1900 grid computing nodes are involved in the experiments. These experiments show the ability of the approaches to deal with scalability and fault tolerance. In fact, the different experiments demonstrate that the approaches scale well and are more efficient compared to the classical MW and single layer HMW-based B&B which is a variant of HMW. Indeed, the approaches allow to speed up the search by minimizing the cost of application deployment, eliminating bottlenecks and minimizing the cost of the decomposition operations. Moreover, the experiments show the ability of FTH-B&B to deal efficiently with FT in large scale environments. Indeed, the overall efficiency of the algorithms reaches easily 98,90% using the FT mechanism. Moreover, the recovered and rescheduled subproblems allow to minimize the redundant work which improves the execution performance. Finally, our approach has proved its ability to avoid orphan branches and to maintain and balance the hierarchy during its lifetime.

This thesis is organized into four chapters: Chapter 1 provides an overview on parallel B&B algorithms and their parallelization strategies. It also presents the concept of Grid Computing, its characteristics and a state-of-the-art on grid-based B&B. Chapter 2 describes our first contribution: *P2P-B&B* framework. Its architecture, its working, and large scale experiments are then detailed. In Chapter 3, we describe the proposed hierarchical B&B (*H-B&B*). We also present, its architecture, work management, exploration strategies, its hierarchical deployment using the ProActive middleware, and large scale experiments on Grid'5000. In Chapter 4, we detail FTH-B&B the proposed fault tolerant B&B. In this chapter, the task recovery, hierarchy maintenance and the distributed checkpointing are described. Finally, we summarize the major achievements of our contributions in this thesis and give some perspectives.

Chapter 1

Parallel Branch and Bound Algorithms Using Grid Computing

1.1 Introduction

Real-world combinatorial Optimization Problems (*COPs*) are CPU time-intensive and require a huge amount of computing resources to be solved optimally. The *Branch and Bound* algorithm (*B&B*) is one of the most efficient methods for exact resolution of COPs. They perform an implicit enumeration of the search space instead of the exhaustive one, so reducing then considerably the computation time required to explore the whole search space. However, this method remains inefficient when large instances of COPs are to be dealt with. To mitigate this constraint, parallelization is one of the most effective ways in terms of improving the computing performances, in particular the use of large scale parallelism provided by the computational grids. In this chapter, we present the B&B algorithms and their parallelization strategies, and overview of Grid computing and the existing grid-based parallel B&B algorithms.

This chapter is organized as follows: Sections (1 and 2) present a state-of-the-art on B&B algorithms, their parallelization, and a short synthesis on the existing classifications. In Section 3, we present an overview of the grid computing, its characteristics, and we position our work in the context of grid computing. We also present the ProActive middleware which is used to develop the frameworks and algorithms we proposed in this thesis. Finally, Section 4 highlights the challenges related to computational grids and summarizes the existing grid-based B&B frameworks and algorithms.

1.2 Sequential B&B

Branch-and-Bound (*B&B*) algorithms are well-known methods for solving to optimality NP-hard optimization problems. This technique proceeds to a partial enumeration of all feasible solutions and returns the guaranteed optimal solution [RLS03, LD60, DAK65, PS98]. Therefore, B&B belongs to implicit enumeration methods for exact resolution of combinatorial optimization problems of type $P : Z(P) = min_{x \in S} f(x)$, where $f : S \rightarrow R, S \subseteq R^n$. P can be solved by enumerating a finite number of elements

in S [GC94, BCG00]. The algorithm decomposes the original problem into subproblems of smaller sizes. In B&B, the search space is organized as a tree of subproblems called search tree. The search tree is explored by dynamically building a tree whose root node designates the original problem. The internal or intermediate nodes represent subproblems obtained by the decomposition of the subproblem associated to their parent. The leaf nodes designate potential solutions or subproblems that can not be decomposed. The construction of the B&B tree and its exploration are performed using four operators: *branching*, *bounding*, *selection* and *elimination* [BCG00]:

- **Branching:** the branching operation, also called decomposition, partitions the feasible domain of a given problem into a number of smaller subsets on which the same optimization problem is defined. Problems are recursively decomposed until either a solution may be easily found, or one determines that further partitioning is unnecessary because the original problem has been solved, or the subproblems resulting from decomposition are infeasible, or one may prove that further branching cannot improve the current best known solution.

- **Bounding:** The bounding operation is used to compute a lower bound on the optimal solution of the subproblem. When a solution is identified, it is compared to this lower bound to decide wether or not it is necessary to continue exploring that branch.

- **Elimination:** The elimination operation is used to determine the nodes to eliminate because they don't lead to the optimal solution. The elimination of a node is based on three steps. Feasibility: a solution is eliminated if it is not feasible. Bound test: the solution is eliminated if it has a lower bound greater than the current upper bound, the upper bound is the best solution found so far. Dominance: the solution is eliminated if the current subproblem is dominated by another one.

- **Selection:** The selection strategy determines the order according to which sub problems are examined that thus determines how the tree is explored. The most used strategies are: best-first (select the subproblem with the lowest lower bound), depth-first (select the most recently generated subproblem).

Sequential B&B algorithms then consist in performing branching and bounding operations, as well as testing the elimination rules using a specific selection strategy.

1.3 Parallel B&B

Although, B&B algorithms showed their efficiency in the literature, they are not sufficient when very large problem instances are to be dealt with. High performance computing such as grid-based parallel computing is required. Therefore, new parallel

versions of B&B have to be designed. The subtrees generated when executing a B&B algorithm can be explored independently simplifying the parallelization of these algorithms. The only global information in the algorithm is the value of the upper bound. To parallelize the algorithm one has to take into account the organization and composition of the target hardware execution environment, the granularity and synchronization of generated tasks, and the communication between different processes [TB92].

1.3.1 Classification of Parallel B&B

The parallelization of B&B is well studied in the literature and many classifications have been conducted [GC94, BCG00, MEL05, CCR06, TRI89, TB92]. In the following we report the different classifications found in the literature.

1.3.1.1 Classification of Trienekens *et al.*

Trienekens *et al.* [TB92, TRI89] have classified parallel B&B into *high level* and *low level* of parallelization according to their degree of parallelization. Therefore parallel B&Bs can be highly or lowly parallelized. This classification is also reported in [CCR06]:

- **Low level:** Only part of the sequential branch and bound algorithm is parallelized in such a way that the interactions between the parallelized part and the other parts of the algorithm do not change. For example, the computation of the bound, the selection of the subproblem to branch from next, or the application of the elimination rule could be performed by several processes in parallel. Because the interactions between the various parts of the algorithm are not changed, low level parallelism does not have consequences for the branch and bound algorithm as a whole. The overall behavior of the thus created parallel branch and bound algorithm resembles the behavior of the original sequential branch and bound algorithm, i.e., the parallel algorithm will branch from the same subproblems in the same order.

- **High Level:** In case of high level parallelization, the effects and consequences of the parallelism introduced are not restricted to a particular part of the branch and bound algorithm, but influence the algorithm as a whole. The thus created parallel algorithm is essentially different. The work performed by the parallel algorithm need not be equal to the work performed by the sequential algorithm. The order in which the work is performed can differ, and it is even possible that some parts of the work performed by the parallel algorithm are not performed by the sequential algorithm, or vice versa. For example, several iterations of the main loop can be performed in parallel (e.g., several processes executing the algorithm branch in parallel from their own subproblem).

1.3.1.2 Classification of Gendron *et al.*

Gendron *et al.* [BCG00, GC94] identified three types of parallel B&Bs according to the degree of parallelism of the search tree:

- **Parallelism of type 1 (node-based):** introduces parallelism when performing the operations on generated subproblems. For instance, it consists of executing the bounding operation in parallel for each subproblem to accelerate the execution. Thus, this type of parallelism has no impact on the general structure of the B&B algorithm and is particular to the problem to be solved.

- **Parallelism of type 2 (tree-based):** consists of building the solution tree in parallel by performing operations on several subproblems simultaneously. Hence, this type of parallelism may affect the design of the algorithm. This type of parallelization is suitable for coarse-grained asynchronous MIMD architectures.

- **Parallelism of type 3 (multi-search):** implies that several solution trees are generated in parallel. The trees are characterized by different operations (branching, bounding, and pruning), and the information generated when building one tree can be used for the construction of another.

1.3.1.3 Classification of Melab

The most recent classification is the one done by Melab in [MEL05] and reported by Mezmaz in [MEZ07]. This classification is based on the classification of Gendron *et al.* [BCG00, GC94]. In this taxonomy, four models of parallel B&B algorithms are identified: *parallel multi-parametric model*, *parallel tree exploration model*, *parallel evaluation of the bounds*, and *parallel evaluation of a single bound*.

- **Parallel multi-parametric model:** consists to consider several coarse-grained B&B algorithms. Each algorithm uses its own parameters and several variants of the algorithm can be obtained by modifying these parameters. Therefore, different B&B algorithms can be obtained and executed in parallel. The algorithms can execute different branching, bounding, selection operations.

- **Parallel tree exploration model:** In this model, different subtrees can be explored in parallel. Therefore, the branching, bounding, selection, and elimination can be executed in synchronous as well in asynchronous way. In the synchronous model, the B&B includes different phases and in each phase the processes explore their subtrees independently before synchronizing themselves. And the asynchronous model, the processes communicate dynamically. Most of existing parallel B&B algorithms belong to this category. The degree of parallelism of

this strategy is considerable and it is suitable for grid environments.

- **Parallel evaluation of the bounds:** This model allows a parallel evaluation of the subproblems generated by the branching operator. This model is exploitable only if the bounds evaluation is entirely executed after the branching operation. However, it is not suitable for grid environments. Indeed, in the case of synchronous model, additional delays are engendered because of the heterogeneous nature of grid resources. In asynchronous model, the evaluation of the bounds can be too fine and then can penalize the system performance.

- **Parallel evaluation of a single bound:** This model does not change the conception of the algorithm because it identical to the sequential version but the evaluation phase is faster. This model depends on the considered problem, synchronous and centralized. It is limited in terms of extensibility, nevertheless, it is efficient when combined with other models.

1.3.1.4 Synthesis

Although the different nominations of the strategies some categories of different classifications converges considerably. For example, in the *parallelism of type 1*, the *low level parallelism*, and the *parallel evaluation of bounds*, the parallelism is done inside one of the four B&B operations (branching, bounding, elimination, and selection). And *parallelism of type 2*, *parallelism of type 3*, *high level of parallelism*, *parallel tree exploration*, and *Parallel multi-parametric model* the parallelism is done at the level of the subtrees. In this thesis, two types of parallel B&Bs are developed: P2PB&B belongs to the second category since the parallelization is performed at subtree level. In the second parallel B&B (H-B&B), the subtrees are built in parallel and the branching operation is also performed in parallel, thus, it belongs to both the first and the second category.

1.3.2 Work pool management

Processes of a parallel B&B pick up and store their work units in a memory location called work pool. Typically, a process picks up a subproblem in a work pool and examines it. When it finishes its action, the process stores the subproblems not yet examined in the same or in a different work pool. Two types of work pools are distinguished: Single work pool and multiple work pool. Single pool algorithms make use of only one memory location, whereas in multiple pool implementations there are several memory locations where processes find and store subproblems yet to be examined.

1.3.2.1 Single work pool

Single work pool algorithms are implemented mainly on shared-memory systems. On message passing architectures, it is possible to implement them by using the Master/Worker paradigm. The master process, which is also the coordinator, executes the B&B search, specifies the tasks to be executed by the other processes, controls the pool of subproblems to be examined, and determines the end of the computations (an empty pool and all processes idle). Worker processes perform bounding and branching operations. Upon completion of this task, workers return the results - the new subproblems or a fathoming message - to the master process. When a worker improves the upper bound, it broadcasts it to the master and all the other workers. The master selects subproblems from the pool and allocates them to idle workers according to a simple round-robin strategy. The selection of subproblems may be based on one of the usual sequential criteria (e.g., depth-first, best-first). In this single-pool strategy, the control of the search is centralized at the level of the master process [BCG00].

1.3.2.2 Multiple work pools

Multiple work pools algorithms can be organized into three possible schemes: collegial, grouped and mixed. In a collegial algorithm, each work pool is associated with exactly one process. This strategy is suitable to hierarchical MW-based algorithms. In a grouped organization, processes are partitioned, and each work pool is associated with a subset of this partition. In a mixed organization, each process has an associated work pool, but there is also a global work pool shared by all processes.

1.4 Grid-Computing

Grid Computing has evolved from earlier developments in parallel, distributed and HPC (High Performance Computing) in terms of hardware and software. It emerged in the early 1990s, when high performance computers were connected by fast data communication with the aim to support calculation and data-intensive scientific applications.

According to the context of the considered work on grids, many definitions have been given [KBM02, GRI02, FOS02, FKT01, BDS03]. In the following we report the most popular ones:

The notion of Grid Computing was first introduced by Foster and Kesselman (1998) [FK98] as: *A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.*

The Globus Project [GLOB] defines the grid as: *and infrastructure that enables the integrated, collaborative use of high-end computers, data bases, networks, and scientific instruments owned and managed by multiple organizations.*

The GirdBus project [GRIDb] is: *a type a parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed dynamically at*

runtime depending on their availability, capability, performance cost, and users' quality-of-service requirements.

From these definitions and within the context of this thesis (combinatorial optimization and large scale computing), we retain the last definition [?] and we consider a computational grid as collections of loosely-coupled, geographically distributed, heterogeneous computing resources that can provide significant computing power over long time periods. Therefore, they are multi-domain, heterogenous, dynamic, and offering resources at large scale.

1.4.1 Characteristics of Grids

Grids are mainly characterized by: their *multiple administrative domains*, their *heterogeneity* and *large scale number*, and *the dynamicity of their resources*, as summarized in [BBL02].

1.4.1.1 Multiple administrative domains

Grid resources are distributed on multiple administrative domains and organizations and each domain may have its own management and security policies. Resources are often protected by firewalls and the users are identified to ensure the security of the resources.

1.4.1.2 Heterogeneity

Sites of a same grid may be owned by different organizations (labs, universities, etc.). Therefore, the same grid gather resources from different hardware vendors, running with different operating systems, and relying on different network protocols. In contrast, each site is usually composed of homogeneous resources, often organized in a single cluster, which provides a high performance environment for computations and communications. Although, the existence of standards like XML and SOAP for resources description and exchange, the developers must take into account the heterogeneity of resources in the development of grid-based applications.

1.4.1.3 Scalability

Grids offer a large number of computational resources. Therefore, developers must deal with the scalability issue to ensure a safe running of grid applications. Moreover, the resources are often geographically distributed. Hence, Grid frameworks have to help applications with scalability issues, such as providing parallelism capabilities for a large number of resources.

1.4.1.4 Dynamicity

The large number of resources that are distributed on different domains implies a high probability of faults, such as hardware failures, network down time, or maintenance.

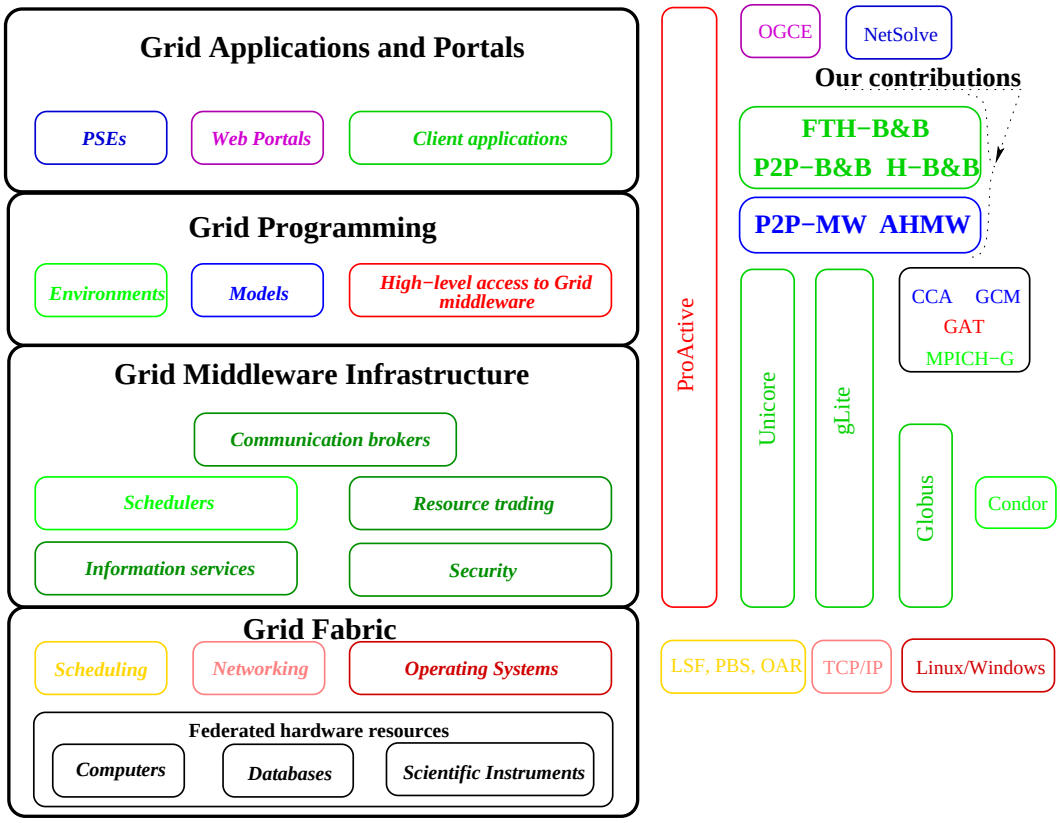

Figure 1.1: Grid Layered Architecture [COS07]

Moreover, the resources can leave and join the grid at any time implying the resources to be highly volatile. This volatility causes issues such as dynamic discovery of resources, fault tolerance, synchronization, etc. These issues are often hard to take into account at middleware level, therefore, the grid developers must deal with the dynamic nature of the resources at application level.

1.4.2 Grid Architecture and positioning

Grid application developers may be classified in three groups, as proposed in [GBF+02]. The most numerous group are end users who build packaged Grid applications by using simple graphical or Web interfaces; the second group is programmers that know how to build a Grid application by composing them from existing application components and Grid services; the third group consists of researchers that build individual components of a distributed application, such as simulation programs or data analysis modules.

Then, from the software point of view, the development and the execution of Grid applications involve four concepts: virtual organizations, programming models, deployment, and execution environments [COS07]. All these concepts may be organized in a layered view of Grid software, shown in Figure 1.1:

- At the bottom, the Grid fabric, which is all resources gathered by the Grid. These

resources consist of computers, databases, sensors, and specialized scientific instruments. They are accessed from operating systems (and virtual machines), network connection protocols (rsh, ssh, etc.), and cluster schedulers (PBS, LSF, OAR, etc.).

- Above, layer 2, is the Grid middleware infrastructure, which offers core services such as remote process management and supervision, information registration and discovery, security and resource reservation. Various frameworks are dedicated to these aspects. Some of them are global frameworks providing most of these services, such as the Globus toolkit [FOS05] and Condor [TTL05].

- Layer 3 is the Grid Programming layer, which includes programming models, tools, and environments. This layer eases interactions between application and middleware, such as Simple API for Grid Applications (SAGA) [GJK05] an initiative of the Global Grid Forum (GGF). This initiative is inspired by the Grid Application Toolkit (GAT) [ALL05] and Globus-COG [LFGL01] which enhances the capabilities of the Globus Toolkit by providing work-flows, control flows and task management at a high level of abstraction.

- The top layer is the Grid Application layer, which contains applications developed using the Grid programming layer. Web portals are also part of this layer, they allow users to control and monitor applications through web applications, such as OGCE [OGCE]. The layer includes Problem Solving Environment (PSE), which are systems that provide all facilities needed to solve a specific class of problems, NetSolve/GridSolve [SNMD02] are complete Grid environment that help programmers for developing PSEs.

Some Grid middlewares cover more than a single layer, it is notably the case for Unicore [UNIC] and gLite [LHPB04], which provide Grid programming features in addition of Grid middleware properties, for instance access to federated Grid resources, with services including resource management, scheduling and security. ProActive is also one of these, we describe in detail ProActive in Section 1.4.3.

The contributions of this thesis belong to the Grid programming and the Grid applications layers. The P2P MW-based B&B framework and the Adaptive Hierarchical MW framework (AHMW) are Grid programming environments and the developed applications: P2P-B&B, H-B&B, and FTH-B&B using P2P MW-B&B framework, respectively AHMW and P2P-MW-B&B frameworks are a Grid Applications. Related work and developed MW and HMW-based B&B algorithms for computational Grids are considered in more details in Section 1.5.

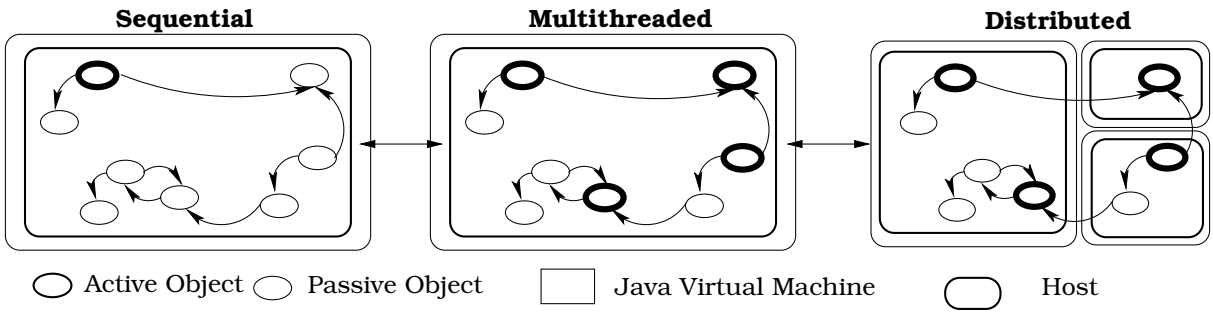

Figure 1.2: Grid Layered Architecture

1.4.3 ProActive: a Grid Middleware

ProActive [PROA] is an open source Java library for Grid computing. It allows concurrent and parallel programming and offers distributed and asynchronous communications, mobility, and a deployment framework. With a small set of primitives, ProActive provides an API allowing the development of parallel applications, which may be deployed on distributed systems and on Grids. The active object model and the ProActive library are used as a basis in this thesis for developing a peer-to-peer infrastructure, a branch-and-bound framework, and performing large-scale experiments.

Active objects model

ProActive is based on the concept of an Active Object *AO*, which is a medium-grained entity with its own configurable activity. A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called active objects (Figure 1.2). Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous with transparent future objects and synchronization is handled by a mechanism known as *wait-by-necessity* [CAR93]. There is a short rendezvous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee [COS07].

The ProActive library

The ProActive library implements the concept of active objects and provides a deployment framework in order to use the resources of a Grid.

Grids gather large amount of heterogeneous resources, different processor architectures and operating systems. In this context, using a language which relies on a virtual machine allows maximum portability. ProActive is developed in Java, a cross-platform language and the compiled application may run on any operating system providing a compatible virtual machine. Moreover, ProActive only relies on standard APIs and does not use any operating-system specific routines, other than to run daemons or to

interact with legacy applications. There are no modifications to the JVM nor to the semantics of the Java language, and the bytecode of the application classes is never modified.

ProActive relies on an extensible meta-object protocol architecture (MOP), which uses reifective techniques in order to abstract the distribution layer, and to offer features such as asynchronism or group communications.

An active object has it own activity thread, which is usually used to pick-up invocations from the request queue and serve them, i.e. execute them.

Active objects are instantiated using the ProActive API, by specifying the class of the root object, the instantiation parameters, and a possible location information:

```
// instantiate active object of class A on node1 (a possibly remote location)
A a = (A) ProActive.newActive("A", new Object[] params, node1);
// use active object as any object of type A
Result r = A.foo();
// possible wait-by-necessity
int b=r.getResult();
```

Communication by messages

In ProActive, the distribution is transparent: invoking methods on remote objects does not require the developer to design remote objects with an explicit remoting mechanism (like Remote interfaces in Java RMI).

Communications between active objects are realized through method invocations, which are passed as messages. These messages are serializable Java objects which may be compared to TCP packets. Indeed, one part of the message contains routing information towards the different elements of the library, and the other part contains the data to be communicated to the called object. Three types of communication are possible: *synchronous invocation*, *one-way asynchronous invocation*, and *asynchronous invocation with future result*.

Synchronous invocation:

The method returns a non reifiable object: primitive type or final class:
public boolean foo()
In this case, the caller thread is blocked until the reified invocation is effectively processed and the eventual result (or exception) is returned.

One-way asynchronous invocation:

The method does not throw any exception and does not return any result:
public void gee()

The invocation is asynchronous and the process flow of the caller continues once the reified invocation has been received by the active object.

Asynchronous invocation with future result:

The return type is a reifiable type, and the method does not throw any exception:
public reifiableType baz()

In this case, a future object is returned and the caller continues its execution flow. The active object will process the reified invocation according to its serving policy, and the future object will then be updated with the value of the result of the method execution. If an invocation from an object A on an active object B triggers another invocation on another active object C, the future result received by A may be updated with another future object. In that case, when the result is available from C, the future of B is automatically updated, and the future object in A is also updated with this result value, through a mechanism called automatic continuation [CDHQ03].

Typed Group Communications

A typed group communication is the local representant of a set of objects distributed on interconnected machines. When a method is called upon a group, the execution environment sends an invocation request of the method on the group members, awaits one or more answers of the members according to the defined policy and returns back the result to the caller.

1.5 Grid-based B&B

In general, several factors impact the executing of parallel B&B on parallel environments: the architecture of the execution environment (SIMD or MIMD), synchronicity of the algorithm (synchronous or asynchronous processes), the granularity of handled tasks (coarse or fine-grained tasks), communication between different processes (shared memory or message passing). Therefore, these issues must be taken into account to achieve better performance.

1.5.1 Grid-based B&B challenges

Grid environments induce new issues to take into account and the classic parallelizations must be rethought to adapt them to these new environments. In this section, we discuss those issues and describe modifications on grid-based B&Bs to take benefit from the advantages of the grids and to deal with these issues so that to increase their performance. In the following, we present issues related to scalability, fault tolerance, and communication cost that are addressed in our work.

1.5.1.1 Scalability

Most of developed parallel B&Bs are based on the master-worker paradigm. The lack of scalability of these implementations comes from the bottleneck created when a single

master process must serve many worker requests. Bottlenecks can be quite serious in the grid computing environment because most of nowadays algorithms aim at exploiting thousands of computing resources, thus thousands of workers served by a single master. There are three ways to increase the efficiency of those algorithms:

- Decrease the task request frequency rate by increasing the granularity of the computation. This can be achieved by handling subtrees instead of handling a fixed number of nodes (see Chapter 2, Section 2.2.2) for our proposed solution about decreasing task request frequency. The grain size can be limited by giving an upper bound on the CPU time (or number of nodes) spent evaluating the subtree.

- Alleviate the master process by assigning to the worker operations such as branching, load balancing, and direct communication between workers to achieve a collaboration work (See also Chapter 2 for direct communication between the workers and Chapter 3 for collaboration between workers to achieve branching and load balancing).

- Develop decentralized or hierarchical B&Bs where multiple masters are considered instead of single one. This pushes the limits of the MW paradigm to support more computational resources (See Chapter 3 for our proposed hierarchical B&B). More details on hierarchical approaches are presented in Section 1.5.2.2 and Section 1.5.2.3.

1.5.1.2 Fault tolerance

Volatility and dynamic nature of grid resources impose to any grid-based B&B to adopt a Fault Tolerance (FT) strategy because no loss of data is tolerated in parallel exact methods designed to solve COPs. In B&B a loss of one or several subproblem(s) can cause the loss of one or several optimal solution(s). Therefore, each parallel B&B designed to be executed in a volatile environment must take into account FT issue. FT can be achieved at the middleware level [PROA, XTRE, CON] or at the application level [MMT07a, MMT07b, IAM00, FM87, GKLY00, GLY00, DVC$^+$09]. FT mechanisms introduced at the middleware level induce additional overheads to the execution time of the algorithms. For instance, the ProActive middleware provides a FT mechanism based on the replication of the processes involved in the computation. This forces the application developers to reserve nodes dedicated to the replicated processes which leads to the loss of computing power. In our case, we focus on FT at the application-level. In the following, we present the state-of-the-art approaches for application-level FT.

FT is relatively easy to take into account in MW paradigm where the master can resubmit the subproblem of the failed worker to another safe worker. However, this solution induce redundant work and then affect the efficiency of the algorithm. Checkpointing mechanisms must be developed. Indeed, the master process must perform periodically regular checkpointing to save the progress of the workers. The master

must also develop a mechanism to detect failures by heart-beating the workers. Nevertheless, these operations introduce new overhead to the master process. In this thesis, the FT is taken into account by resubmitting only a part of the unsolved work to a safe worker (See Chapter 4). In addition to the primary work pool, the master process holds secondary work pools. There are as much secondary work pools as active workers. Each secondary work pool contains the subproblems as decomposed by the workers and then each time a worker solves a subproblem it is updated. This allows the master process to resubmit only the unsolved part of the subproblem of a failed worker instead of the entire subproblem.

1.5.1.3 Communication cost

Grid resources are distributed on different sub-networks with different bandwidth and connected to each other by shared nationwide networks. This induces significant latency in communication between processes of the parallel B&B decreasing considerably their efficiency. Three strategies can be used in order to reduce communication cost [MEZ07]: (1) reduce the communication delay, (2) reduce the number of communicated messages, and (3) reduce the size of the messages. In this thesis, we reduced the number of communicated messages by enabling direct communication between workers reducing then the number of flowing messages in the network infrastructure.

Solving combinatorial optimization problems using large pool of resources is not straightforward. Nevertheless, using communications between distributed processes may increase the computation speedup. However, Grids are not the most adapted environment for communicating because of issues such as heterogeneity, high latency between sites, and scalability.

1.5.2 Grid-based B&B frameworks and applications

As seen previously, scalability and FT are major issues to be tackled when designing a parallel B&B for large scale environments. Executing a parallel B&B on large instances of problems spawn a huge number of work units which must be executed and/or communicated. Consequently, bottlenecks can be created on some parts of the network or some computational resources. Many investigations of parallel B&B for distributed memory systems have adopted the MW paradigm [ACK+02, GLY00]. Although this approach simplifies the management of information and multiple processes and FT can be easily tackled, it is clearly not scalable. Scalability can be improved through a hierarchical or fully decentralized organization of processes [XRL05, EPH00, CBCM07], by varying the size of work units [ANF03, AFO06], or by rethinking the B&B representation and its adaptation to large scale environments [MMT07a, MMT07b]. In the following we present the different developed grid-based B&B algorithms and frameworks according to there architecture and we explain for each approach how it deals with scalability and fault tolerance.

1.5.2.1 MW-based B&B Algorithms

In the literature, most of applications developed for large scale environments are based on the MW paradigm. SETI@home [ACK+02] is one of the first large scale MW-based applications. A central master process distributes computational tasks to workers at the edge of the Internet. When a worker completes its computation, it sends the results to the central master. Linderoth *et al.* have proposed *MW* [GLY00], a framework allowing to design large scale applications according to the MW paradigm. MW uses Condor [CON] as its resource management system and PVM [PL96] as a message passing interface between the master and workers. The master is launched on a dedicated machine and the workers are created on non-dedicated resources from the Condor pool. The Condor system matches user-submitted jobs with idle machines in its pool. Three main classes are defined: MWDriver which corresponds to the master process and contains the control center for distributing tasks to workers, MWTask which represents a task, and MWWorker class which represents a worker which executes tasks assigned to it by the master.

Mezmaz *et al.* [MMT07a, MMT07b] have proposed *B&B@Grid* for large scale B&B algorithm using the MW paradigm. The authors have not made any change on the MW architecture but they have applied adaptation on the B&B. They rethought the representation of the search space and then minimized the quantity of flowing information in the network in order to alleviate the master process. Their approach is based on an efficient coding of work units. A list of subproblems is represented by a unique interval defined by only two integers. *Fold* and *unfold* operators are defined to deduce a search sub-space from an interval and *vice versa*. Therefore, the transferred and stored information in the grid is an interval instead of a list of subproblems. The approach also facilitates and optimizes load balancing, FT, and termination detection.

FT is ensured by a checkpointing mechanism. The optimal solution and the unexplored subproblems are stored as intervals in a backup file. When the master fails, the file is checked and then the unexplored subproblems are deduced using the *folding* operator. Workers update the intervals regularly and inform the master of any new solution. When a worker fails, the last copy of its interval is either fully allocated to another B&B process or shared between several B&B processes. However, this FT mechanism has been only applied to the original B&B@Grid and has not been extended to the P2P distributed version.

The approach has been experimented using the Grid'5000 [GRIDa] French nationwide grid and some academic clusters. The reported results show that the approach scales well and is more efficient compared to the best known approaches in grid-based exact optimization. The approach allowed the first optimal resolution of an instance of 50 jobs and 20 machines of the flow shop scheduling problem (Ta056).

1.5.2.2 HMW-based B&B Algorithms

According to the role of the different processes the Hierarchical Master-Worker paradigm (*HMW*), two layers can be distinguished: a control layer composed of one or more levels

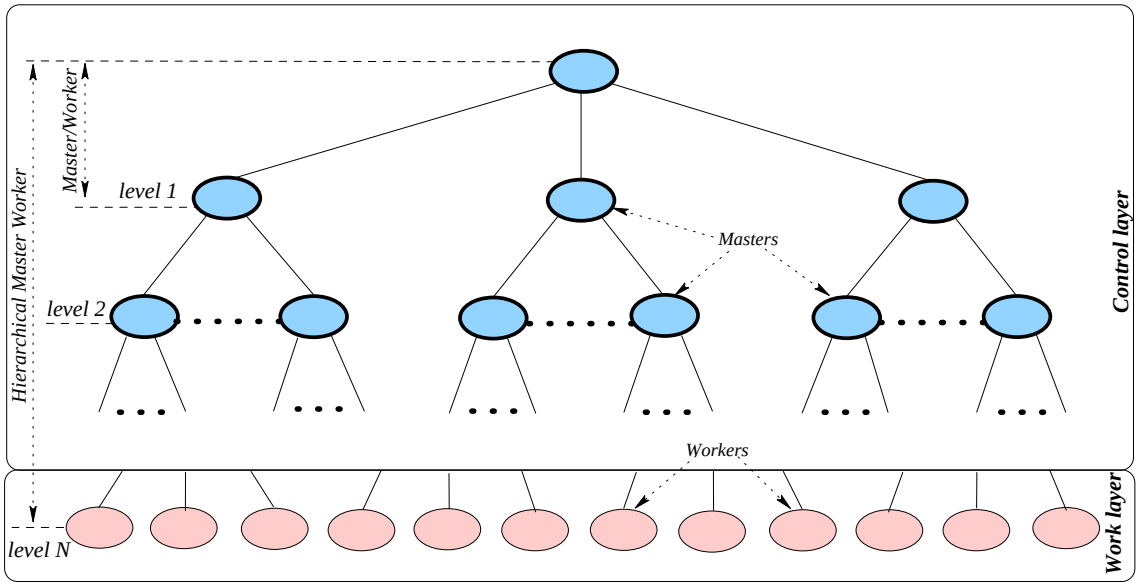

Figure 1.3: HMW classification

of masters and a work layer which is composed of several workers. We can classify existing works according to the number of levels constituting the control layer (See Fig.1.3). A system is considered as HMW when it is composed of, at least, one level of masters. A no-level system is a particular case which represents the classical MW architecture. In the single-level class, the main master only communicates with some sub-masters, and each sub-master manages a set of workers.

Xu *et al.* [XRL05] and Eckstein *et al.* [EPH00] proposed respectively *ALPS* and *PICO* which are parallel B&Bs based on Master-Hub-Worker *MHW* in which a layer of medium-level management is inserted between the master and the workers where each hub manages a static set of workers. They consider in their approaches cluster-based architecture. Each cluster contains a local Hub and one or multiple workers. The number of hubs increases with the number of workers and they avoid becoming overburdened by limiting the number of workers by cluster. Therefore, some computational burden is moved from the master to the hubs. Nevertheless, at the best of our knowledge, no FT mechanism is developed in these two approaches.

Aida *et al.* [ANF03, AFO06] proposed a similar approach of B&B algorithm parallelization using the hierarchical MW paradigm. Their algorithm performs MW at two levels, computing among PC clusters on the Grid and that among computing nodes on each PC cluster. They aim at avoiding performance degradation caused by the communication overhead between the master process and worker processes. In their algorithm, a single supervisor process controls multiple process sets, each of which composed of a master process and worker processes. The master process dispatches subproblems to multiple worker processes, receives back the computed results from the worker processes, and performs load balancing operations with the supervisor. The supervisor process shares the upper-bound and performs a load balancing among master processes

by migrating subproblems among master processes. A worker that receives a task decomposes it, sends the obtained and not-yet computed subproblems to its master, and returns back the result to it. The authors discuss the granularity of tasks, notably when tasks are fine-grained, the communication overhead is too high compared to the computation of tasks. The algorithm has been implemented using GridRPC middleware [SNMD02], Ninf-G [TNS+03], and Ninf [SNS+97].

FT mechanism does not attempt to detect failures of processes and to restore their data, but rather focuses on detecting not yet completed problems knowing completed ones. The approach is based on the fact that the generated subproblems of a B&B are tree nodes. Therefore, subproblems are represented only by their position in the tree. Given a set of tree nodes, its complement can be easily found.

Each process maintains a list of new locally completed subproblems and a table of the completed problems. When a problem is completed, it is included in the local list. After a period of time or after processing a fixed number of subproblems, the list is sent to a set of other processes, selected randomly, as a work report message. When a process receives a work report, it stores it. When a process runs out of work, it chooses an uncompleted problem and solves it.

Di-Costanzo *et al.* [CBCM07] have proposed a four entities-based HMW for B&B algorithms: *master*, *sub-master*, *worker*, and *leader*. The master is the entry point of the system as well as the unique responsible for branching (task decomposition) among other roles such as managing task allocation to sub-masters and/or workers, handling failures and building the different groups of workers. Sub-masters are just intermediary entities whose role is to ensure scalability. They are hierarchically organized and they forward tasks from the master to workers and *vice versa* by returning results to the master (or their sub-master parent). The unique role of workers is to execute tasks. The role of a leader process is to forward messages from one worker to another when they need to communicate. In this work FT is ensured at middleware level using ProActive.

The system has been implemented on top of ProActive [BBC+06, BCM03, CDCL06] and dedicated to solve the permutation Flow-Shop problem. They conducted low (between 20 and 60 CPUs) and medium (up to 700) scale experiments on Grid'5000. The reported results show that the system scales well up to 272 CPUs. However, for more than 272 CPUs, the total execution time decreases slowly.

Di-Costanzo *et al.* have proposed an interesting hierarchical system but in our opinion there are two weak points. First, the sub-masters are under-used and do not participate in the alleviation of the master in terms of execution of tasks. Indeed, the main work of the master is the decomposition and distribution of tasks; the system would be more efficient if sub-masters participate in the decomposition process. Second, the unique criterion of the choice of group members is their localization in the same cluster, which can overload the master when the cluster contains a large number of computational nodes.

Drummond *et al.* [DUGS06] have proposed an FT hierarchical B&B designed to be

run on grid infrastructure applied to the *Steiner* problem in graphs. The branching process of a steiner problem spawns exactly two subproblems. The algorithm is organized as follows: a root master is launched on a process of a given cluster. The root master launches and manages a set of leaders on the first processor of each cluster. Each leader represents the processors of the cluster on which it is launched. A leader process branches a given problem into left and right subproblems and assigns the right subproblem to a leader according to a specific assignment algorithm. The process is repeated until no leader is available, then the branched subproblems are assigned to the processes of the same cluster.

FT is ensured using uncoordinated checkpoints [EAWJ96] where each process keeps the unsolved subproblems and sends checkpoint messages to its neighbor (the neighborhood is defined according to a specific formula). The leaders do the same but they only send checkpoints to the other leaders. The main drawback of the algorithm resides in the number of tolerated failures. Only one failure per cluster can be handled and beyond one failure, the whole cluster is considered as failed. Moreover, if a leader fails, the entire cluster is also considered as failed and no flexibility in this sense is tolerated. The scalability of their algorithm cannot be proven since they only perform experiments using between 16 and 33 computing resources.

Berthold *et al.* [BDLP08] have proposed single and multi-level HMW skeletons to overcome the master bottleneck caused in a simple MW paradigm. In this model, they investigate techniques for hierarchically nesting the basic MW scheme presenting a skeleton implementation for nesting several MW instances. With this scheme the administrative load of task handling has been shifted to the whole hierarchy of masters. The experiments show that, in general, MW hierarchies speed up execution keeping workers busy and avoiding bottlenecks.

An improvement to this work has been made by GhasemiGol *et al.* [GSDB09]. They proposed a linda-based [AND86] HMW to decrease the communication cost. They defined sub-masters as shared spaces (linda tuple spaces) that can be accessed by their own workers. Therefore, several workers can refer to a sub-master concurrently and many communications are eliminated. Nevertheless, both of Berthold *et al.* and GhsemiGol *et al.* have experimented their approaches only on a reduced number of processors (32 for the first and 9 nodes pool for the second). This makes it difficult to conclude on the scalability of their approaches.

Dai *et al.* [DVC+09] have proposed a single-level hierarchical master-worker designed for divisible tasks which is similar to divide and conquer paradigm. In their framework, a main master only communicates with some sub-masters, and each sub-master manages a set of workers using multiple pool collegial strategy. Nodes of the hierarchy are organized into groups. Each group is composed of nodes of the same cluster or LAN and closer to each other to minimize communication latency. A load balancing strategy is proposed since a group of workers may be idle while other groups are very busy. In this case, a sub-master asks its main-master and if this later has no more tasks, it steals tasks from one of the busy groups.

Both the middleware-level and application-level FT mechanisms are used. The

middleware-level mechanism proposed in ProActive [PROA] is used only by the main-master and the authors do not specify if it is also used by sub-masters and workers. When an inner master fails, the challenge is to manage its orphan workers. Two techniques are proposed to address this issue: the election of a sub-master from the remaining sub-workers and the use of redundant sub-workers. The selected sub-workers are a replica of sub-masters, when a sub-master fails they are used to generate a new sub-master.

The main drawback of this approach resides in the use of middleware-level FT and then the definition of redundant processes which will replace the failed ones. Therefore, this approach leads to the loss of computing power. Moreover, no solution is proposed to minimize the redundant work in case of failures.

The model has been experimented on a small cluster of 32 cores. The speed-up ratio between simple MW and single-level HMW is reported. The results show that the speed-up ratio increases as the number of nodes increases. On the other hand, in another experiment, they noticed that their single-level HMW will finally perform as well as the simple MW does in solving divisible tasks when the hierarchical model deals with of a large number of tasks. We notice that the experiments they conducted are not sufficiently large scale, while their model is supposed to scale up the ProActive MW API [PROA].

Even the ProActive MW API is an efficient tool and easy to use for grid applications, no mean is however provided to allow direct communication between workers. This makes the ProActive MW API not suitable to solve problems that need direct communication between the different tasks. As the hierarchical model is based on this API, the processes of the system cannot communicate between them making it unsuitable to solve problems that need direct communication.

1.5.2.3 Decentralized B&B

In addition to these hierarchical schemes, we can find peer-to-peer *P2P* based parallel B&B algorithms which are fully decentralized such as those presented by Finkel *et al.* [FM87] in *DIB* and Iamnitchi *et al.* [IAM00]. Iamnitchi *et al* proposed a fully decentralized parallel B&B algorithm. Each process maintains its local work pool and sends requests to others when it is empty. The process which receives a work request sends a part of its work to the requester. The fully decentralized scheme provides better scalability since no central point is considered. The solution is propagated in circulating the best known solution among processes embedded in most frequently sent messages.

To ensure FT, Iamnitchi *et al.* proposes a coding approach of the subproblems based on their representation according to their position in the B&B tree. The interest of this representation is to deduce a subproblem from its encoding. This approach requires that all subproblems belong to the same subtree to be designated with the same code.

In order to overcome the limits of *B&B@Grid* presented in the previous section in terms of scalability, Djamai *et al.* [DDM11] designed a pure P2P approach for the

algorithm. It provides fully distributed algorithms to deal with B&B mechanisms like work sharing, best upper bound sharing and termination detection. However, the FT is not taken into account in this work.

1.6 Conclusion

Solving to optimality real-world combinatorial optimization problem instances is CPU time-intensive. The Branch-and-Bound (*B&B*) algorithm is one of the most known methods for their exact solving. Nevertheless, such a technique remains insufficient for very large problem instances. To mitigate this constraint, parallelization is one of the most effective ways in terms of improving the computing performances, in particular the use of large scale parallelism based on *Grid Computing*. However, using grid computing is not straightforward and the traditional parallel B&B algorithms must be rethought to meet the characteristics of grids, particularly their large scale and the volatility and heterogeneity of their resources.

In this chapter we provided an overview on the topics related to the context of this thesis, i.e., parallel B&B, grid computing and grid-based B&B algorithms. First, we presented sequential and parallel B&B algorithms. We then summarized the different classifications conducted in the literature. Second, we presented the grid computing concept, the characteristics and architecture of grids, and we positioned our work in the context of grid computing. We also presented the main features of the ProActive middleware we used to develop our contributions. Finally, we highlighted grid-based B&B challenges and we gave a state-of-the-art of the existing grid-based B&B.

In the next three chapters, we will detail the contributions we proposed to meet the requirements of grids: Scalability, heterogeneity, and complexity of grids in Chapters 2 and 3, and fault tolerance in Chpater 4.

Chapter 2

P2PB&B: A P2P MW-based B&B

2.1 Introduction

Large scale environments such as computational grids provide a huge amount of computing resources that can offer the power required by parallel Branch and Bound algorithms *B&B* to solve large Combinatorial Optimization Problems *COP* instances. Grid resources are usually organized in clusters, which are autonomous and managed by different administrative domains, these resources are volatile and heterogenous. Consequently, these characteristics induce new challenges to the classical B&B algorithms such as scalability, fault tolerance, load balancing, communication delays, and programming issues related to the complexity of managing grid resources.

Most of developed parallel B&Bs for large scale environments are based on the Master/Worker paradigm (*MW*) [ACK+02, GLY00]. The MW paradigm consists in defining two entities: a single master and a pool of workers. The master decomposes an initial task into multiple smaller ones and distributes them among the workers. The workers, on their side, perform the execution of the different tasks. After a worker finishes its calculation, it sends back the result to the master and asks for a new task. This simple mechanism makes the MW paradigm widely studied and successfully used for many parallel applications. Consequently, many sequential applications can be easily brought to the MW paradigm since all the algorithm control is done by the master. Indeed, users only have to find a way to suitably decompose the problem to be solved, to distribute tasks, to gather results and to terminate the calculation.

However, the MW paradigm is strongly limited regarding scalability in large scale environments [ANF03, AFO06]. Indeed, the central master process is subject to bottlenecks caused by the *many-to-one* requests submitted by its different workers. This slows down the master in serving the workers and these later in executing their allocated tasks, thus degrading the global performance of the exploration process. Additionally, developing grid-based B&B algorithms is not straightforward for non specialized developers because of the grid complexity. This reduces the size of the community that can develop grid-based applications.

In this chapter, we propose *P2P-B&B* which is a P2P MW-based B&B framework enabling direct communication between workers. The aim of this framework is

twofold: First, it facilitates the development of grid-based B&Bs and hiding the complexity of the grid to the users and non specialized developers. Thus, developers only need to code the algorithms for solving their optimization problems. Second, it deals with scalability, so it offers tools to alleviate the master process by enabling direct communication between workers and minimizing task request frequency. Moreover, the framework is used to develop a complete grid-based parallel B&B for the Flow-Shop scheduling Problem (*FSP*). We also present how to exploit direct communication of the P2P-MW paradigm to achieve better performance and how this framework is used to facilitate the development of a grid-based B&B. This work has been published in [BMT07, BGMM08, BMT09].

The remainder of this chapter is structured as follows: Section 2 presents *P2P-B&B* framework, its architecture and the different components. We also detail the way direct communication between workers are performed and the way the task request frequency towards the master process is minimized. Section 3 highlights the use of this framework to develop a grid-based parallel B&B. In Section 4, we present its implementation concerning deployment, communication, fault tolerance, and work management using ProActive. The experimental environment, large scale deployment and performance evaluation on grid formed and managed by ProActive are presented in Section 5. We conclude this chapter in Section 6.

2.2 P2P MW-based framework for B&B

In this section, we present the proposed (*P2P-B&B*) framework for P2P MW-based B&B solvers. It aims at facilitating the development of grid-based B&Bs to deal with the scalability issue by enabling direct communication between workers and reducing task request frequency towards the master process. Moreover, we present the framework P2P-B&B for P2P MW-based B&B solvers to facilitate the development of grid-based B&Bs.

2.2.1 Communications in Master-Worker

Most of developed MW frameworks are limited in terms of enabling communications between the different workers. They are either allowing communication through the master process such as in [GKLY00, GLY00], or do not allow communication between workers at all such as in the ProActive MW API [CDCL06, PROA] and BOINC [BOI]. In the scope of this thesis we focus on the ProActive MW API which we have used to implement our contributions in this thesis.

The ProActive middleware proposes a Master-Worker API which is an easy to use framework for parallelizing embarrassingly parallel applications. It is destined to solve problems that are easy to segment into a very large number of parallel tasks, and that don't need any communication between those parallel tasks. Using the MW API, all the internal concepts of ProActive are hidden to the user. It allows load-balancing,

tasks scheduling, fault tolerance, and a mechanism for solution gathering.

The usage of the Master-Worker API is simple and consists of four steps:

1. Deployment of the Master-Worker framework.
2. Task definition and submission
3. Results gathering
4. Release of acquired resources

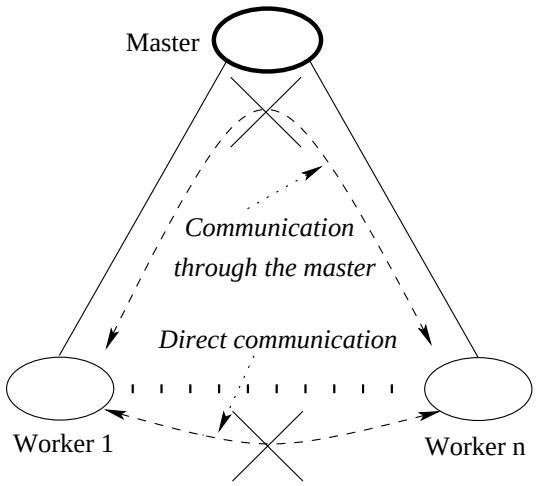

Figure 2.1: ProActive MW Application. No communication are allowed between the workers

However, as the most MW frameworks, this API presents a major drawback. It does not allow communication between the workers neither direct nor through the master process. The absence of communication degrades considerably the performance of the algorithms needing real-time communication such as B&B. In fact, the upper bound must be communicated as soon as possible to prune more branches in the exploration tree and to avoid the exploration of unnecessary branches.

This drawback can be circumvented by emulating communications between the workers. Nevertheless, the emulated communications are delayed and do not happen at real-time. In fact, to communicate between two workers, each of them needs to wait for the termination of the other one and then recovers its result. However, the overhead caused by the waiting of results of other tasks, especially for highly coupled tasks, is another limitation to consider.

In the case of B&B, the upper bound cannot be communicated at real-time, that degrades the system performance because when a worker finds a new upper bound, this latter will be communicated to the master after the worker finishes its current task. Therefore, the whole workers must wait the termination of that worker to recover the upper bound within the result of its task.

Before detailing the framework, we present in the following, our proposition to remedy to this drawback by enabling direct communication between workers.

2.2.2 Direct communication between workers

As mentioned before, the drawback of the ProActive MW API and most of the proposed MW frameworks is that there is no mean to communicate the upper bound at real-time. As a consequence, the system performance is considerably decreased especially when the workers handle coarse-grained tasks, because they take a long time before communicating the new found solution. Moreover, the collaboration capabilities of the developed applications are minimized when a worker takes a long time to respond to other requests coming from other workers for collaboration matter.

The system performance can be improved as the workers handle fine-grained tasks and the upper bound is communicated at real-time. However, handling fine-grained tasks can cause bottlenecks at the level of the master when the workers take a short time to explore their tasks. On the other hand, handling coarse-grained ones can penalize the system and enforce the workers to take a long time to solve their tasks increasing their silence. Therefore, a tradeoff must be found between the size of the tasks and the engendered latency/bottleneck. In fact, the granularity must be fixed in a way to avoid too coarse/fine-grained tasks. Unfortunately, in algorithms such as B&B, the generated exploration tree is highly dynamic and it is hard to fixe a suitable granularity.

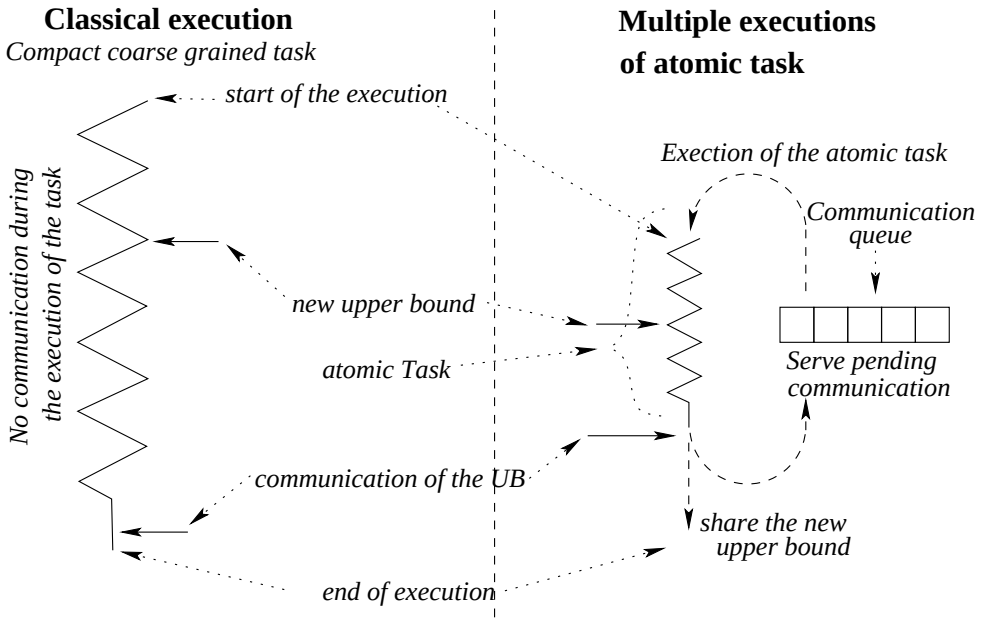

Figure 2.2: Single coarse-grained compact task vs multiple executions of fine-grained atomic task.

To communicate between workers at real-time, we propose to decompose the initial

task assigned to a worker into multiple fine-grained tasks without causing any additional overheads of decomposition and without any additional work request towards the master process. The decomposition is not done explicitly but it is theoretical and no modification is performed on the initial task. We define an *atomic task* which is the smallest task that a worker can perform without any need to perform communication (see Figure 2.2). The workers then, execute multiple smaller tasks and perform multiple communications instead of processing a unique coarse-grained task before communicating once the obtained result and handling once the upcoming communication. The worker then can communicate its new upper bound at any time and can also listen about upcoming communication after each execution of the atomic task.

Figure 2.2 represents a classical execution of a single coarse-grained task versus multiple executions of an atomic task. The process takes a long time executing the coarse-grained task before it communicates its result once after it finishes the task. Whereas, when executing the atomic task, the process can communicate the result and check for pending communication at each iteration. The use of atomic task is explained by the algorithm hereafter:

```
while (!endOfComputation())
.       runAtomicTask()
.       if (needCommunication())
.       .       shareSolution()
.       end if
.       if (newCommunication())
.       .       //routines handling communication
.       end if
end while
```

The decomposition of the initial task is done algorithmically and the master process only sends the initial coarse-grained task. It does not send any information in addition to the initial task. Moreover, there is no additional processing time at the level of the workers except the time of the communication of the obtained result -if it exists- and the time of checking the communication list in the case that another worker sends something to it.

From the point of view of developers, the four methods (*endOfComputation*, *runAtomicTask*, *needCommunication*, and *shareSolution*) must be implemented to adapt the algorithm to the framework design (see Section 2.3).

2.2.3 Architecture and working of the framework

The P2P MW-based framework is situated between the ProActive middleware and the user application layers (see Figure 2.3). It uses features of ProActive to acquire/deploy computing nodes from/on the grid infrastructure. In addition, it uses ProActive typed

group communications to facilitate collaborations and other features to hide the complexity of the grid to the user. It then provides services to the user to develop a communicating B&B without dealing with the grid complexity as well as B&B programming difficulties using the new B&B solver framework (*B&B-Slover*) providing basic routines to develop a generic B&B (see Section 2.2.5).

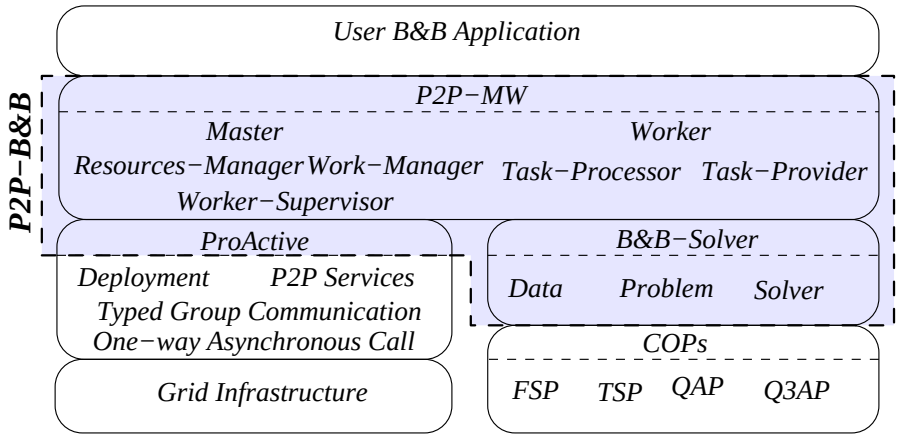

Figure 2.3: Layered stack of P2P-B&B.

Figure 2.4 presents the general architecture of the framework. The master process is composed of one thread (*Work-Manager*) and three active objects (*Resources-Manager*, *Worker-Supervisor*, and *Statistician*). The *Work-Manager* thread allows the master to manage the work pool. It decomposes the initial task into smaller ones, puts them into the master's work pool and provides tasks to the asking workers. The *Resources-Manager* serves to manage the computing resources. It acquires free computing nodes from the computational pool (grid infrastructure) and deploys the workers. It also gathers the workers into communicating groups. The *Worker-Supervisor* allows the master to detect any failure or disconnection among the workers. It sends periodically heartbeats to the workers and waits for their responses. If a worker does not respond then it is marked as failed. The *Worker-Supervisor* informs the master process about the failed worker in order to recover its task and to reassign it to another free worker (see Section 2.4.5). Finally, the *Statistician* allows to recover and report statistics during the execution of the framework such as the processors' load, efficiency, work progress, number of launched and failed workers, communication load, etc.

The worker is composed of three threads (*Task-Provider*, *Task-Processor*, and *Statistician*). The *Task-Processor* thread is the most important process, it allows to execute the task assigned to the current worker. *Task-Provider* provides the worker in terms of tasks by asking the master process. It works in mutual exclusion with the *Task-Processor*, it wakes up the *Task-Processor* when a new task is acquired and then sleeps waiting for task request event. The *Task-Processor* on its side, wakes up the *Task-Provider* after it finishes its task and then sleeps waiting for new tasks. The *Statistician* thread periodically sends statistics to the master process. It interacts with the local threads (*Task-Processor* and *Task-Provider*) and with the master's *Statistician*.

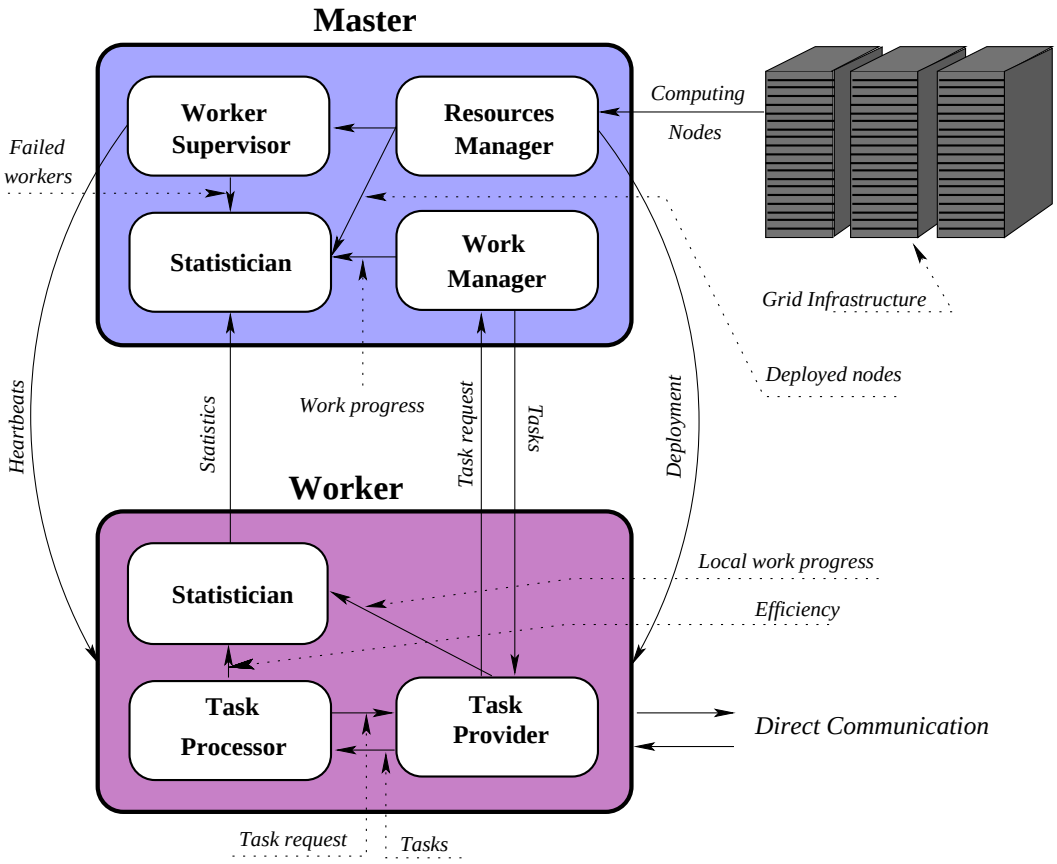

Figure 2.4: General architecture and interactions between the components of the framework.

It collects statistics concerning the state of the worker (in execution or waiting state) and the task work progress and sends them to the master.

2.2.4 The P2P MW-based framework (*P2P-B&B*)

Direct communication imposes new requirements to the classical MW paradigm to adapt the developed applications. Therefore, it is necessary to rethink the classical MW and to design new routines allowing MW applications to behave as P2P ones.

The designed P2P-based Maser/Worker framework allows direct communication between workers and facilitates deployment of P2P applications. The framework (Figure 2.5) uses the functionalities of ProActive and is essentially composed of four entities: *P2PMaster*, *P2PWorker*, *Resources-Manager*, and *Worker-Supervisor*. *P2PMaster* respectively *P2PWorker* are two Java interfaces proposing the most important routines that a Master, respectively a Worker have to implement in order to respond to the new design requirements. The Master and the Worker classes are also Active Objects *AOs* to allow them to benefit from the ProActive features.

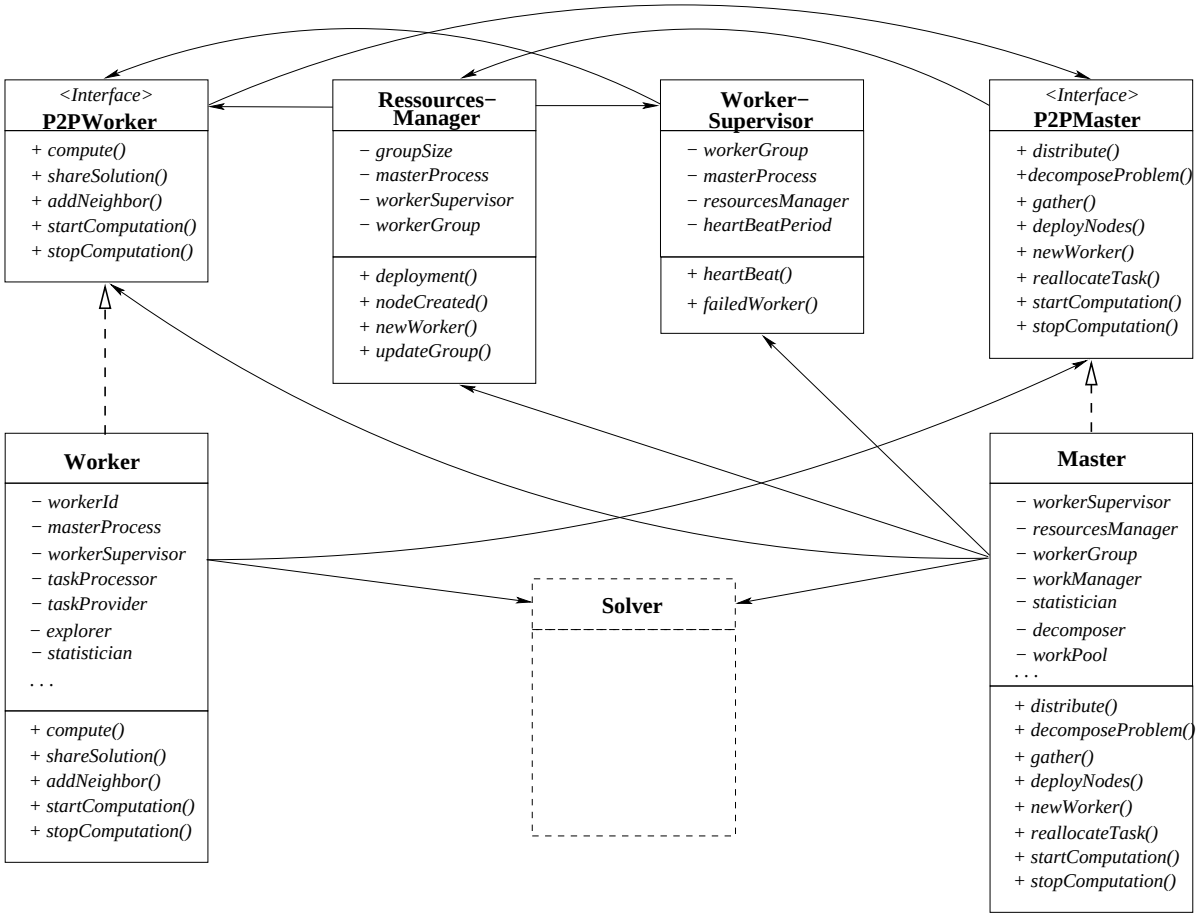

Figure 2.5: P2P MW-based framework class diagram

2.2.4.1 P2PMaster interface

The P2PMaster interface proposes methods allowing a master process to manage the workers (connection, disconnection, deployment, and failure detection). In addition, it includes methods to manage the work (work decomposition, task distribution, result recovering, and task reallocation). The Java source code of the interface is given bellow:

```
public interface P2PMaster {
.     Problem distribute(P2PWorker askingWorker);
.     Vector <Problem> decomposeProblem();
.     void gather(Problem newSolution,String workerWhichHasFoundTheSolution);
.     void deployNodes();
.     String newWorker(P2PWorker newWorker);
.     void reallocateTask(Problem subProblemToRealocate);
.     void startComputation();
.     void stopComputation();
}
```

- *decomposeProblem*: It decomposes the initial problem into smaller independent subproblems. These subproblems are dispatched among the workers that execute them independently. The Master stores the obtained subproblems into its work pool. This operation is performed by the *Work-Manager* thread.
- *distribute*: It distributes the sub-tasks among the workers. This method can be executed in parallel with *decomposeProblem* when the master takes a long time in the decomposition process. Therefore, the overall waiting time of the workers is minimized and thus the execution time is reduced (see Section 2.4.2 for more details on the distribution process).
- *gather*: It is used to gather results coming from the workers. This method also can be executed in parallel with *decomposeProblem* and *distribute* when a worker finishes its calculation part and provides a result before the master finishes the decomposition and the distribution processes. Unlike all existing MW-based solvers, *gather* is not only called at the end of the worker execution but rather called by a worker each time it finds a new solution in order to inform the Master process about it in real-time.
- *startComputation* and *stopComputation*: These two methods start and stop the computing of a group of workers by sending a *start* or *stop* message to participants.
- *deployNodes*: It recovers P2P nodes (Workers) from the P2P network managed by ProActive. These P2P nodes are transformed into computing resources and then independent workers are created (see Section 2.4.1 for the deployment).
- *newWorker*: It manages connections of new workers wishing to participate in the computation (see 2.4.4 for the management of new connections).
- *reallocateTask*: It is used to reassign a task to another worker when the worker it is assigned to fails (see Section. 2.4.5).

2.2.4.2 P2PWorker Interface

P2PWorker interface offers methods that allow the workers to process their tasks, to collaborate, and to behave as peers in a P2P environment, i.e., to communicate directly between them without an intermediary. Therefore, the workers can normalize with the new feature of the proposed communicating framework. The interface is defined bellow:

```
public interface P2PWorker {
.     void compute();
.     void shareSolution(Integer i);
.     void addNeighbor(P2PWorker worker);
.     void startComputation();
.     void stopComputation();
}
```

- *compute*: It is used when a worker receives a task from the master and stops when it receives *stop* message.
- *shareSolution*: This operation is necessary for an efficient collaboration between the workers which share their best known solutions without flowing via the master. For example, in the case of a B&B algorithm, they share their upper bounds.
- *star/stopComputaion*: They are remotely called by the master to start or stop the computation of the worker.

2.2.5 B&B-Solver: a MW-based B&B Solver for COPs

To facilitate the development of any B&B solver, we have developed the *B&B-Solver* framework (see Figure 2.6). It allows an easy implementation of solvers on specific combinatorial optimization problems using tree-based search algorithms such as *Branch and Bound/Price/Cut* and in general *Divide and Conquer* algorithms. Nevertheless, in this work we only focus on B&B solvers.

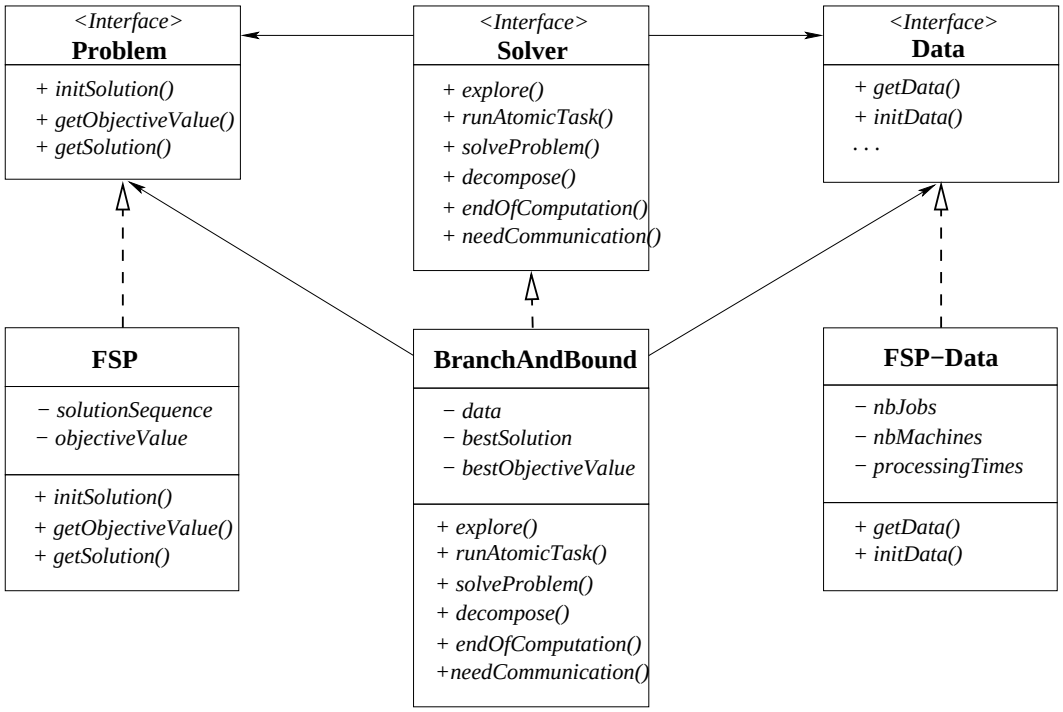

Figure 2.6: Class diagram of the *B&B-Solver* framework

Three main classes are defined: *Data*, *Problem*, and *Solver*. The class *Data* contains essential methods to extract data from benchmarks and to initialize them and the data of the considered problem. For example, in the case of Flow-Shop Problem *FSP*, the data will contain the matrix of the processing times of the different tasks on the different machines. The class *Problem* represents the problem to be solved. It contains the

necessary methods to model the considered problem. It also contains the cost of the best solution of the subproblem. For example in the case of an FSP, a solution is modeled as a vector of integers representing the order sequence of the tasks on the machines.

The *Solver* (see the Java code bellow) interface is the most important one. It contains the methods that a developer should implement to design a solver algorithm responding to the requirements of a communicating environment. Before implementing the solver algorithm, the developer has to model his/her problem first as an optimization problem having a solution and an objective function. After that, he/she has to develop his/her own solver algorithm according to the *Solver* interface requirements. Indeed, he/she must decide how to branch a specific problem. How to evaluate a problem. How to define the bounding (in the case of B&B). How to decide that a specific information is global (to be communicated). Finally, how to decide that a problem is already solved.

Both the master and the workers use this interface to decompose/explore the search space. *Solver* includes all methods required to allow direct communication between workers according to the solution proposed in Section 2.2.2. Therefore, the user must implement the methods: *explore()*, *runAtomicTask()*, *solveProblem()*, *decomposeProblem()*, *endOfComputation()*, and *needCommunication()*.

```
package bb-solver;
public interface Solver extends Serializable {
.       void explore();
.       boolean endOfComputation();
.       void runAtomicTask();
.       void initSolver();
.       void solveProblem(Solution task);
.       Problem getResult();
.       boolean needCommunication();
.       boolean endOfComputation();
.       Vector decomposeProblem(int level);
.       long evaluateSolution();
.       // ...
}
```

The *explore* method serves to define the way the *Solver* algorithm explores the considered problem. *endOfComputation()* indicates when the solver decides if the considered problem is already solved in order to stop the worker executing the solver. *runAtomicTask()* serves to hold the part of the code of the elementary task of the problem. Compared to existing frameworks, this method is new. It allows the worker executing it to avoid taking long time executing coarse-grained tasks. More details on the atomic task are done in Section 2.2.2. Using *initSolver()* the developer can perform some initializations to improve the initial upper bound before beginning the calculation. *solveProblem(Problem problem)* allows to assign the problem to solve to the solver. *decomposeProblem()* serves to decompose an initial problem. In the point of view of the

master process, each task is a subproblem which is represented by the class *Problem*, therefore, the master process contains a vector of classes that implement the *Problem* interface.

2.3 A Parallel P2P-based B&B using P2P-B&B

In the following, we present the proposed parallel B&B algorithm to deal with scalability in large scale environments such as computational grids. It belongs to *type 2* of Gendron *et al.* classification [BCG00, GC94]. It is a Master/Worker-based algorithm allowing direct communications between the different processes, i.e., (worker ↔ worker and worker ↔ master) using the proposed approach in Section 2.2.2. This helps to alleviate the master process and then avoid the creation of bottlenecks at the level of the master. The master divides the initial problem into a set of subproblems. A single work pool is available at the level of master which distributes the tasks among workers and waits for results of each one of them.

In the following, the Java source code of the main methods of a generic B&B. Note that the code is simplified and some parts are omitted to facilitate its assimilation.

```
package BB-Solver;
public class BranchAndBound implements Solver, Serializable{
.     private Data data; //Data of the problem
.     private Problem partialSolution;// Inner node in the search tree
.     private Problem bestSolution;
.     private int bestObjectiveValue;
.     public Stack nodesToExplore; //Priority based stack
.     public BranchAndBound(Data data){
.     .     this.data=data;
.     .     //...
.     }
.     public void solveProblem(Problem problem){
.     .     this.partialSolution=problem;
.     .     bestObjectiveValue=evaluateSolution(partialSoluton);
.     .     partialSolution.setObjectiveValue(bestObjectiveValue);
.     .     nodesToExplore=new Stack();
.     .     nodesToExplore.push(partialSolution); //push the root problem
.     .     //...
.     }
.     public boolean endOfComputation(){ //Termination condition
.     .     return nodesToExplore.empty();
.     }
.     public boolean needCommunication(){
.     .     return needToShareSolution;//This variable is modified when
.     }                                  //a new solution is found
```

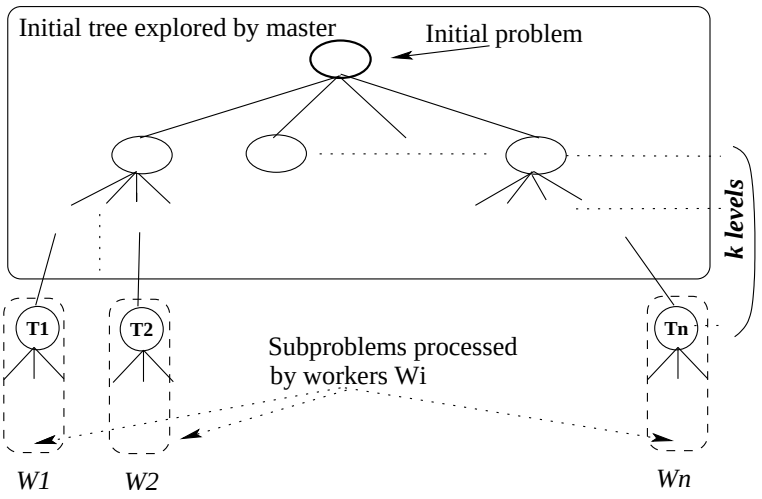

Figure 2.7: General scheme of ParallelBB

The implementation of *runAtomicTask* respectively *decomposeProblem* are given in Sections 2.2.2 and 2.3.1, respectively. In the following, we present the principal operations of the algorithm.

2.3.1 Branching

The branching operation is performed by the master process. It builds its own work pool by performing a breadth first exploration of the initial tree. The size of the work pool depends on the number of explored levels in the tree (Figure 2.7). Let k be the number of explored levels, n the size of the master's work pool, N the initial size of the problem: $n \leq \prod_{0 \leq i \leq k} (N - i)$. The definition of N depends on the considered problem. For example, in the case of an FSP, N is the number of not yet scheduled tasks.

$T_1, T_2, ...T_n$, in the figure represent subproblems (subtrees), each one contains a partial solution having a size equal to the current level in the tree. k is a parameter which depends on two important factors: the size of the considered problem and the size of the computational pool. k must be sufficiently great to generate large number of subproblems which will be processed in parallel by the workers. Therefore, the subproblems must have an acceptable granularity to be performed by a single worker. k also depends on the size of the computational pool. In our case, k depends on the number of available workers. If there is few workers, it is more interesting to have a reduced number of parallel tasks to avoid loosing more time in distribution of tasks.

```
.    public Vector decomposeProblem(){
.    .      Vector listOfTasks=breadthExploration(k);
.    .      return listOfTasks;
.    }
```

The attribution of tasks to workers is performed by the master process. If the number of workers is greater than the number of tasks, the master only considers the workers it needs. Otherwise, it will make a redistribution of tasks to each new available worker. A worker is said available when it finishes its part of calculation.

2.3.2 Selection and Elimination

The elimination operation is only used to eliminate subtrees whose roots have a lower bound greater than or equal to the upper bound. Tree exploration policies used by the master and the workers are different. The master performs a breadth first exploration on the initial tree in order to build subtrees and to prepare the work pool. The master explores nodes by priority to the most promising nodes, i.e., the nodes having a lower bound less than or equal to the upper bound found so far by other workers. These subproblems are stored in a priority-based queue work pool.

The workers implement *runAtomicTask* and perform a *Best First Search (BFS)* selection policy. They use a priority-based stack with opposite priority stacking of subproblems according to the least promising, i.e., at the top of the stack we find the most promising nodes. Therefore, the most promising nodes are explored first.

An example of *runAtomicTask* Java source code is given in the following:

```
public void runAtomicTask(){
.    needToShareSolution=false;
.    partialSolution=new FSP((FSP)nodesToExplore.pop());
.    childrenList=generateChildren(partialSolution);
.    for (int i=childrenList.size()-1;i>=0;i- -){
.    .    partialSolution=(FSP)childrenList.get(i);
.    .    if (partialSolution.size()<data.size()){ //inner node
.    .    .    nodesToExplore.push(partialSolution);
.    .    }
.    .    else{ //leaf node
.    .    .    partialSolution.evaluateSolution();
.    .    .    if (partialSolution.getObjectiveValue()
.                                <bestObjectiveValue){ //new upper bound
.    .    .    .    bestObjectiveValue=partialSolution.getObjectiveValue();
.    .    .    .    bestSolution=partialSolution;
.    .    .    .    needToShareSolution=true;
.    .    .    }
.    .    }
.    }
}
```

The use of atomic task is not static and does not give the same result for all problems and for all exploration methods. The user must take into account the resolution method in the implementation of the *runAtomicTask* method of the *Solver* interface.

For instance, in the case of the B&B algorithm, the result of one execution of *runAtomicTask()* is shown in Figure 2.8. The result of one execution of *runAtomicTask* is represented by a quorum in the search tree.

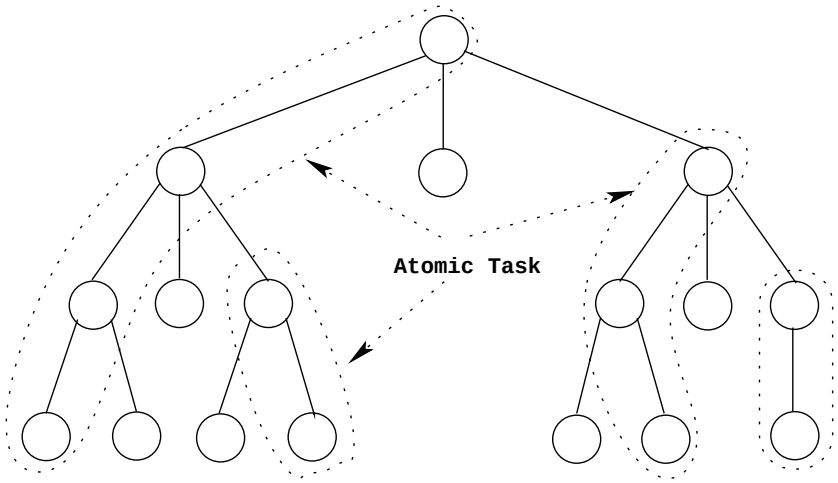

Figure 2.8: runAtomicTask and decompose methods tree exploration

B&B-Solver is used by both of the Master and the Worker classes. The master uses it to decompose the initial task and the workers use it to explore their tasks.

2.3.3 Communication and knowledge sharing

The global knowledge related to the upper bound is increased and updated each time a given worker finds a new upper bound. This operation is performed by communicating the upper bound to other workers. The collaborative work between workers obtained by the sharing of the upper bound, allows to gain much in computation time. Several branches can be eliminated more quickly than in a traditional B&B (sequential B&B) without exploring them, quite simply by consulting the best solution found so far. Unlike traditional B&B, where the upper bound is known only when the exploration process reaches the current node. By using this algorithm, a significant number of branches can be eliminated. These branches cannot be pruned in a sequential B&B because the upper bound making it possible can be found only in the future, i.e., this solution is situated in a search space which will be explored only later.

In Figure 2.9, the upper bound (solution S^*) is found by worker W_3. This solution belongs to the future search space compared to the search spaces of W_1 and W_2. When W_3 sends the upper bound S^* to W_1 and W_2, it allows them to eliminate the branches: (2 and 3.1) in the subtree of W_1 and (1.1, 2 and 3) in the subtree of W_2. Without multiple execution of atomic task and then enabling real-time direct communication, S^* could not be found by W_1 and W_2 at real-time.

The master increases also the workers knowledge related to all other workers executing in the system (dynamic management). The different types of communications

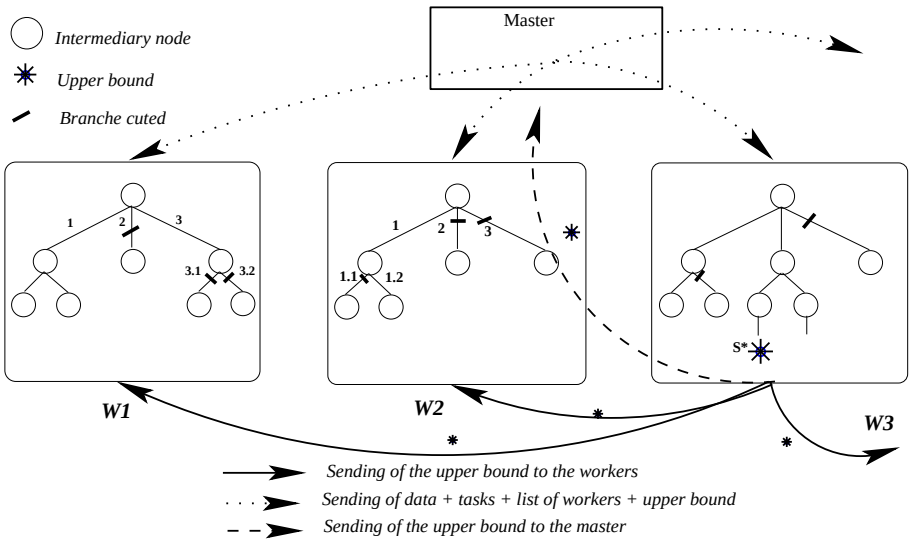

Figure 2.9: Communications between processes of the algorithm

are summarized in the following (see Figure 2.9):

- *Master to Worker*: The master sends the task to execute (data of the problem and the subtree to explore). Given that the algorithm is dedicated to large instances, the exploration time is more important than the sending of the instance itself. It, also sends the pool of executing workers. This information allows each worker to know its environment concerning other workers in progress for a collaborative work. Finally, it sends the initial Upper Bound (UB) to each newly created worker to eliminate branches from the beginning of the search space.

- *Worker to Master*: The unique information that a worker sends to the master is the UB. Each time a worker finds an UB which is better than the current global UB of the algorithm it sends it to the Master. This allows the master to improve the knowledge of the future workers with this upper bound.

- *Worker to Worker*: Each worker sends the upper bound to all the workers in its communication window (workers in progress) so that these workers will be able to reduce the search space by eliminating a great number of branches. The communication window of a worker is reduced to its neighbors, i.e., a worker only communicates with the workers in progress (its neighbors).

2.3.4 Application to the Flow-Shop Scheduling Problem

The general Flow-Shop problem (FSP) can be formulated as follows [LLK78]: FSP consists in scheduling a pool of n jobs on a set of m machines such that each of the

jobs J_1, J_2, ..., J_n has to be processed on the machines M_1, M_2, ..., M_m in that order. Job J_i $(i = 1, 2, ..., n)$ consists therefore of a sequence of m operations O_{i1}, O_{i2}, ... O_{im}; O_{ik} being the processing of J_i on M_k during an uninterrupted processing time p_{ik}. M_k $(k = 1, 2, .., m)$ can handle at most one job at time. The objective is to find a processing order on each M_k such that the time required to complete all jobs is minimized. If the problem is restricted to the minimization over all permutation schedules, meaning with the same processing order on each machine, the resulting problem is called the permutation Flow-Shop problem, which is the focus of our work. Figure 2.10 shows an example of an FSP instance (with $n = 3$ and $m = 4$) and its associated optimal solution.

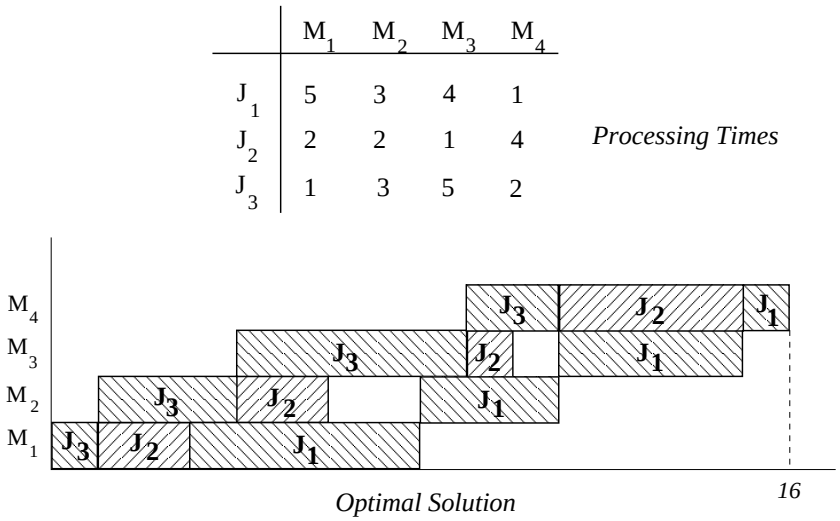

Figure 2.10: Illustration of a permutation FSP with $n = 3$ and $m = 4$. The table reports the processing times of the jobs on the machines. The Gantt diagram shows the optimal solution to the problem instance.

For $m = 2$, an optimal schedule can be found in $O(n.logn)$ steps using Johnson's algorithm [JOH54]. For $m \geq 3$, the problem has been shown to be NP-complete [GAR79]. Due to such complexity, the enumerative solution approach provided in B&B algorithms is well-suited to solve the problem to optimality. As illustrated in Figure 2.11 using the example above, the B&B enumeration scheme is based on a search tree whose root node contains the original problem (empty schedule).

The decomposition of this problem generates n sons, each of them designates a subproblem. The son number i represents the subproblem in which job J_i is scheduled first on all machines. The recursive application of the decomposition operator on the generated subproblems allows to develop the search tree.

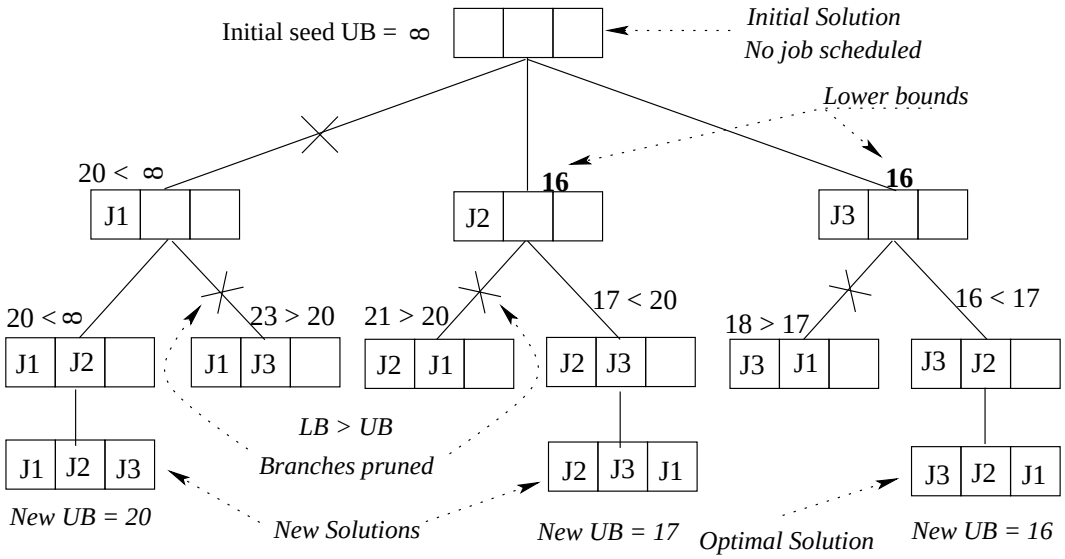

Figure 2.11: The search tree generated and explored by a B&B algorithm for solving an FSP with 3 jobs. Nodes with a lower bound (LB) greater (resp. lower or equal) than the upper bound (UB) are pruned (resp. decomposed or branched).

2.4 P2P implementation using ProActive

2.4.1 Deployment

The deployment of grid application is challenging because of the complexity, diversity, and the huge number of computing resources offered by grid environments. It is usually done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers and clusters. Therefore, the deployment of applications is another issue to deal with when developing parallel applications. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code in order to gain in flexibility [BCMH02].

Using ProActive deployment features the *Resources-Manager* hides all details about the grid infrastructure to deploy applications. It implements *deployNodes()* method to provide the application by virtual nodes (VNs) that correspond to JVMs which contain active objects. In our case, the active objects are the workers, so they are remotely created and activated on the corresponding nodes to receive calculation:

```
.
ResourcesManager resourcesManager;
. . .
resourcesManager.deployNodes(xmlDescriptor);
.
```

xmlDescriptor is a XML descriptor file where ProActive finds the computing resources. The user of the application has just to specify where the application will find these resources in the file.

The *Resources-Manager* also implements *nodeCreated()* method of ProActive to acquire P2P connections. In this case, a daemon *p2pService* is launched on all the hosts wishing to participate in the calculation. When the *Resources-Manager* detects a new connection of a worker, it remotely calls the *newWorker()* method on the master process (see Section 2.4.4 for new connections).

2.4.2 Task distribution

After initializing the workers, the master generates a set of independent tasks by implementing *decomposeProblem()*:

```
private BranchAndBound decomposer;
. . .
decomposer=new BranchAndBound(data);
. . .
public Vector <Problem> decomposeProblem(){
.       Vector <Problem> workPool=decomposer.decomposeProblem(k);
.       return workPool;
}
```

These tasks are stored into the local work pool and are represented by the *Problem* interface (see Figure 2.12). The master selects tasks from the work pool according to the used selection policy and implements the *distribute* method which is launched in parallel with *decomposeProblem*. *distribute* is remotely called by workers:

```
//At the level of a worker:
P2PMaster masterProcess; //
Problem localTask=masterProcess.distribute();
. . .
//and at the level of the master:

public Problem distribute(){
.       return workPool.getNextTask();
}
```

Before the master sends a task to the workers, it increases their knowledge concerning their environment. This knowledge concerns the set of workers executing other tasks and the best solution found so far. This operation is done by the *Resources-Manager* active object when it creates the worker. The master activates the calculation by *startComputation()* and the workers launch the computing method using *compute()* to

explore their tasks:

```
.
//At the level of a worker
private BranchAndBound explorer;// used to explore a problem
. . .
explorer=new BranchAndBound(data);
. . .
public void compute(){
.       explorer.initSolver(bestObjectiveValue);
.       explorer.solveProblem(localTask);
.       while (!explorer.endOfComputation()){
.               explorer.runAtomicTask();
.               if (explorer.needCommunication())
.                       shareSolution(explorer.getBestObjectiveValue());
.       }
}
. . .
public void shareSolution(int bestObjectiveValue){
.        masterProcess.recover(bestObjectiveValue);
}
.
```

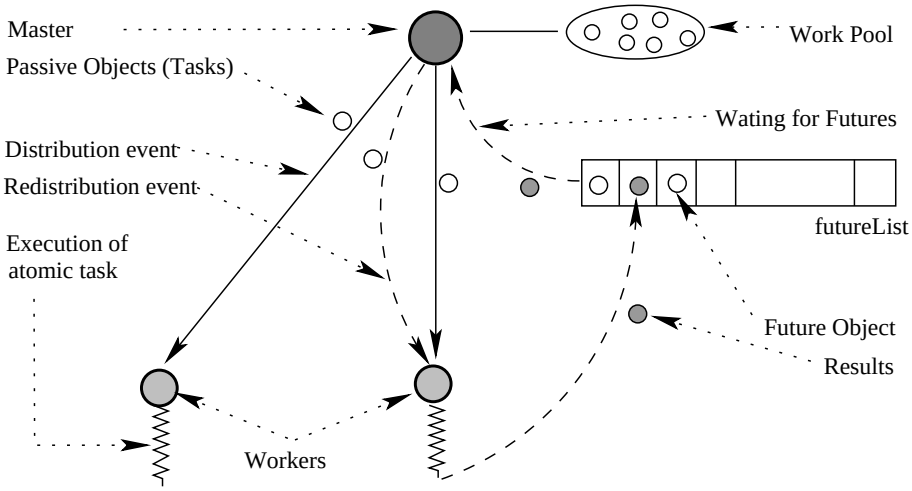

Figure 2.12: Tasks distribution on workers

The master implements *recover* to recover results coming from the workers. Each time a task is assigned to a worker, a future object is created and added to a list of future objects named *futureList*. The master waits for all future objects coming from the workers appearing in its list. A future of a worker designates the calculation result of its task that the master assigned to it. This is accomplished by listening of any response (computation result) in the future. The listening is of type *wait for any event* made by the method *waitForAny(futureList)* and is accomplished by waiting for any

event coming from workers appearing in the list *futureList*. The event is activated at each termination of a task processing. The master creates and reallocates a new task through *Problem*.

Figure 2.12 does not represent any chronology of events, because all operations are made in an asynchronous way, i.e., an event of type redistribution can arrive before other events of type distribution when one of the workers returns back a result before the master finishes the distribution of all tasks. We can see well on Figure 2.13 an example of a sequence diagram of the chronology of all task distribution events using two workers. Operations (1), (2), (7) and (10) are for task distribution. (3) and (4) are for the sharing of upper bounds between the workers. Note that the upper bound can be communicated at real-time. (5), (8) and (11) represent delay time of the master waiting for future objects. (6), (9) and (12) are results coming from the workers. Finally, the master stops computing when it sends *stopComputation* in (13).

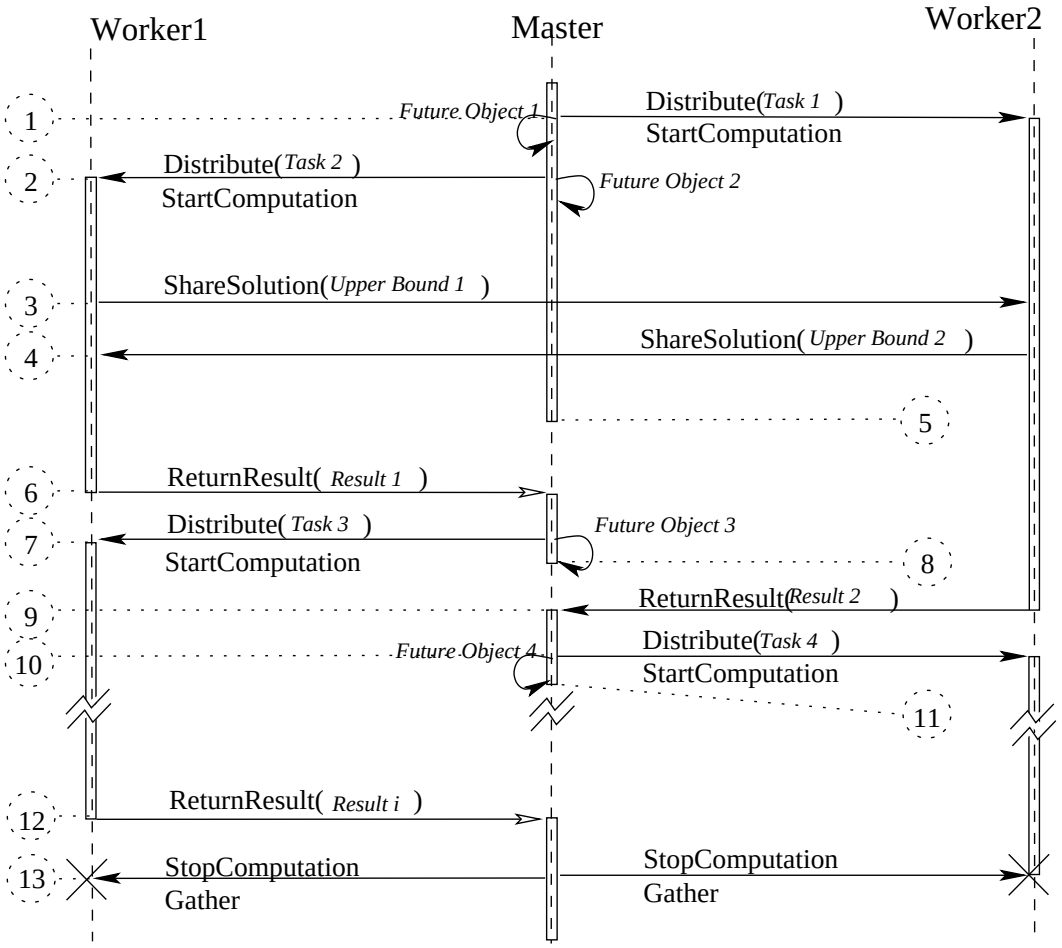

Figure 2.13: Sequence diagram of tasks distribution

2.4.3 Group Communications

We have seen previously that communication between different components is very important for an efficient functioning and workers frequently communicate to ensure the freshness of the upper bound. The use of classical communication between the workers, i.e., by sending one message for each worker, is not efficient in this type of application where the communication cost is very high.

We exploited the typed group communication provided by ProActive and one-way non blocking and asynchronous invocation methods. *workerGroup* is created, it is also an active object and it is a local representant of a set of workers participating in the computation. When a worker wants to send a message to its colleagues, it passes by this typed group, which implements the same communication method *shareSolution()* implemented on the whole workers. The *workerGroup* calls this same method on the set of workers that it represents, using multicast features. When a new solution is found, few messages are sent. A worker sends only one message to all participants of a typed group communication.

2.4.4 Management of new connections

The grid resources are often dynamic and volatile; they frequently join and leave the system. ProActive offers peer to peer framework to discover and to acquire these resources. To discover these resources, the master implements *newWorker* method of P2PMaster which implements *nodeCreated* interface of ProActive. This interface, creates a listener which listens for possible peer connections (see Figure 2.14). In the P2P infrastructure, a P2P daemon is launched on each host participating in the computation. When a new node is detected, an active node is created there and a worker is established. The master adds this new worker to the corresponding typed group communication (*workerGroup*) through the *Resources-Manager*. Therefore, this worker will be able to receive the upper bound value at real-time. On the other hand, other workers can know about the progress of this new worker and the solutions obtained by this worker.

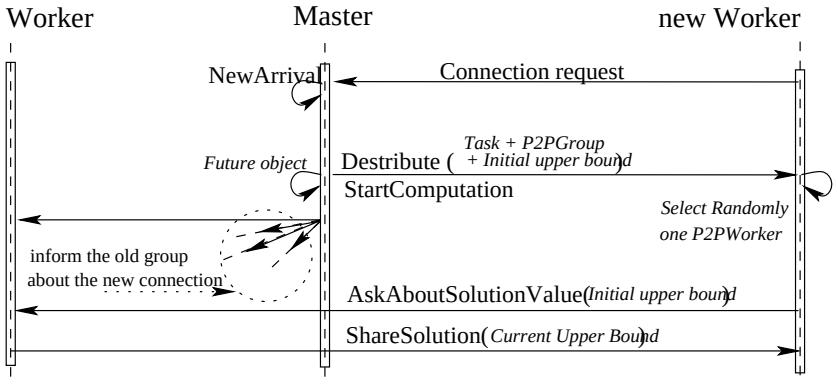

Figure 2.14: Sequence diagram of new connections

The workers forming the old group update their group by adding this new worker. This operation is managed by the *Resources-Manager* which asks the members of the old group to add this new worker by remotely calling *addNeighbor*.

```
P2PWorker workerGroup;
. . .
workerGroup.addNeighbor(newWorker);
```

2.4.5 Fault Tolerance

The fault tolerance issue is taken into account by both ProActive (middleware-level) and the P2P MW-based framework (application-level). With ProActive two types of servers are created: *Resource server* and *Fault tolerance servers*.

The *resource server* returns a free node that can host the recovered active object; this server can store free nodes by two different ways:

- at deployment time: the user can specify in the deployment descriptor a resource virtual node. Each node mapped on this virtual node will automatically register itself as free node at the specified resource server.
- at execution time: the resource server can use an underlying P2P network (see [PROA]) to reclaim free nodes when a hosting node is needed.

Fault tolerance servers are used for checkpointing operations, the localization of AOs, and failure detection.

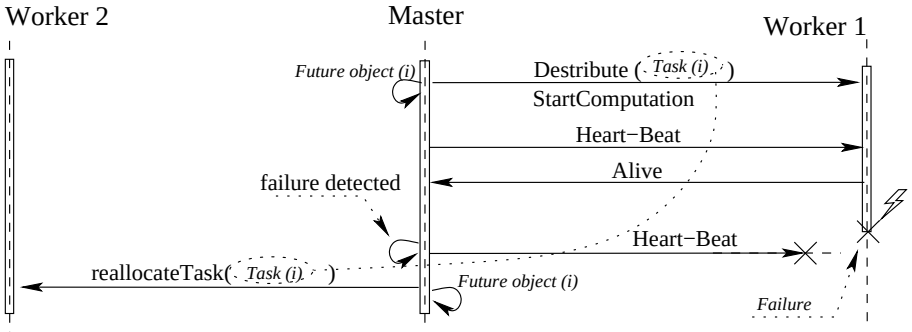

Figure 2.15: Task reallocation sequence diagram in case of failure.

In our application, the *Resources-Manager* periodically sends heart-beats to the workers. If a worker fails, it is removed and its task is recovered and resubmitted to another safe worker (see Figure 2.15). This operation is done by implementing *reallocateTask* of P2PMaster interface. The task is assigned to one or more available worker(s) and only the first returned result of the same task is taken into account. Other results of the same task are ignored. This process is performed at the end of the computation of all tasks.

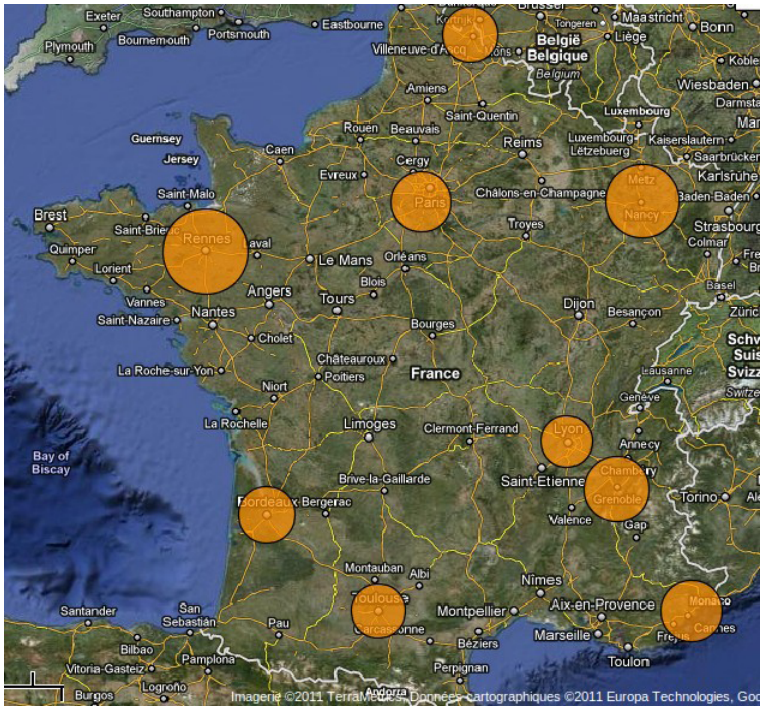

Figure 2.16: Grid'5000 French nation-wide grid infrastructure

2.5 Experimentations

P2P-B&B has been experimented on the Flow-Shop scheduling problem (*FSP*) considering the total completion time (C_{Max}) as cost function. We considered the Taillard instances [TAI93]: I_1 : $ta\_20\_5\_2$, I_2 : $ta\_20\_5\_3$, I_3 : $ta\_20\_10\_1$, I_4 : $ta\_20\_10\_2$ and I_5 : $ta\_20\_20\_1$[1].

2.5.1 Experimental Environment

P2P-B&B has been implemented on top of the ProActive middleware [PROA, BBC+06, CDCL06]. Recall that ProActive is an open source Java library aiming at simplifying the programming of multi-threaded, parallel and distributed applications for Grids, multi-cores systems, clusters and data-centers. It allows concurrent and parallel programming and offers distributed and asynchronous communication, mobility and a deployment framework.

The approach has been experimented on Grid'5000 [GRIDa] which is composed of a set of clusters distributed over 9 sites located in 9 different towns in France (see Figure 2.16). The different sites are interconnected using a dedicated fiber of the 10Gbps *Renater* [RENA] network infrastructure. In our experiments, 6 sites have been involved. The experimental hardware platform characteristics are presented in Table 2.1. The use of Grid'5000 is done through the OAR [OAR] reservation system. Once reserved, the

[1] $ta\_i\_j\_k$: a Taillard benchmark with i: number of jobs, j: number of machines and k: the instance number

Site	CPU characteristics	Cores
Lille	AMD Opteron 285, 2.6 GHz	104
	Intel Xeon E5440 QC, 2.83 GHz	368
	AMD Opteron 248, 2.2 GHz	106
	AMD Opteron 252, 2.6 GHz	40
Lyon	AMD Opteron 246, 2.0 GHz	112
	AMD Opteron 250, 2.4GHz	158
Bordeaux	Intel Xeon EM64T 3GHz	102
	AMD Opteron 2218 2.6 GHz	372
	AMD Opteron 2218 2.6 GHz	80
Orsay	AMD Opteron 246, 2.0 GHz	60
	AMD Opteron 246, 2.0 GHz	372
Rennes	AMD Opteron 6164 HE, 1.7 Ghz	960
	Intel Xeon X5570, 2.93 Ghz	200
	Intel Xeon L5420, 2.5 Ghz	512
	Intel Xeon 5148 LV, 2.33 Ghz	132
	Intel Xeon 5148 LV, 2.33 Ghz	264
Sophia	AMD Opteron 246, 2.0 GHz	98
	AMD Opteron 275, 2.2 GHz	224
	AMD Opteron 2218, 2.6GHz	200
	Intel Xeon E5520, 2.26GHz	360
Total		4302

Table 2.1: Experimentation hardware platform

# of Proc	Depl Time	I_1	I_2	I_3	I_4	I_5
06	15	03	492	1815	277	2411
20	46	10	409	170	111	2362
50	112	16	277	100	59	2362
100	234	-	194	91	-	2358
200	504	-	160	79	-	2345
300	713	-	158	63	-	2345
600	1949	-	121	64	-	2339
1500	4186	-	-	-	-	2331

Table 2.2: Some obtained deployment and resolution times

machines of the Grid are exclusively owned by the user (dedicated machines). However, the network is not dedicated, it is shared by the different users of the Grid. Therefore, up to 1500 grid nodes are involved in the experimentations according to their availability.

2.5.2 Experimental Results

The application has been deployed on 6, 20, 100, 200, 300, 600 and 1500 processors. It was launched in P2P mode where the whole processes run with the lowest priority to reach one of the P2P Computing characteristics which is the exploitation of idle CPU cycles.

Table 2.2 shows execution times (in seconds) of the exactly solved instances obtained using different numbers of processors, and the reached upper bounds for the instance I_5. The total deployment times are showed in the second column.

First, we notice that the performance raises when the number of processors increases. Indeed, the instance I_3 was not solved along 03 hours and 40 minutes on a single machine. It was solved exactly in 30 minutes and 15 seconds on six machines. Moreover, it takes only 1 minute 40 seconds using 50 machines (more than 18 times more efficient than using 6 machines). The only exception is when solving I_1, it was solved three times faster on 6 machines than on 20, and 5 times more efficient than on 50. This can be explained if we take a look at the situation of the solution regarding the space of solutions. It was found in the 3^{rd} node of the solutions tree, this means that six machines were sufficient to find the solution and 50 machines take an additional time to manage tasks and free all workers deployed. However, deployment times in the table show how this time increases when the number of machines increases. In some cases this duration is greater than the calculation time itself.

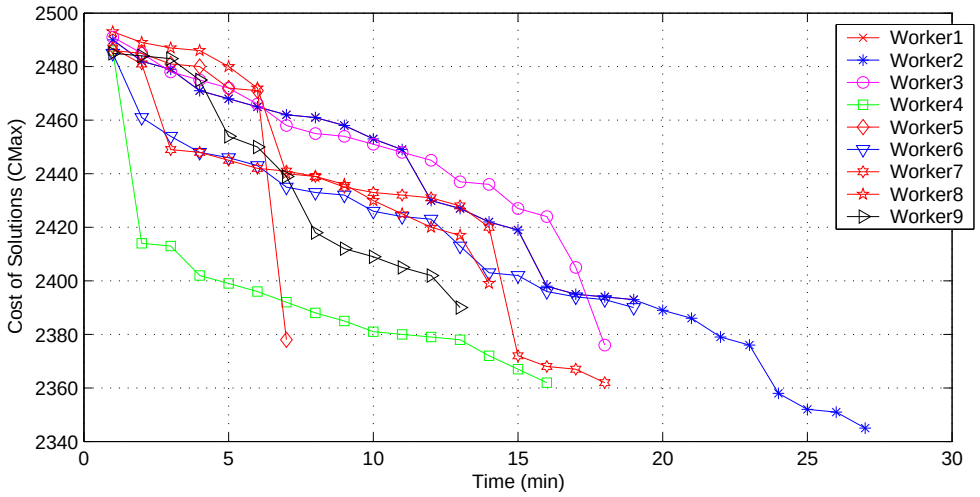

Figure 2.17: Solutions costs without enabling real-time direct communication between workers that succeeded to improve the global upper bound

Second, we notice that the communications were very beneficial. The workers share the obtained upper bound, they communicate it between them and form a global knowledge base. The workers cooperate between them to find quickly solutions that allow them to eliminate more branches in their subtrees. With this functionality, no worker finds a solution less significant than the upper bound or explores a branch with a lower bound greater than the global upper bound.

In the following, we evaluate the benefit of the direct communication enabled by our approach. Figure 2.18, respectively Figure 2.17 show the evolution of solutions costs using real-time direct communication, respectively without using this feature of our approach. In other words, in Figure 2.18, the workers perform multiple executions of atomic task and multiple real-time communication, whereas, in Figure 2.17 the workers perform a single execution of compact task before communicating once.

In the first figure, the curves are crossed between them and no worker knows the value of the other workers' upper bounds. Each worker only uses its local knowledge base and ignores new knowledge generated by the other workers. For example, *Worker4* has

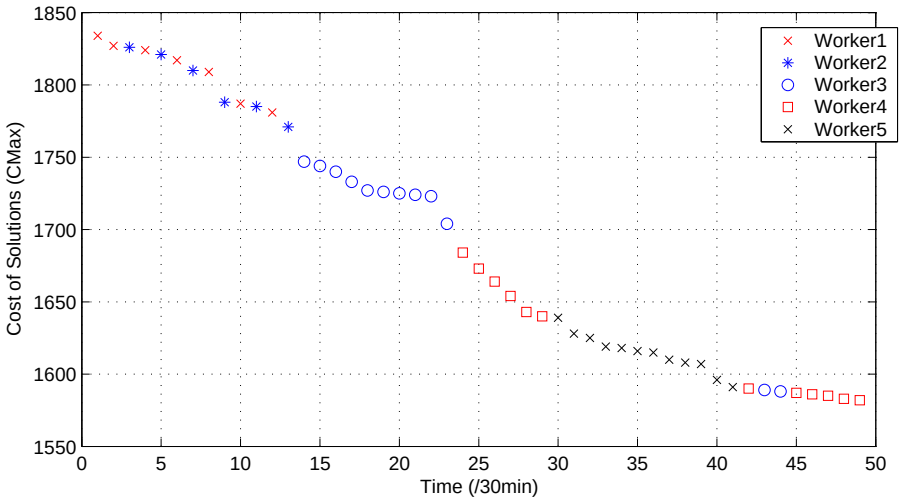

Figure 2.18: Solutions costs with communication between workers that succeeded to improve the global upper bound

reached a cost equal to 2416 at only the second minute. If this solution is communicated to other workers, all less interesting solutions situated in the downstream in comparison with the current value, would not be explored. The same remark is true for *Worker6* when it finds the solution with the cost 2379 at the seventh minute.

In Figure 2.18, the curve of solutions found by the whole of the workers is decreasing. This means that the workers benefit from the multiple execution of atomic tasks and the multiple realtime communication using typed group communication provided by ProActive. These workers, after they know new solutions values, they do not search in this space again.

2.6 Conclusion

The use of exact methods for the resolution of COPs, such as B&B which is one of the most known methods is beneficial. However, their use on applications of industrial size is only possible by the use of a great computational power. Large scale parallelism based on the use of Grid Computing is shown today as a potential tool which offers such power. Several factors must be taken into account to develop MW grid-based methods to take benefit from direct communication and for a better exploitation of the computing power: (1) A study and a good choice of a suitable parallelism model; (2) A good management of the knowledge generated by these algorithms; (3) An exploitation of all the tools that the grid-middleware offers on the control of the computing network.

In this chapter, we proposed a P2P MW-based B&B framework (P2P-B&B) aiming at facilitating the development of grid-based B&Bs and hiding the complexity of the grid to the users. The scalability is achieved by reducing the task request frequency. The task request frequency is reduced by performing multiple executions of an atomic

task and doing multiple communications rather than single execution of a compact coarse-grained task before doing a unique communication. This enables direct communication between workers allowing them to share their upper bounds and to perform other collaboration tasks alleviating the master process.

The algorithm has been implemented on top of ProActive to take benefit from its numerous features especially related to deployment and communication. We used the typed group communications and one-way asynchronous invocation methods for the knowledge sharing by real-time direct communication between the workers. We also used the listeners in order to take into account new arrivals. Finally, we used the futures related to objects for the collection of the computation results.

The performance evaluation of our contributions is performed on Grid'5000. These experiments showed the interest of the collaborative work and demonstrated the benefit of the real-time direct communication between workers. However, this approach has shown its limits in terms of deployment time cost. Indeed, in some cases the deployment takes more than the computation time.

Chapter 3

H-B&B: A Hierarchical Master/Worker-based B&B Algorithm

3.1 Introduction

In the first chapter, we introduced principles about parallel B&B, and we also identified requirements that a B&B has to fulfill when considering large scale environments. As mentioned, the scalability is one of the major issues to deal with when developing grid-based B&Bs. In the previous chapter, we have proposed a framework which facilitates the developing grid-based B&B using a P2P MW-based paradigm allowing direct communication between the workers. However, we have seen that P2P-B&B is still limited in terms of deployment when facing larger number amounts of grid resources.

We have seen before in Chapter 1 the proposed approaches in the literature to circumvent the limits of MW-based B&Bs. Let us recall that these techniques fall into two categories. The first category includes techniques maintaining the use of the MW scheme and modifying the used resolution methods such as in [MMT07a, MMT07b, EPH00] adapting them to the large scale environment. In the second class, modifications are brought to the main scheme of the MW paradigm [ANF03, AFO06, GSDB09, BDLP08, DVC+09] based on the Hierarchical Master-Worker paradigm (*HMW*). In this paradigm, there are multiple masters, each of them supervises multiple workers. Therefore, the hierarchical organization allows pushing far the limits of the MW to support more worker processes.

However, most of the proposed HMW-B&B approaches are static and only composed of one level of masters each of them manages a set of workers. They still have a limited scalability in large scale environments of thousands of processors such as computational grids. In addition, they handle tasks of fixed grain sizes and do not evolve over the time to deal with the dynamic nature of grids.

In this chapter, we propose a new HMW-based B&B (*H-B&B*) aiming at improving

the scalability of the conventional MW-based B&B paradigm eliminating the bottlenecks created on the central master process. H-B&B is based on the *P2P-B&B* framework. Unlike the literature approaches, *H-B&B* is fully dynamic as it is composed of several levels of masters, and evolves over time according to the dynamic acquisition of new computing nodes. This work has been published in [BMT12, BMT11b].

This chapter is organized as follows: In Section 2, we present AHMW a framework on which the hierarchical B&B we developed *H-B&B* is based on. The architecture, components of the framework and its working, work management, and the adaptive feature are then detailed. Section 3 presents the implementation of H-B&B using the AHMW framework. In this section the exploration strategies, load balancing, and work management are detailed. In Section 4, we show how to implement it on top of ProActive. The performance evaluation is presented in Section 5. Finally, we conclude this chapter in Section 6.

3.2 AHMW: an Adaptive HMW Framework

The AHMW framework developed in this chapter is based on the previously proposed framework *P2P-B&B* [BMT09] performing direct communication between processes. Before detailing the architecture and working of AHMW some concepts used throughout this paper are defined in the following.

Super-Master: a super-master is the root process of the system which plays the role of a master in the classic MW paradigm. There is only one instance in the system.

Master: a master is a process which has at least one child. Its children can be masters and/or workers.

Worker: a worker is a process which has no children. It is similar to a worker in a simple MW paradigm but has other roles than simply executing tasks.

Child: a child of a master or the super-master can be a master or a worker.

Parent: a parent of a master or a worker process is the process which creates and manages it, it can be the super-master or another master.

Colleagues: colleagues of a process are all its neighbors except its parent and its children. A process can have at a time masters and workers as colleagues.

3.2.1 Processes of the framework

AHMW is composed of three processes with different roles: *super-master*, *master* and *worker*. In the following, we survey each of them and explain their functioning, while details are presented in the next section.

Super-Master: The super-master is the unique entry of the system and it is the top master but it is not the unique responsible of managing the processes of the system. Its main role is to acquire computing resources from the computational grid, to decompose the initial problem to be solved, to distribute obtained sub-tasks among masters and/or workers it manages and recovers the results, to reassign tasks of failed workers,

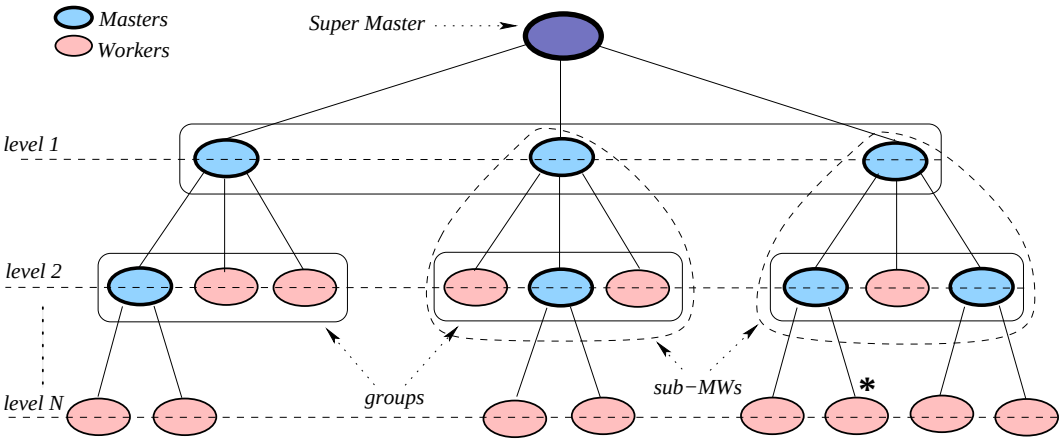

Figure 3.1: Hierarchical organization of AHMW

and it is the initiator of the construction of the entire hierarchy.

Master: Master processes are intermediate processes between leaves (workers) and the root of the hierarchy (super-master). Unlike all proposed hierarchical systems, it has no restriction in its role. Indeed, a master can do all the features the super-master can do except the acquisition of computing resources. In addition, it acquires tasks from its parent, participates to the execution of tasks, forwards all upcoming and outgoing communications (to/from the internal group of processes it manages), and detects failed processes among its children.

Worker: A worker in AHMW is not a simple worker as in MW paradigm. In fact, it has other roles than the calculation of the assigned task. In addition, it participates in the building of the hierarchy and the decomposition of the assigned task preparing itself to change its behavior to become a master when it acquires sufficient computational nodes.

3.2.2 Hierarchical organization and architecture of AHMW

We have seen previously in chapter 2 that the P2P-B&B is composed essentially of two entities a master and a worker. Their roles and their components are predefined. Roles of AHMW processes are either defined by their components and the components that are running. Before we detail the general architecture of AHMW and summarize the composition and the functioning of the different processes, we present its hierarchical organization.

3.2.2.1 Hierarchical organization

AHMW is a multi-layer HMW composed of several sub-MW (*sub-MW*) systems (see Figure 3.1). Each sub-MW contains the same processes as the simple MW where each

master manages a set of workers. The whole sub-MWs are organized in a hierarchy with several levels of masters. Each sub-MW of the system is based on the *P2PBB* framework [BMT09] where a master manages a dynamic group of communicating workers. Hence, the hierarchy generated by AHMW is multi-layered, fully dynamic and evolves over time according to the dynamic acquisition of new computing nodes. In the hierarchy, the workers are represented by the leaves and the masters by the inner nodes. Let us recall that a master do not only manages workers but also other masters of a lower level. Moreover, a master can have at a time masters and workers in its descendants.

Width and depth of the hierarchy are both dynamic. They depend on the number of acquired computing nodes and the size of the groups which form a sub-MW. These two parameters affect the expected outcome of the system. The width determines the number of tasks that can be processed in parallel. When the width grows, the number of parallel tasks grows and *vice versa*. The depth determines the granularity of processed tasks. When the hierarchy is deep, the system generates more and more fine-grained tasks, therefore easy to process. The ideal configuration is to have a deep and large hierarchy but two important parameters must be taken into account: the capacity of masters in choosing the size of groups and the minimum authorized granularity of tasks to compute. For the first parameter, a large sized group of workers, overload their master then degrade the performance of the whole system. On the other hand, too fine-grained tasks lead masters to spend much more time in distribution of tasks which degrades considerably the overall performance.

3.2.2.2 Architecture and components of AHMW

Figure 3.2 shows the general architecture of the framework. The super-master process is similar to the master process in the P2P-B&B framework. It is composed of one thread (*Work-Manager*) and three active objects (*Resources-Manager*, *Worker-Supervisor*, and *Statistician*). Their roles are also similar to those in P2P-B&B (see Section 2.2.3 in Chapter 2).

The worker and the master processes in AHMW are composed of the same components and their roles are defined by the components that are running.

The worker is composed of four threads (*Task-Provider*, *Task-Processor*, *Work-Manager*, and *Statistician*) and two Active Objects (*Resources-Manager* and *Worker-Supervisor*). The *Task-Processor* thread allows to execute the task assigned to the current process (it can be a worker or a master). *Task-Provider* provides the current process in terms of tasks by asking the master process. It works in mutual exclusion with the *Task-Processor*, it wakes up the *Task-Processor* when a new task is acquired and then sleeps waiting for task request event. The *Task-Processor* in its part, wakes up the *Task-Provider* after it finishes its task and then sleeps waiting for new tasks. The *Statistician* thread periodically sends statistics to the super-master process. It interacts with the local threads (*Task-Processor* and *Task-Provider*) and with the master's *Statistician*. It collects statistics concerning the state of the current process (in execution or waiting state) and the task work progress and sends them to the master. The *Work-Manager* thread serves when the current process plays the role of master to manage the work pool and to serve the task requests coming from the processes it

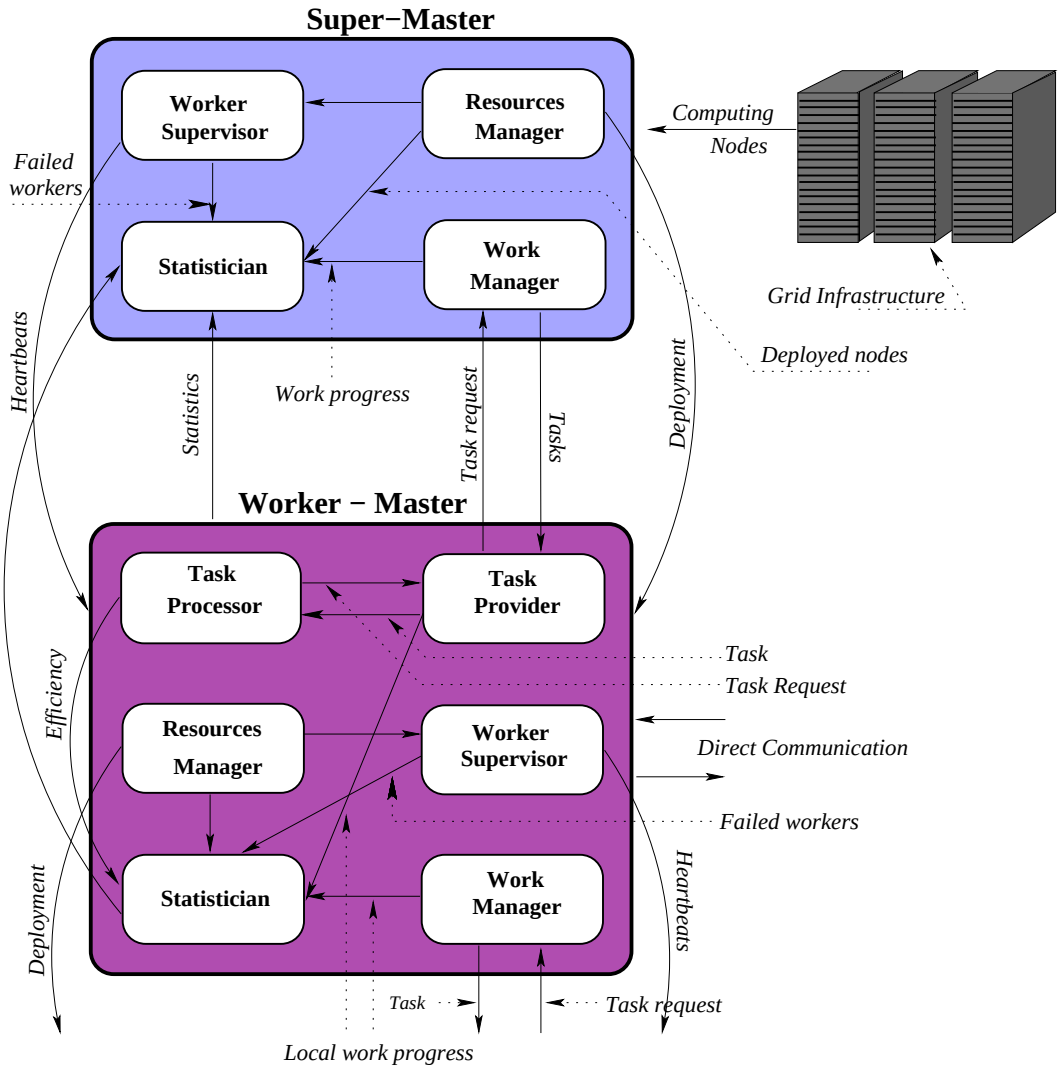

Figure 3.2: Architecture of AHMW

manages.

The *Resources-Manager* of a master serves to manage the computing resources. Unlike the super-master's *Resources-Manager* which acquires free computing nodes from the computational pool (grid infrastructure), its role is to only deploy the nodes received from the master at the upper level and to organize the deployed processes into groups according to the used strategy. The *Worker-Supervisor* allows a master process to detect any failure or disconnection among the processes it supervises. It sends periodically heartbeats to the processes. If a process does not respond then it is marked as failed. Finally, the *Statistician* allows to recover and report statistics during the execution of the framework such as the processors load, efficiency, work progress, number of launched and failed workers, communication load, etc.

These threads are not running all the time they are launched if necessary according

Component	Worker	Worker and Master	Master
Task-Processor	✓	✓	×
Task-Provider	✓	✓	✓
Work-Manager	×	✓	✓
Resources-Manager	×	✓	✓
Worker-Supervisor	×	✓	✓
Statistician	✓	✓	✓

Table 3.1: Process roles according to the launched components

to the role of the process. The same for the Active Objects, they are only created when the process must play the role of master. In the following we present how a process in AHMW switches from a role to another.

3.2.2.3 Adaptive feature of AHMW

The interest of AHMW lies in the flexibility of managing different types of processes. Indeed, the role of a node in the hierarchy is not predefined but it is defined in a dynamic way. Initially, any newly created node behaves as a worker but it changes its behavior and becomes a master as soon as it acquires computational nodes. Its colleagues will always consider it as a simple worker. A worker newly converted to a master, has the choice to terminate its task or to suspend its work and reassign its task to another worker chosen from its descendants depending on its load. The flexibility in behavior is also valuable for a master. Indeed, a master becomes a worker when it looses all its computing nodes.

From implementation point of view, the behavior of a process is defined by the function it executes. Each process deployed on a computational node launches the two threads *Task-Provider* which provides it in terms of tasks and *Task-Processor* which executes the current task. Therefore, it behaves as a worker.

When a worker receives computational nodes from its master, it creates the two Active Objects (*Resources-Manager* and *Worker-Supervisor*) to deploy, manage and supervise the processes it creates. It also launches the *Work-Manager* thread to manage the work pool and to serve the requests of the workers. At this time, the process changes its behavior and plays the role of a master and a worker at a time, i.e., it executes its task and manages and serves its children. During the execution, if it is overloaded, i.e., it receives a huge amount of task requests, it gives up to the role of worker and keeps only the role of master. The different roles of a process according to the launched components are summarized in Table 3.1.

3.2.2.4 Construction of the hierarchy

Initiated by the super-master, the construction of the hierarchy is performed by the collaboration of masters and workers of the different sub-MWs. It is built gradually as and with the arrival of computational resources in the grid through the super-master.

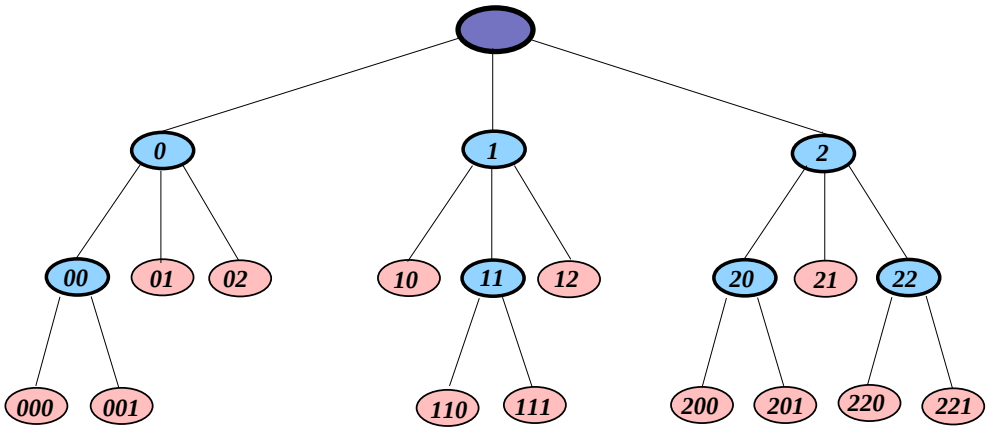

Figure 3.3: Identifiers of AHMW processes

Each time a computational node joins the Grid it is integrated by the super-master as a worker if the group size threshold is not achieved. Otherwise, the computational node is redirected to one of its children which adapts its role to become a master with the new node as a first worker. Each master (and worker) does the same until the entire hierarchy is built.

Each process in the system has a unique identifier which identifies it during its life time. The identifier represents its serial number associated to it at its creation and that of the sub-MW it belongs to (see Figure 3.3). A sub-MW is identified by its master. So the identifier of a process is its serial number concatenated with serial numbers of its ascendance. Let l be the level of the current process, p_k^i the i^{th} ascendant of the process p_k, and $C(p_i)$ the identifier of the process p_i in the sub-MW it belongs to. The identifier of a process p_k is obtained as follows: $I_k = \bigcup_{i=0}^{l} C(p_k^i) \cup C(p_k)$.

The dispatching of nodes is done according to a policy which maintains the mean degree[1] of the entire tree built by the hierarchy. Maintaining the same degree of the tree during the lifetime of the system is crucial. When the hierarchy maintains the same degree, a balanced tree is obtained allowing to preserve the same load in all parts of the tree as well as it permits to handle a specific granularity of tasks.

3.2.3 Working and work management

3.2.3.1 Task management

AHMW is especially dedicated to problems that can be solved by the *divide and conquer* paradigm. This paradigm recursively breaks down a problem into two or more subproblems of the same type. In combinatorial optimization, the *Branch and X* algorithms dynamically spawn new tasks. They perform recursive decompositions of the

[1]The degree of a node is the number of its sibling nodes. It is also the size of each group in the hierarchy.

initial problem into a set of N smaller subproblems and then solve each of them independently. Population-based heuristics are another type of algorithms in which several decompositions at different levels can be obtained. For example, in *Genetic Algorithms*, a population of individuals can be decomposed into several sub-populations and then each sub-population can evolve independently.

In AHMW, initially the super-master decomposes the initial task into multiple sub-tasks. After that, each master getting a new task from its parent, decomposes it into smaller sub-tasks and saves them into its work pool. In the following, we present the different mechanisms related to the task decomposition and distribution, the communication between the different processes and the termination detection.

3.2.3.2 Dynamic decomposition and distribution of tasks

In a classical MW, the decomposition of tasks is centralized, it is made by the central master process. This causes additional computation to the master which has also in charge the distribution of tasks, recovery of results, communication, and management of the different workers. In AHMW, we propose a distributed decomposition. In fact, it is done by both the super-master and the different masters and workers at the different levels of the hierarchy. Many advantages can be noticed regarding this decomposition:

- The super-master gains in terms of calculation time and can do other tasks such as the management of the hierarchy, result recovery, statistics, etc.
- In a centralized MW, the decomposition is sequential whereas in AHMW it is done in parallel. It reduces considerably the decomposition time, thus the idle time of workers.
- It allows rapidly reaching fine-grained tasks, which is not easy to obtain with a centralized decomposition. The same granularity can be reached by the centralized master but only after a long execution time which blocks workers waiting for tasks.

The decentralized decomposition process allows masters and workers of AHMW to handle tasks of different grain sizes. Each sub-MW at each level handles its own granularity because of the dynamic decomposition done by the masters at each level. In fact, each group of masters of a given level handles a task granularity different from other groups that are at a different level (higher or lower level). A master decomposes a task with granularity g and obtains a set of tasks with granularity g-k, k is the level of the decomposition used by the master. As said previously, the granularity of generated tasks depends on the size of groups. The smaller the groups are, the deeper the hierarchy is, thus finer tasks at the level of final workers are generated.

Each master has its own work pool which contains the tasks to be performed. This pool is obtained by the decomposition of the task received from its parent. When a master receives a task request from one of its children, it assigns one task from the work pool and then updates the list of assigned tasks which is a mapping between the assigned task and the worker assigned to. If the master has no more tasks in its pool, it assigns an empty task, that will force the child to block waiting for a new task.

A master that has at least one task, either received from its parent or obtained after decomposition of the local task, wakes up all blocked children waiting for tasks.

3.2.3.3 Communication

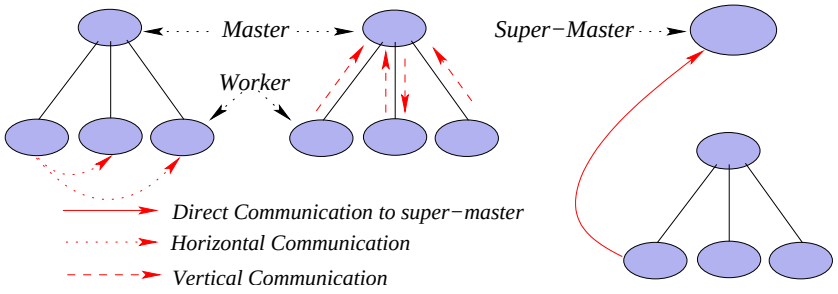

Figure 3.4: Communication types

Masters and workers of the same sub-MW perform direct communications between them without flowing through an intermediary. Workers inside the same sub-MW are visible to each other. Two workers of two different sub-MWs are not visible to each other, so their communication flows through other processes. Two types of communication are considered: horizontal and vertical (see Figure 3.4). Horizontal communication occurs between colleagues (*1-to-N* communication). Vertical communication occurs between a parent and its children in both directions: *1-to-N* for top-down communication and *1-to-1* for down-top ones. When a worker communicates information to another worker, it uses horizontal and vertical communication throwing masters of intermediate sub-MWs.

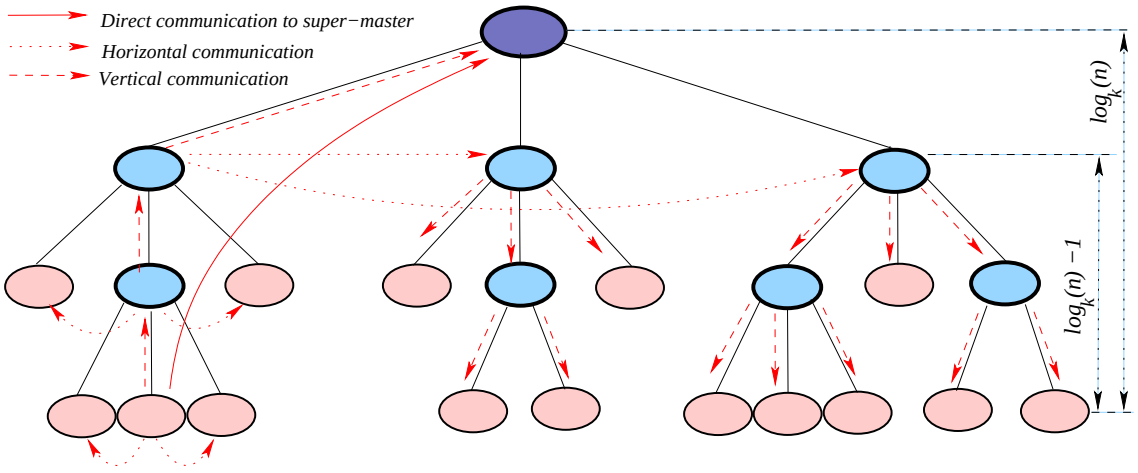

Figure 3.5: Broadcasting a solution

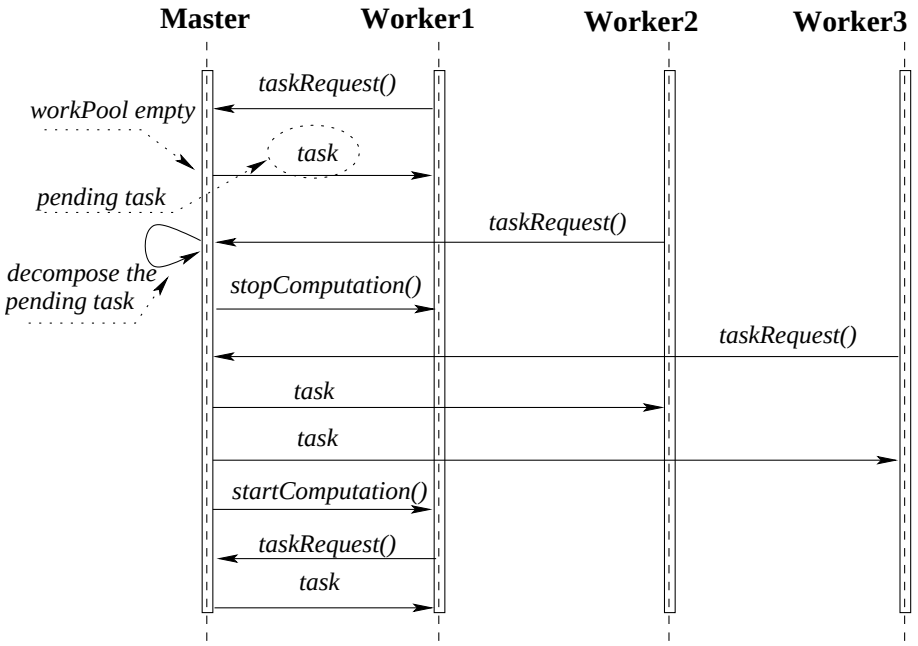

Figure 3.6: Load balancing sequence diagram.

In combinatorial optimization, broadcast communication is often needed. For instance, in B&B algorithms, global *1-to-N* communications are needed. A worker that finds a new upper bound broadcasts it to the others. To perform a broadcast in AHMW, a worker needs to make one horizontal *1-to-N* communication to its colleagues, one vertical *1-to-N* communication to its children and one vertical *1-to-1* communication to inform its parent. Parent, colleagues and children do the same until the new global solution reaches all the processes in the hierarchy. In order to avoid the loss of the global solution along the path to the super-master, the worker which finds a new global solution makes one direct *1-to-1* communication with the super-master to share the information with it.

In a balanced hierarchy, a message from one worker (in a leaf) takes $log_k(n)$ hops to reach the super-master, k being the size of each group and n the number of computational nodes in the hierarchy. Therefore, a message between two farthest workers[2] takes at most $2 \times log_k(n) - 1$ hops (see Figure 3.5).

3.2.3.4 Load Balancing

The heterogeneity and dynamic nature of grid resources and the irregularity of the B&B tree make variable the exploration time of subproblems. This irregularity may affect the overall performance of the algorithm. Indeed, some workers take a long time to explore their subproblems and then block other workers waiting for their termination. To deal with this issue, idle workers are involved in the exploration of subproblems of

[2]Farthest workers are workers belonging to the descendance of two distinct masters localized in the first level of the hierarchy

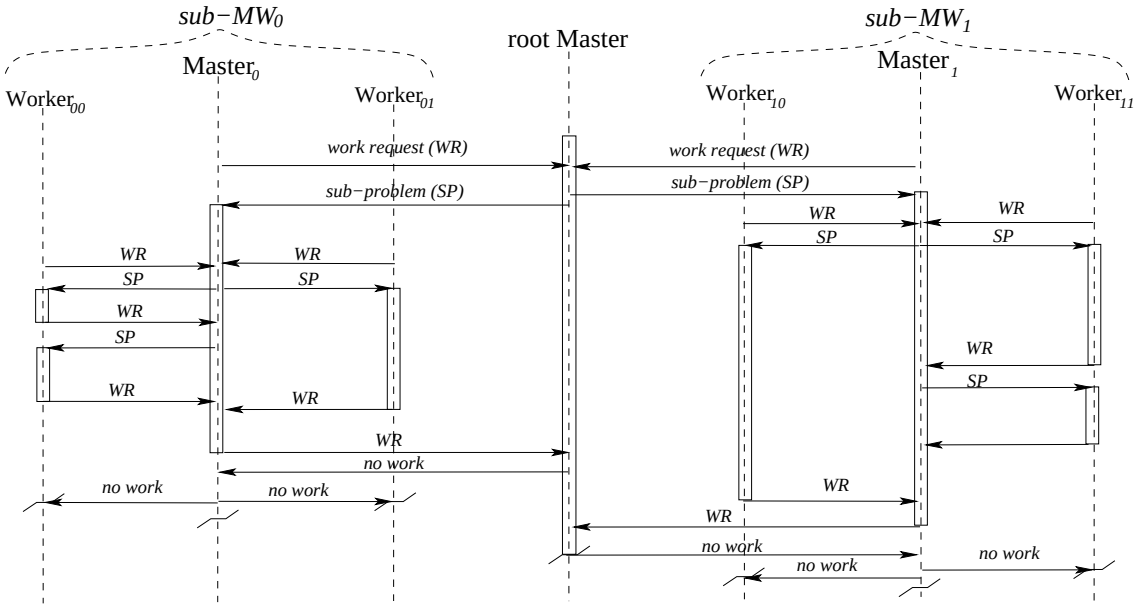

Figure 3.7: Termination detection sequence diagram

other pending workers. The subproblems of the pending workers are decomposed by the master into smaller ones and then distributed among the idle processes (see the sequence diagram in Figure 3.6). The master holds a list of unexplored subproblems LUS and a list of pending processes LPP in charge of these subproblems. When the master receives work request from an idle worker and its work pool is empty, it picks one subproblem from LUS and branches it into smaller subproblems. The obtained subproblems are dispatched among the idle workers. The pending worker is removed from LPP and when it is idle again, it is considered as new idle worker.

3.2.3.5 Termination detection

In massively parallel environments, the termination detection of applications is crucial and must be taken into account since several parameters make it hard to manage such as failures, the dynamic nature of B&B trees, and the absence of global information of the work progress. In our approach, we distinguish two types of termination: local and global ones. The local termination is trivial, it is detected when the local work pool is empty and the upper process does not have any more tasks. Let B be the set of launched sub-MW processes, b a sub-MW process, p a master, $Pool(p)$ the work pool of p, and $A(p)$ the list of assigned subproblems of p. The termination detection is performed using the following condition: $\forall b \in B, \forall p \in b, Pool(p) = \phi \wedge A(p) = \phi$. The verification of such condition is impossible because we have no global view of the hierarchy, so the termination cannot be detected because each master has a local (thus partial) view of the work progress. Each process only knows the state of its own progress and that of its children. Therefore, each process can detect its own termination when $Pool(p) = \phi \wedge A(p) = \phi$. Based on such statement, the termination of a group of processes causes the termination of their parent and the termination of each master p

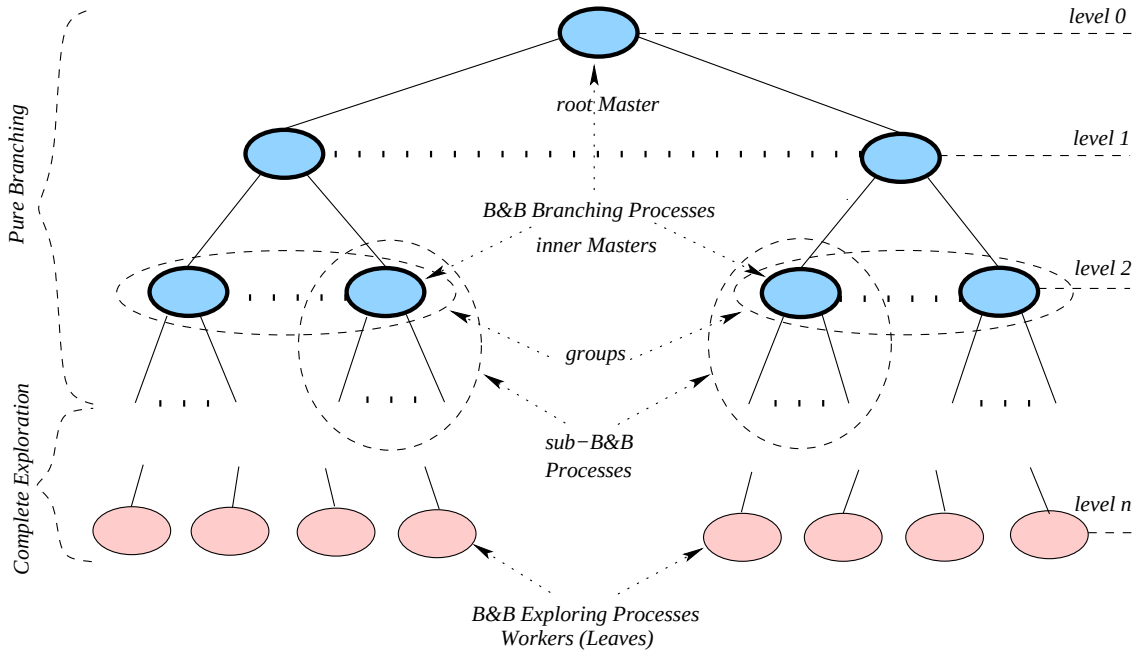

Figure 3.8: General design of H-B&B

causes the termination of the sub-B&B b it represents, and so on until the root of the hierarchy is reached (see the sequence diagram in Figure 3.7).

The figure presents a HMW subdivided hierarchically into two sub-MWs organized into groups of 2 processes. A root master manages two masters, each of which manages two workers. The termination of $sub\text{-}MW_0$ is detected when $Master_0$ has an empty work pool and receives no work from the root master. Whereas, the global termination is detected when the work pool of the root master is empty and it receives the last unsatisfied work request from its children $Master_1$.

3.3 H-B&B: An AHMW-based parallel B&B

H-B&B is a parallel B&B based on the HMW paradigm using AHMW. It aims at dealing with the scalability issue and performance degradation of the MW-based B&B in large scale environments. According to the AHMW framework, the proposed approach is composed of several sub-B&B algorithms. They are launched in parallel and act independently on different sub-trees so that they are organized hierarchically in different levels. At the level of each sub-B&B, a MW paradigm is built where one master manages several B&B workers. Therefore, two types of processes are considered: Masters at the root and the inner nodes of the hierarchy and workers at the level of the leaves. As mentioned before, AHMW is based on *P2P-B&B* therefore, H-B&B is composed of communicating groups of MW-based sub-B&B processes (see Figure 3.8).

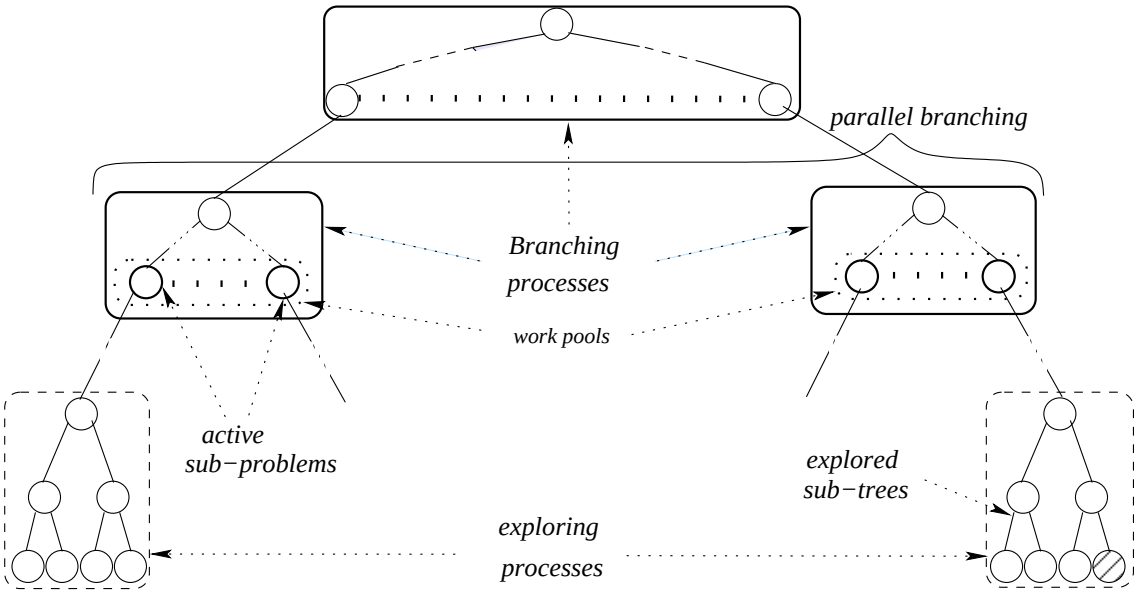

Figure 3.9: Search tree subdivision. The search tree is subdivided hierarchically into several sub-trees and each sub-tree is assigned to one sub-B&B process.

H-B&B performs an initial branching on the initial problem to eliminate a great number of branches at the top of the search tree. Therefore, the overall performance is improved with the early elimination of subproblems that do not lead to the best solution. This work is done once by the root master. The inner masters perform a branching in order to decrease the size of subproblems until reaching sufficiently fine-grained subproblems which can be explored sequentially by workers at the leaves of the hierarchy. subproblems are handled by giving the priority to the most promising ones according to the value of their lower-bounds.

3.3.1 Search Tree Subdivision

The search tree is subdivided hierarchically into several sub-trees (see Figure 3.9), each sub-tree is assigned to a sub-B&B process according to the available computing nodes. According to this design, each sub-B&B acts on its own subproblem in parallel. In addition to the parallel exploration of sub-trees, H-B&B also performs a parallel branching. Indeed, the inner nodes of the hierarchy perform a parallel branching on subproblems and the leaves perform a parallel exploration of the obtained subproblems. Therefore, two types of processes are distinguished: *Branching processes (BP)*, hosted by the root and the inner masters, and *exploring processes (EP)*, hosted by the workers at the leaves.

Considering the parallel exploration of sub-trees and the parallel branching, H-B&B overlaps *type 1* and *type 2* of the Gendron *et al.* classification according to the degree of parallelism. In the synthesis presented in Chapter 1, the algorithm belongs to both the first and the second categories. It belongs to the first category since the search tree

is built in parallel where each sub-B&B process handles its own subproblem independently, and it belongs to the second category since the branching operation is executed in parallel on the different inner nodes of H-B&B. Starting from the original problem, the branching processes perform independently recursive branching on subproblems in order to obtain smaller ones that can be explored in a reasonable time by the exploring processes. Considering the distributed branching, the branching time is reduced thus it minimizes the idle time of workers waiting for work.

Each master has its own local work pool represented by a list of active subproblems and multiple work pools are considered, thus a collegial strategy is used. During the search, the local pool evolves continuously and when it is empty, the process sends a work request to the master at the upper level. A master receiving such a request consults its local work pool and asks its upper level if its work pool is empty. It also performs an additional branching if the available subproblems do not correspond to the requested work unit size.

3.3.2 Exploration strategies

To adapt the B&B algorithm to AHMW, three different exploration strategies are adopted according to the type of the process performing the exploration and to its role: *breadth-first* search by the super-master, *smart best-first* search (a new exploration strategy introduced in this work) by masters, and *best-first* search by workers. A smart best-first search is similar to the best-first search exploration but without reaching leaves of the subtree. The role of the super-master and masters is to decompose tasks and spawn new smaller ones, whereas the role of workers is to provide solutions. In H-B&B, the masters generate and handle different grain sizes. The granularity of a task is defined as the size of the subproblem. In other words, it is given by the level of the explored subtree.

3.3.2.1 Breadth Search (BS)

The root master uses the breadth-first selection strategy (see Figure 3.10). It explores k levels of the original search tree and generates at most $n = \prod_{i=0}^{k-1}(N-i)$ subproblems assigned to the different branching processes of the lower level. Subtrees are explored by giving priority to the most promising ones according to the value of their lower bound. The root master obtains as a result a list of sorted subproblems which are assigned to its children according to their order.

3.3.2.2 Smart Best-First Search (SBFS)

We have defined a new exploration strategy appropriate to an online branching. This strategy is not greedy in terms of execution time which makes it rapid and appropriate to generate a great number of fine-grained subproblems in a short time, thus improving the overall performance.

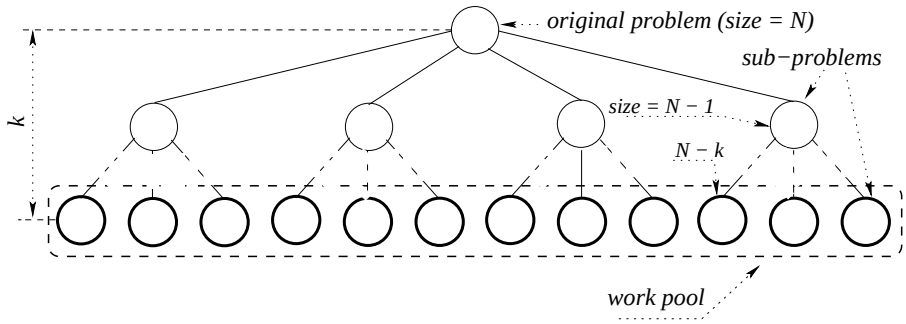

Figure 3.10: Breadth Search exploration. The root master explores the tree by levels.

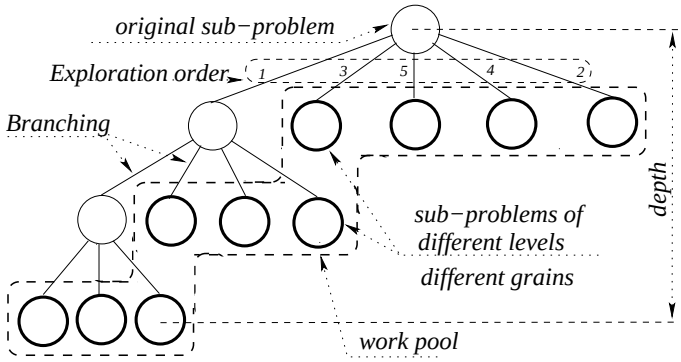

Figure 3.11: Smart Best-First Search. Masters explore subtrees partially and the obtained subproblems are considered as work pools.

The role of the masters is to perform branching on subtrees. Therefore, the search algorithm is adapted to this role. Indeed, the masters do not perform a complete exploration of their subtrees but only a partial exploration using SBFS (see Figure 3.11). They explore subtrees until the threshold of the considered granularity is reached. Then, the obtained list of subproblems is considered as the local work pool. We define the granularity of a subproblem as the absolute depth of the handled subtree. A work pool contains subproblems located at different levels of the subtree. Moreover, according to the level in the subtree, the subproblems have different grain sizes. As workers of the same sub-B&B handle the same grain size, the master makes a dynamic on-demand branching to get the required granularity when it is necessary.

Masters perform a multilevel branching and generate subproblems d times smaller than the current subproblem, d being the depth of the explored subtree. An SBFS of degree d is an exploration algorithm allowing to explore d levels in the subtree. The used degree differs from a master level to another in order to balance the work load of masters. The SBFS degree must be chosen dynamically in order to obtain, sufficiently small subproblems at the leaves of the hierarchy. Moreover, it must be fixed in such a way to avoid too coarse/fine-grained tasks. Handling coarse-grained tasks can penalize some workers belonging to the same sub-B&B when one or more worker(s) take(s) much

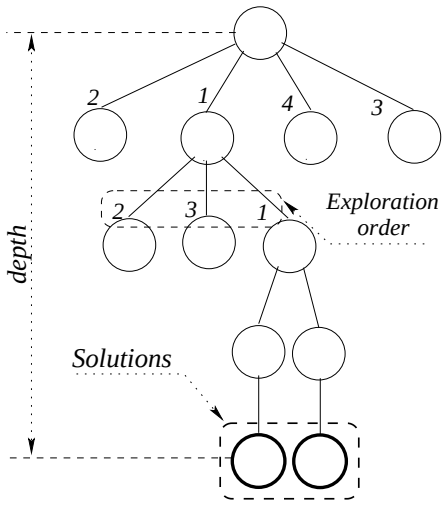

Figure 3.12: Best-First Search. The workers explore nodes according to the most promising ones.

more execution time than others. Whereas, fine-grained tasks can create bottlenecks at the level of the masters when the workers take a short time to explore their tasks.

Masters do not perform branching indefinitely to avoid having too small subproblems. When the minimum size of problems is achieved, the masters do not perform any more branching. They act like switches and forward subproblems from their parent to their children. The minimum size of problems is variable from a problem to another one and from an instance of the same problem to another one according to their search space landscape.

Let d be the depth of the subtree explored by a master, l the level of the current master in the hierarchy of H-B&B, and k the number of explored levels by the supermaster in the initial search tree, the granularity of tasks (size of subproblems) handled by a master mg is fixed as $mg = N - (k + (l-1) \times d)$ and the granularity of tasks assigned to its children is fixed as follows: $wg = N - (k + l \times d)$. Each master generates at most $n = \sum_{i=0}^{d-1}(mg - i) - d$ subproblems of different grain sizes (from wg to mg).

3.3.2.3 Best-First Search (BFS)

Since the role of workers is a complete exploration of their subtrees, they perform a best-first search to reach more quickly the solutions contained in the leaves of the subtrees. A worker which finds a solution better than the best solution found so far broadcasts it as soon as possible to the whole hierarchy using the policy described in Section 3.2.3.3. The priority is also given to the most promising branch according to its lower bound (see Figure 3.12).

The subproblems are smaller as the processes exploring them are closer to the leaves of the hierarchy. Therefore, the execution time (their total exploration time) is shorter.

That makes masters in the bottom of the hierarchy receiving more work requests because their children take a short time to explore their subproblems. Therefore, these masters become rapidly overloaded compared to those at the upper levels which are idle. To overcome this drawback, the number of work requests must be balanced for all the masters. Masters at the upper levels must generate more subproblems than their children. In order to obtain better performance in terms of the frequency of work request the SBFS degree must be increased for the upper masters and decreased for the low-level ones. That enforces upper masters to have work pools greater than the lower ones. Therefore, the work request frequency increases at the level of upper masters and decreases at the level of the low-level masters.

3.4 Hierarchical Deployment Using ProActive

Deploying efficiently a large number of processes in large scale environments is a great challenge. Let us remember that in the P2P-B&B framework, the deployment task is performed by the unique *Resources-Manager*. In AHMW, therefore in H-B&B, every master has its own *Resources-Manager* in order to decentralize the deployment process and involve the whole processes in this process. Each master is responsible on the deployment of the current master's children. Therefore, the deployment process is not restricted to the *Resources-Manager* of the super-master, but rather performed collaboratively by the different *Resources-Manager(s)* of the different masters. Using this strategy, a large number of nodes can be deployed in a reasonable time. If we assume that the deployment of one process takes x seconds, the deployment of n processes takes $x \times n$ seconds using the traditional M/W paradigm. Using hierarchical deployment ensured by H-B&B, the deployment takes only $k \times x \times log_k(n)$, k is the size of a group in one sub-M/W B&B. That is to say whatever the size of the computational pool, the whole nodes can be deployed in just few seconds. Let us assume that one node is deployed in 2 seconds. Therefore, using H-B&B with sub-MWs organized into groups of 10, the deployment of 1000.000 (1 million) nodes takes only $10 \times 2 \times \log_{10}(1000.000) = 120$ *sec.* However, it takes $2 \times 1000.000 = 2000.000$ *sec* $\simeq 23$ *days* using the MW paradigm.

In computational grids, the computing nodes are organized into geographically distributed sites implying a huge communication cost. Using ProActive deployment features the super-master's *Resources-Manager* implements *deployNodes()* method to provide the application by virtual nodes (VNs) that correspond to JVMs which contain active objects. To adapt the groups to the physical organization of the grid and to minimize the intra-group communication cost, the super-master's *Resources-Manager* introduces a new method. Indeed, *deploymentAccordingToSites()* handles the *xmlDescriptor* file and extracts the different sites the nodes belong to. After that, the first level of masters are deployed and organized according to the existing sites (see Figure 3.13). However, organizing the groups considering only the sites can penalizes the system performance because the same site can contain a large number of computational nodes inducing to the creation of bottlenecks. Unlike in [CBCM07] and in [DUGS06] where the unique criterion according to the groups are formed is the physical localization of the nodes on the clusters, in our approach, the groups are formed according

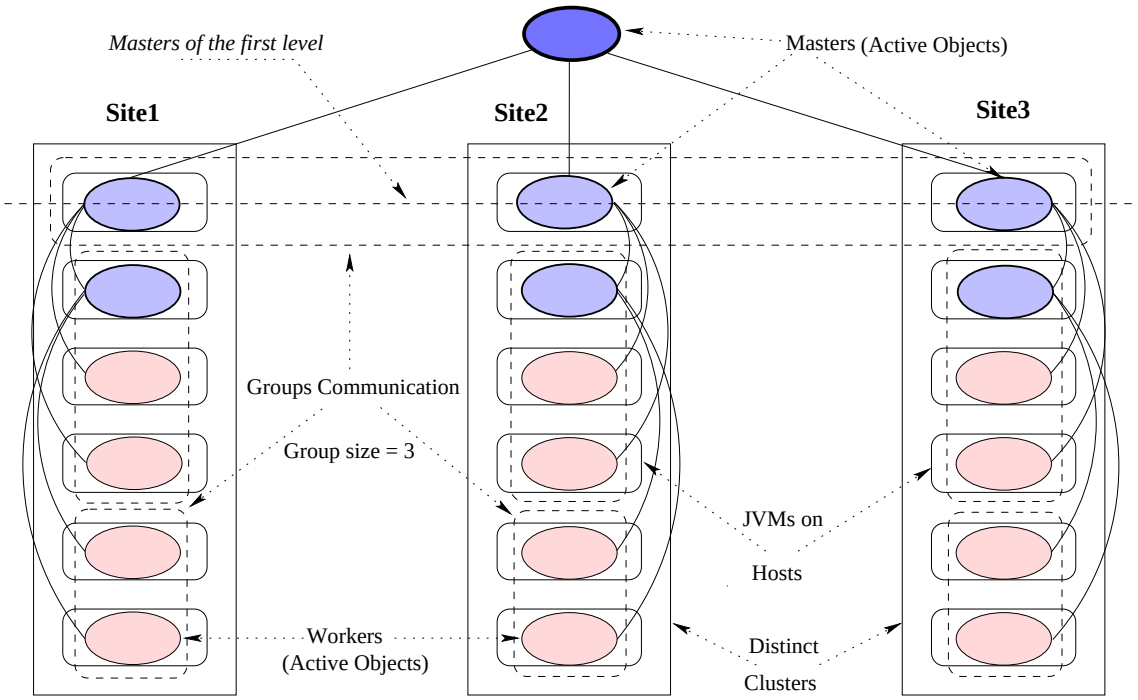

Figure 3.13: Deployment according to the localization of grid nodes.

to both their physical localization on the different sites and the maximum authorized group size. Indeed, when the maximum group size is reached, the *Resources-Manager* starts dispatching the nodes among the already deployed masters so that the formed hierarchy remains balanced. Accordingly, the processes belonging to the same group are deployed on the same grid site except for the first level of masters which contains masters deployed on different sites.

From the point of view of the user, he/she only has to define the list of nodes through the *xmlDescriptor* file and all the details about the grid infrastructure and the groups organization are hidden.

3.5 Experiments

To validate our contribution, H-B&B has been experimented on the Flow-Shop scheduling problem (*FSP*).

H-B&B has been experimented on Grid'5000 which is composed of a set of clusters distributed over 9 sites located in 9 different towns in France. The use of Grid'5000 is done through the OAR [OAR] reservation system. Once reserved, the machines of the Grid are exclusively owned by the user (dedicated machines). However, the network is not dedicated, it is shared by the different users of the Grid. Therefore, up to 1600 grid nodes are involved in the experimentations according to their availability.

In the following, we experimentally evaluate the ability of H-B&B to deal with the

large scale issue of grids. Indeed, we evaluate the impact of the size of the computational pool on the H-B&B deployment CPU time cost. In addition, we evaluate the benefit of the adaptive feature of the masters on the performance of H-B&B. We also evaluate the ability of our approach to minimize bottlenecks. Finally, we evaluate the impact of the dynamic distributed decomposition on the improvement of the execution times in large scale environment. The performance of H-B&B is compared to single-level HMW-based B&B (*1-H-B&B*) and MW-based B&B (*MW-B&B*) approaches. The 1-H-B&B is developed according to the characteristics of the one developed by Di-Costanzo *et al.* [CBCM07]. Indeed, a single Main-Master manages a set of sub-masters localized on the different sites of the grid, and each of them manages a set of workers belonging to the same site.

3.5.1 Study of the scalability: H-B&B vs. 1-H-B&B and MW-B&B

Deploying efficiently distributed applications in large scale environments is a great challenge. The deployment cost includes: the initial launching of H-B&B masters and workers on the grid infrastructure, the construction of the hierarchy, and the distribution of tasks among workers. Each site of Grid'5000 is composed of at least a frontend machine (from which the user interacts with the Grid), an administration machine (for system services like NFS) and a pool of computational processor nodes. Proactive is installed on the frontend of each site involved in the experiment. It is then used to deploy locally (using NFS) the code of our H-B&B framework on the different processor nodes involved in the experiment and located in its site.

Figure 3.15 shows time taken to deploy a number of grid nodes by a simple MW-B&B, by a single level HMW approach (1-H-B&B) and by H-B&B using the group sizes (10, 50, and 100). To obtain more representative results, 10 runs are performed for each approach and the average values and the confidence intervals are computed. The figure shows that the two hierarchical approaches (H-B&B and 1-H-B&B) are much faster than MW-B&B in terms of deployment time and H-B&B (using different group sizes) is clearly more efficient than both the MW-B&B and the 1-H-B&B approaches. For example, H-B&B deploys 1500 grid nodes in only 70 seconds, whereas 1-H-B&B deploys them in 900 seconds and MW-B&B deploys them in 1700 seconds. 1-H-B&B gives an acceptable deployment time up to 600 nodes but the number of deployed nodes over time degrades beyond 600 nodes. MW-B&B gives the worst performance in terms of deployment times where the number of deployed nodes decreases in time. This major improvement can be explained by the distribution of the deployment load (launching nodes, decomposition of tasks, and their distribution) among the whole masters in the different levels of H-B&B, whereas the same work is dispatched by the reduced number of masters of the first level in 1-H-B&B and all the work is done by the centralized master in MW-B&B.

Eliminating bottlenecks is another challenge in large scale environments. The main function overloading the central master in MW-B&B is the task distribution when the

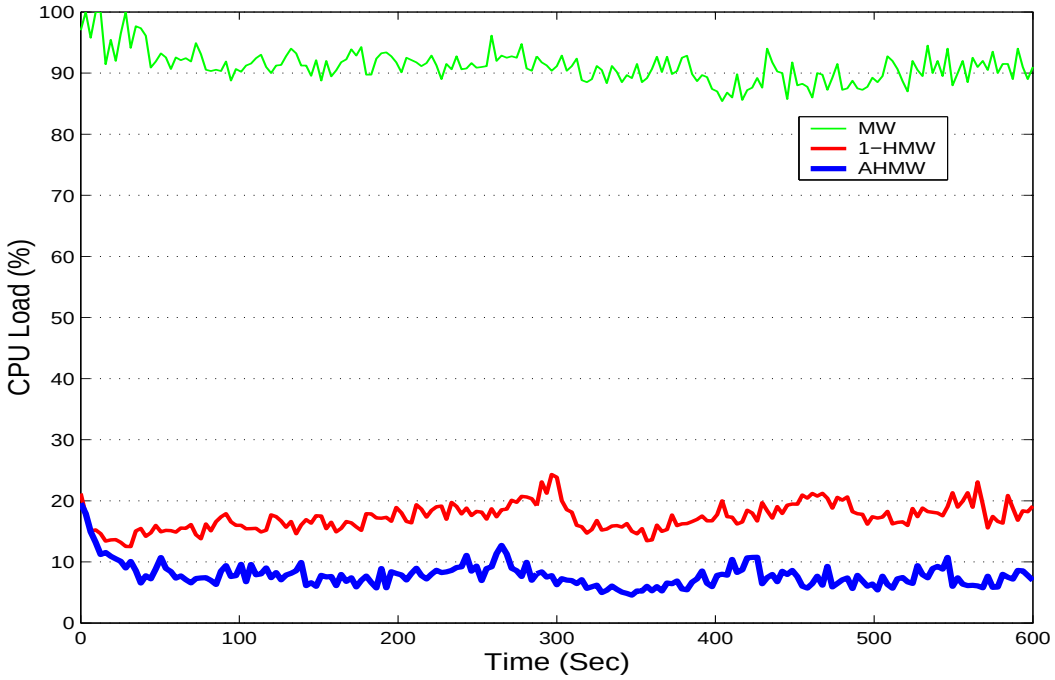

Figure 3.14: Evolution over time of the average CPU load on the master(s) of H-B&B, 1-H-B&B and MW-B&B

number of workers becomes important. In H-B&B, this function is distributed among the different masters in the hierarchy so that they are alleviated. Figure 3.14 depicts the evolution over time (10 minutes) of the average CPU load recorded on the master(s) for the three approaches H-B&B, 1-H-B&B, and MW-B&B for the same number of computational nodes (1590). The load presented in the figure concerns only the load generated by the approaches ignoring all additional load of the system. Ten runs are performed to solve the 10×20 Taillard's instances [TAI93]: Ta021 to Ta030 defined by 20 jobs and 20 machines, and the averages are computed. First, the master of MW-B&B presents a bottleneck and its load is around 90% during all the execution lifetime. Second, the masters of 1-H-B&B and H-B&B are underloaded compared to MW-B&B. Therefore, the time wasted by the workers of the two hierarchical approaches waiting for work is decreased compared to the MW-B&B one. However, masters of 1-H-B&B are more loaded compared to those of H-B&B. Indeed, the load of the masters of 1-H-B&B is around 20% whereas, it is only around 10% for the H-B&B making the masters of H-B&B more alleviated than those of 1-H-B&B.

3.5.2 Tuning of the group size parameter

The group size has a great influence on the number of deployed nodes over time. Figure 3.15 show that for more than 200 computational nodes, the H-B&B approach is by far more efficient than 1-H-B&B and MW-B&B especially for small (around 10) values of the group size. Indeed, H-B&B allows to deploy 1500 grid computing nodes

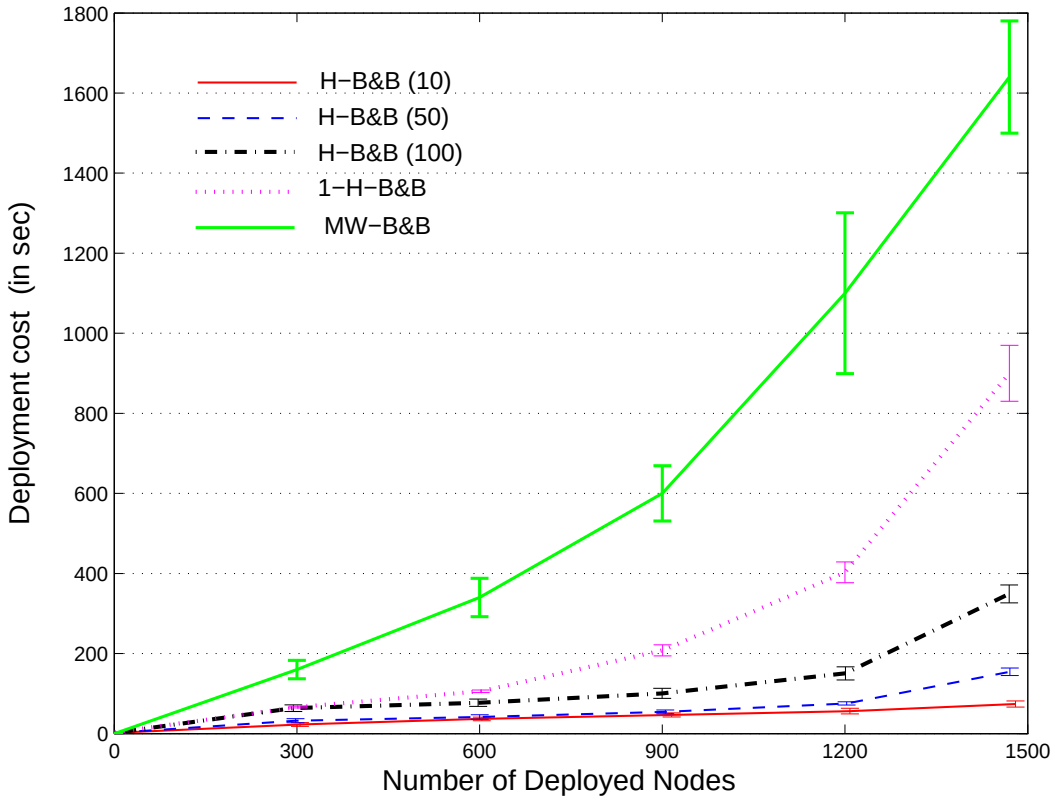

Figure 3.15: Evolution of the deployment cost for different sizes of the computational pool. H-B&B (using different group sizes) is compared to 1-H-B&B and MW-B&B.

in only 70 seconds using group size 10 whereas they are deployed in 150 seconds, and 350 seconds using respectively the group sizes 50, and 100. Therefore, the deployment cost grows with the group size. With a larger scale (number of computational nodes), the cost is highly important. In this case, for problem instances of several dozens of minutes the framework would spend much more time in deploying the hierarchy than in solving the problem at hand. Moreover, when using small group sizes, the number of deployed nodes using H-B&B grows exponentially making it insensitive to the size of the computational pool. That is to say whatever the size of the computational pool, the whole nodes will be launched, and assigned their tasks after their decomposition in just few seconds.

To evaluate the pertinence of the value 10 of the group size, on the one hand, we have studied the impact of the group size around this value (values 5 and 20) on the time cost of the deployment of the hierarchy. On the other hand, to evaluate the robustness and sensitivity of the group size to new applications we have considered 15 Taillard's problem instances with three classes representing three different sizes: small (20 jobs on 20 machines), medium (50 jobs on 20 machines) and large (100 jobs on 20 machines). Each problem instance can be seen as a new application as it is represented at execution by an irregular B&B tree completely different in size and shape.

Instance Size	Instance	Group size					**Average**
		5	10	20	50	100	
20×20	Ta021	55,60	65,57	93,00	168,25	319,00	**140,28**
	Ta022	58,00	72,50	103,00	137,00	238,00	**121,70**
	Ta023	62,00	78,00	105,00	172,00	280,33	**139,47**
	Ta024	64,00	81,00	105,00	161,00	277,00	**137,60**
	Ta025	61,00	75,00	103,00	197,00	326,00	**152,40**
	Average	**60,12**	**74,41**	**101,80**	**167,05**	**288,07**	**138,29**
	Stan. dev.	**3,33**	**5,88**	**5,02**	**21,58**	**35,65**	**10,96**
50×20	Ta051	59,25	71,33	94,00	172,50	334,00	**146,22**
	Ta052	61,00	73,50	106,50	204,00	371,50	**163,30**
	Ta053	61,00	76,50	102,50	199,50	371,50	**162,20**
	Ta054	61,00	71,00	108,00	213,00	393,00	**169,20**
	Ta055	62,00	72,00	106,00	210,00	371,00	**164,20**
	Average	**60,85**	**72,87**	**103,40**	**199,80**	**368,20**	**161,02**
	Stan. dev.	**0,99**	**2,25**	**5,63**	**16,13**	**21,30**	**8,70**
100×20	Ta071	174,00	66,33	92,25	174,66	347,66	**170,98**
	Ta072	285,00	76,66	105,50	215,66	394,33	**215,43**
	Ta073	186,00	75,00	106,00	214,00	387,00	**193,60**
	Ta074	200,00	75,00	106,00	213,00	402,00	**199,20**
	Ta075	180,00	75,00	103,00	214,00	426,00	**199,60**
	Average	**205,00**	**73,60**	**102,55**	**206,26**	**391,40**	**195,76**
	Stan. dev.	**45,75**	**4,13**	**5,89**	**17,69**	**28,51**	**16,06**
Average		**108,66**	**73,63**	**102,58**	**191,04**	**349,22**	**NA**
Standard deviation		**74,66**	**4,08**	**5,16**	**24,75**	**53,15**	**NA**

Table 3.2: Deployment times of H-B&B using different group sizes and different instances classified by their sizes.

Intensive experiments have been carried out using 1450 processing cores and the obtained results are reported in Table 3.2. For each of the 15 problem instances, the deployment time cost expressed in seconds (averaged over 10 runs) is reported for different group sizes (5, 10, 20, 50 and 100). Table 3.2 is organized as follows: the first and second columns represent respectively the size of the Taillard's instance and its name. Columns 3 to 7 represent the deployment time cost obtained using the different group sizes, and the last column shows the average values for each problem instance using the different group sizes. The average value for a class of instances using the same group size is presented at the last row of the class.

According to the obtained average values of the deployment time for each class of instances (at the last line of each instance family), we can notice that the size of the instance has not a significant impact on the deployment time cost. Indeed, the deployment time is approximately the same for all the instances when using the group sizes 10 and 20 (respectively $73,63$ and $102,58$ seconds) with a small standard deviation (respectively $4,08$ and $5,16$) and it grows softly when using the group sizes 50 and 100 (from 167 to 206 for the group size 50 and from 288 to 391 for the group size 100) with a larger standard deviation (respectively $24,75$ and $53,15$). However, this is not the case for the group size 5 for which the deployment time grows significantly. This may be caused by the fact that when using the group size 5 the sub-MWs take more time in decomposing and distributing tasks than in the launching of their children.

Instance	SH-B&B	H-B&B
Ta021	20779763020	22804681476
Ta022	3701361244	4552674332
Ta023	6012176904	7214612284
Ta024	23552625964	25833960748
Ta025	15154613724	17265836556
Ta026	33968795636	39877981352
Ta027	5822311444	6065503812
Ta028	7159046172	7509392584
Ta029	2596539776	3635155684
Ta030	8121546708	8365193108
Average	12711242699	14288134554

Table 3.3: The number of explored B&B tree nodes by H-B&B using the adaptive behavior of the masters compared to SH-B&B with static roles of the masters. H-B&B outperforms SH-B&B on all the instances.

This encourages the newly launched processes to request more rapidly tasks from their parent delaying the forwarding of the requests to integrate grid nodes in the hierarchy to the lower levels, which is necessary to build the rest of the hierarchy (see Section 3.2.2). Therefore, the launching of the processes located at the lower levels is delayed. We can conclude that the use of the group sizes 10 and 20 provide more efficient results. The group size 10 enables more robust execution in terms of standard deviation. The experimental results presented in the following sections are obtained using a group size fixed to 10.

3.5.3 Study of the adaptive feature

As mentioned in Section 3.2.2, one of the features of AHMW is that a master at any level of the hierarchy can change its behavior when the number of generated tasks becomes high. It means that as any worker the master contributes to the exploration process (not limited to dispatching work units).

Figure 3.16 shows the behavior of a sample process of H-B&B. The figure is composed of two graphics. The graphic at the top shows the evolution of the task request frequency over time and the graphic at the bottom shows the behavior of the sample process according to the received task frequency. At the beginning every process has the role of a worker. After it receives the first task request, it becomes worker and master at a time. When the task frequency exceeds the considered threshold (fixed here to 40 requests per second) it becomes a pure master and viceversa.

To evaluate the benefits of such adaptive feature, H-B&B has been experimented using 1500 Grid'5000 processing nodes and considering 10 Taillard's problem instances (20 jobs on 20 machines). Two versions of HMW have been considered: H-B&B and SH-B&B, which designate respectively Adaptive HMW and Static HMW. In SH-B&B,

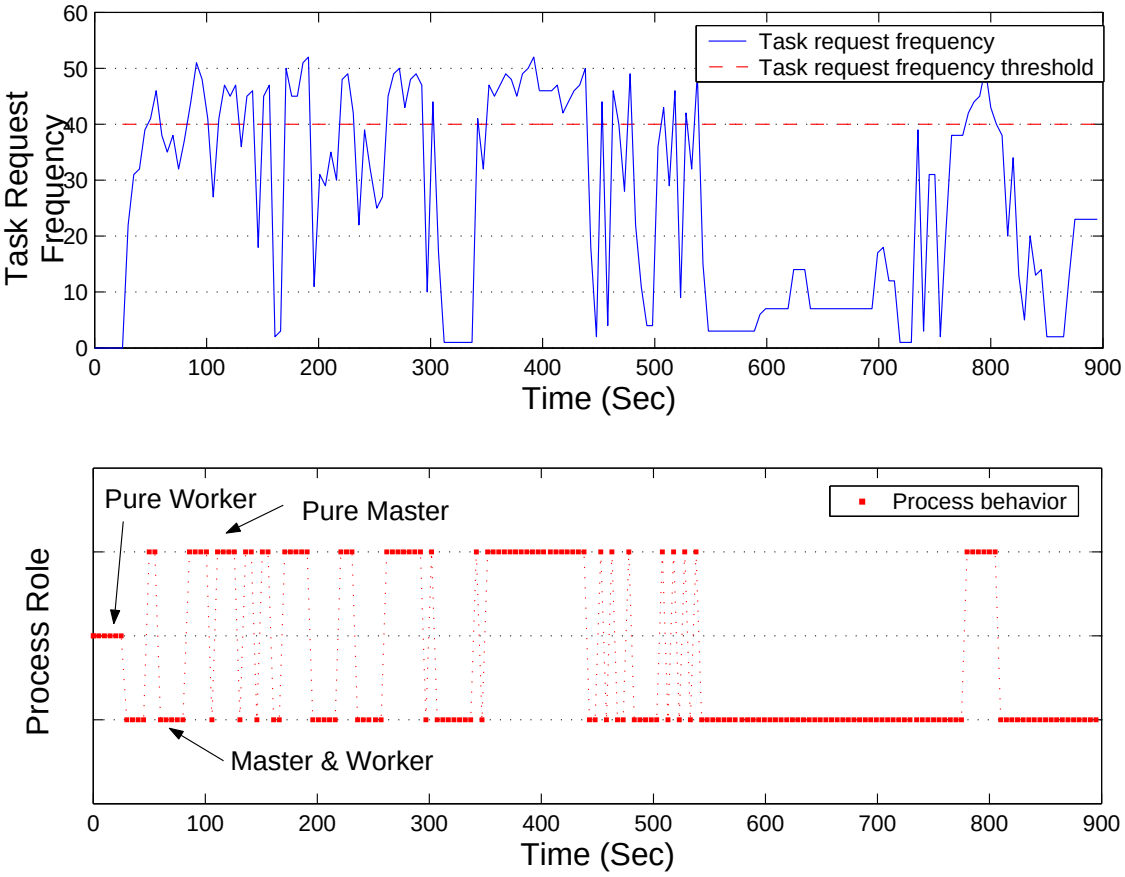

Figure 3.16: Adaptive roles of H-B&B processes.

the masters do not contribute to the exploration process whatever the load is. Table 3.3 reports the number of explored B&B tree nodes using H-B&B and SH-B&B for each problem instance. The results show that the use of the adaptive feature of masters is beneficial and H-B&B outperforms SH-B&B on all the experimented problem instances. Indeed, in average H-B&B explores more than 1,5 billion of B&B tree nodes more than SH-B&B.

3.5.4 Study of the efficiency: H-B&B vs. 1-H-B&B and MW-B&B

Table 3.4 allows us to evaluate the efficiency and effectiveness of the H-B&B compared to 1-H-B&B and MW-B&B. The reported results have been obtained within 10 minutes of execution of the ten 20×20 Taillard's instances Ta021 to Ta30. Column 2 reports the best solution found after 10 minutes by H-B&B. Columns 3, 4 and 5 report the CPU times (in seconds) required exactly to find the corresponding solutions in column 2 by respectively H-B&B, 1-H-B&B and MW-B&B. The task grains handled by the approaches are 12 to 18 for H-B&B and 16 for both 1-H-B&B and MW-B&B. For all

Instance	Best found sol	H-B&B	1-H-B&B	MW-B&B
Ta021	**2317**	368	2236	2981
Ta022	**2120**	368	392	760
Ta023	**2336**	118	287	565
Ta024	**2223**	680	1726	2101
Ta025	**2307**	452	557	866
Ta026	**2254**	341	660	742
Ta027	**2273**	366	940	1298
Ta028	**2220**	414	905	1222
Ta029	**2301**	344	476	960
Ta030	**2178**	445	967	955

Table 3.4: Execution times exactly required to H-B&B (column 3), 1-H-B&B (column 4) and MW-B&B (column 5) to provide the solutions reported in column 2

instances, H-B&B finds the best solution more quickly demonstrating its efficiency over 1-H-B&B and MW-B&B. Masters of H-B&B perform a distributed decomposition on the tasks and reach more rapidly fine-grained tasks (subproblems of size 12) which are distributed among the workers. Whereas, in 1-H-B&B and MW-B&B, the decomposition operation takes more time and the workers take more time waiting their tasks, the fact that affects the whole execution time. Indeed, the use of several levels of masters in H-B&B is beneficial and allows to find more quickly the solutions than 1-H-B&B and MW-B&B.

It is shown previously that the masters of H-B&B are alleviated and are not subject to bottlenecks. Therefore, workers do not waste time waiting for tasks. In addition, the dynamic distributed decomposition improves the performance by minimizing the waiting time of workers. In this experiment, we evaluate the ratio R between the effective execution time t_{Exec} and the idle time t_I recorded on the different workers. The idle time includes the communication time t_{Com} and the time t_{Dec} a master takes to decompose its tasks.

$$R = 100 \times \frac{t_{Exec}}{t_{Exec} + t_I}$$
$$t_I = t_{Com} + t_{Dec}$$

Table 3.5 shows the efficiency obtained using H-B&B compared to that obtained using 1-H-B&B and MW-B&B approaches. Ten runs are performed solving the 20 jobs and 20 machines Taillard's instances. The table contains 3 sub-tables representing the results of H-B&B, 1-H-B&B, and MW-B&B respectively. Within each sub-table, the resolution time (t_{Exec}), the idle time (t_I), and the efficiency (R) are represented. As shown in the table, H-B&B outperforms 1-H-B&B and the two hierarchical approaches outperform MW-B&B. Indeed, H-B&B presents on average an efficiency (99,02%) with an insignificant standard deviation (0,68). Whereas, the average efficiency for 1-H-B&B is 92,63% with the standard deviation 5,57. The efficiency of MW-B&B is only 83,43% with the standard deviation 6,61. The workers of H-B&B spend 99% of their time

solving their tasks and less than 1% in communication and waiting for task distribution. In the point of view of the masters, 1% of their time is spent in decomposition and distribution of tasks to their children. H-B&B remains insensitive to the size of the computing pool because whatever the size of the computing pool, the masters of H-B&B only manage the workers in their sub-MW and the overhead caused by the huge number of computing nodes is dispatched over the masters on the different levels.

Instance	H-B&B			1-H-B&B			MW-B&B		
	Solving t_{Exec}	Idle t_I	Efficiency R %	Solving t_{Exec}	Idle t_I	Efficiency R %	Solving t_{Exec}	Idle t_I	Efficiency R %
Ta021	54989,24	238,43	**99,56 %**	61192,18	1054,27	98,30 %	37489,82	2205,08	94,56 %
Ta022	12561,80	688,14	**99,46 %**	13885,57	1383,34	90,94 %	09834,28	0277,09	81,53 %
Ta023	17063,36	348,10	**98,00 %**	11343,88	1420,56	88,87 %	09003,14	2326,29	79,46 %
Ta024	58801,92	172,30	**99,70 %**	68724,53	1001,86	98,56 %	33148,90	2119,46	93,99 %
Ta025	10369,25	160,87	**98,48 %**	06549,61	1445,64	81,92 %	08075,39	2258,39	78,14 %
Ta026	23692,37	096,17	**99,78 %**	46969,42	1153,09	97,60 %	08704,40	2148,58	80,20 %
Ta027	43703,89	263,74	**99,28 %**	34479,40	1231,02	96,55 %	12290,16	2125,87	85,25 %
Ta028	13064,99	110,09	**98,79 %**	12791,99	1397,62	90,15 %	09319,94	2131,10	81,39 %
Ta029	21986,17	367,40	**98,13 %**	13561,35	1372,41	90,81 %	07159,49	2212,75	76,39 %
Average			**99,02 %**	Average		92,63 %	Average		83,43 %
Standard Deviation			0,68 %	Standard Deviation		05,57 %	Standard Deviation		06,61 %

Table 3.5: Efficiency of H-B&B compared to 1-H-B&B and MW-B&B – H-B&B clearly outperforms 1-H-B&B and MW-B&B in terms of average efficiency

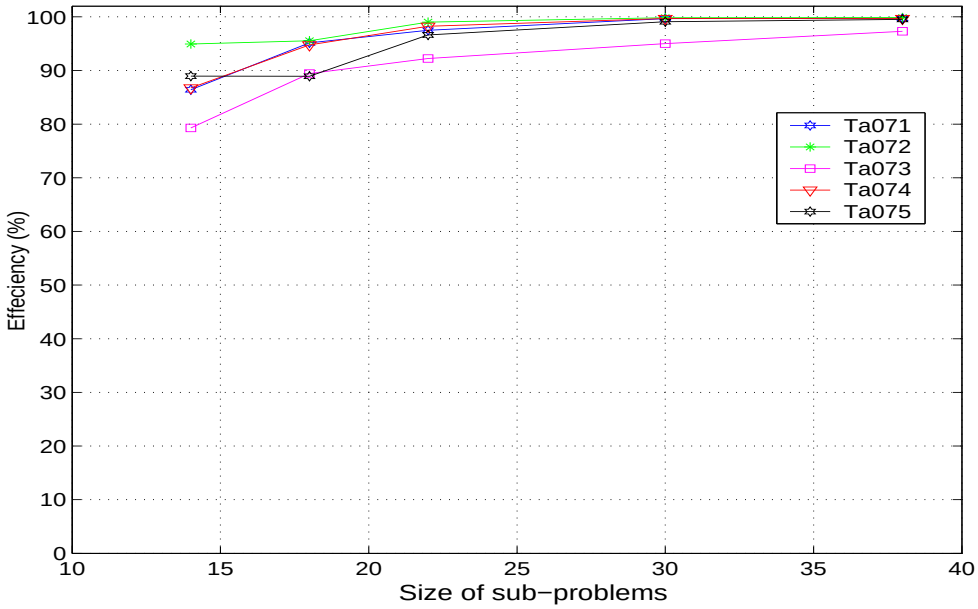

Figure 3.17: Impact of the granularity on the efficiency of H-B&B

3.5.5 Impact of the granularity on the efficiency of H-B&B

Figure 3.17 shows the efficiency of execution times of H-B&B obtained on 1164 grid nodes solving the FSP instances Ta071 - Ta075 described above using different grain sizes. They are obtained by varying the SBFS degree making masters generating different subproblem sizes. The grains represented in the graph are those recorded at the level of final workers. We tested subproblems of sizes: 14, 18, 22, 30, and 38.

The curves show that the efficiency is less than 90% for most of the instances when the grain size is lower than 15. It is around 95% when the grain size is lower than 18 and exceeds 99% beyond the grain size 22. To choose the appropriate granularity, we should also take into account the work request frequency and a tradeoff must be found. Therefore, regarding these results and those above, for an efficient execution of H-B&B, we can chose tasks of grain size between 18 and 30. Considering fine grained tasks using H-B&B allows one to achieve an efficiency of execution similar to the one obtained using coarse grained tasks on a MW-B&B. The exception is the instance Ta073 which gives relatively bad efficiency of execution compared to the others. It is probably due to the search space landscape of the instance which is easy to explore. That makes workers take a short time to explore the subproblems, thus the masters receive more frequently work requests slowing them and workers spend more time acquiring subproblems.

3.6 Conclusion

In this chapter, we have presented a new HMW model (*AHMW*) in order to make highly scalable parallel algorithms for solving very large size combinatorial optimization problems. AHMW has been used to develop an adaptive hierarchical MW-based B&B (H-B&B). The new presented scheme allows to prevent from bottlenecks likely to be created at the level of the central master process in the conventional MW paradigm. The hierarchy built by our approach is multi-layered and dynamic adaptive. Indeed, it evolves over time with the arrival of computing resources in the grid, and allows generating and handling tasks of different grain sizes. H-B&B is composed of three types of processes: a super-master, masters and workers. Each process has its own role with the ability to change its behavior from master to worker and *vice versa* according to the availability of resources. The strength of our approach resides in the collaboration of all the processes of H-B&B to perform the different functions such as the construction of the hierarchy, the decomposition of tasks, and their distribution. Therefore, the load is shared between all the processes. Processes of H-B&B consider tasks of different grain sizes allowing to improve the efficiency of our approach by minimizing bottlenecks at the level of masters and reducing wasted time at the level of workers.

Different large scale experiments of our approach, implemented on top of ProActive, have been performed using the Grid'5000 real grid hardware infrastructure. The different experiments demonstrate the efficiency of H-B&B compared to existing single-level HMW (1-H-B&B) and more significantly compared to the classical MW-B&B in terms of scalability. In fact, H-B&B deploys nodes much faster thanks to the participation of all the launched processes in the parallel deployment. This makes the number of deployed nodes increasing exponentially in time remaining insensitive to the large scale number of resources. The adaptive nature of the masters' behavior demonstrates its strong impact on the efficiency of H-B&B. Indeed, the contribution of the masters in the exploration process allows to gain in terms of computing power. The average CPU-load on masters is minimized and prevents from the creation of bottlenecks among the masters. In H-B&B, the masters are assisted in the management of work requests and in the decomposition to achieve acceptable grain size of the tasks. This allows them to distribute tasks much faster without blocking the workers requesting tasks. Therefore, the workers spend less than 1% of the total time waiting for tasks from their parents. From the point of view of the masters, this time (1%) represents mainly the decomposition time to achieve medium and small task grains, which facilitates the work control and minimizes the waiting time of workers.

Chapter 4

FTH-B&B: A Fault Tolerant Hierarchical B&B

4.1 Introduction

In the previous chapter we have presented H-B&B, a hierarchical MW-based B&B, together with experimental results showing its ability to deal with the scalability issue of grids. However, scalability is not the unique issue to take into account when designing grid-based B&B. In fact, the second major issue of grids is Fault Tolerance (FT). The grid computational resources are highly unreliable and volatile. They are unforeseeable, they frequently join and leave the system. Moreover, in an environment including several thousands of resources, the appearance of faults is unavoidable [HT05]. Recent experimental studies have shown that jobs submitted by users to large scale, multi-institutional Grid infrastructures often fail to complete successfully. For example, data collected and analyzed by the WISDOM project [WISD], which submits tens of thousands of jobs to the EGEE/EGI infrastructure [EGEE] [EGI] indicate that only the 65% of submitted jobs are executed successfully, therefore, the rate of failure is 35%.

FT can be achieved at two levels: *application-level* or *middleware-level*. Several middlewares provide FT mechanisms to reliable execution of applications: ProActive [PROA, BBC+06, CDCL06], XtremWeb [XTRE], and Condor [CON, GSDB09, PL96]. Nevertheless, they are generally costly in terms of CPU execution time and slow down the application. The second strategy is application-level and consists in including FT mechanism(s) into algorithms [MMT07a, MMT07b, IAM00, FM87, GKLY00, GLY00, DVC+09]. Three major aspects must be taken into account when designing FT at application-level: First, one must ensure fault recovery to avoid the loss of work units and to gain in terms of execution time by minimizing the redundant work. Second, one must ensure to maintain the same topology (formed between the different processes of the B&B) and avoid orphan branches during the lifetime of the algorithm in order to guarantee a valid functioning and then a valid result of the algorithm. Third, one must ensure an efficient restart of a great number of failed processes.

In this chapter, we propose our third contribution which is the design and implementation of a Fault Tolerant Hierarchical B&B (*FTH-B&B*) dedicated to large scale

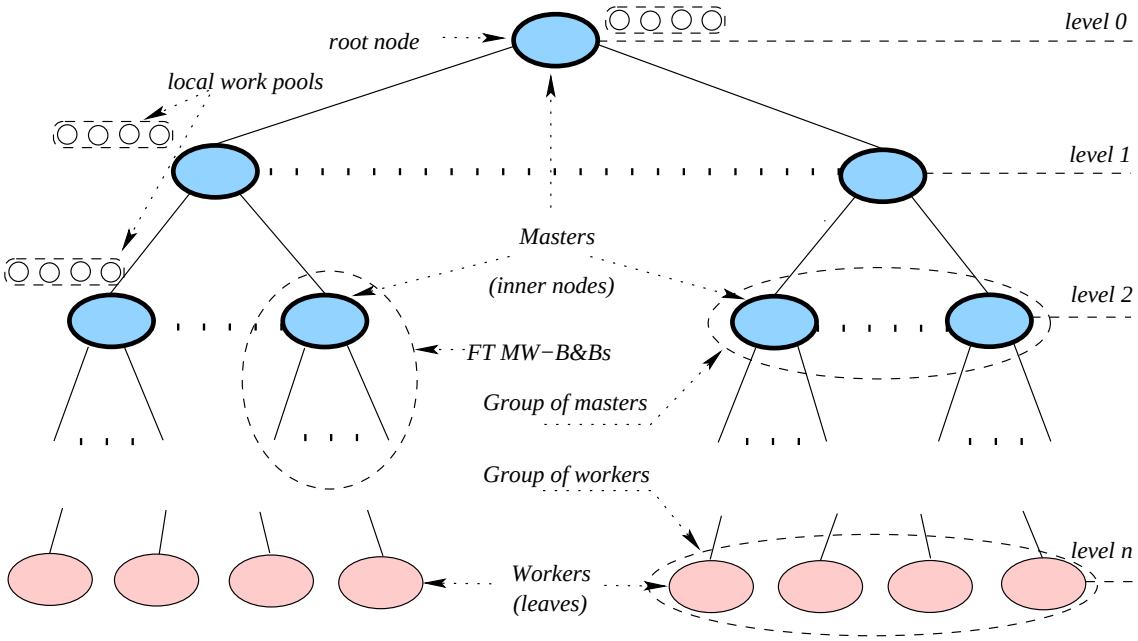

Figure 4.1: General design of FTH-B&B. Several fault tolerant sub-B&Bs are organized hierarchically where inside each sub-B&B one master manages several workers.

unreliable environments such as computational grids. FTH-B&B, also based on *P2P-B&B*, is an application-level distributed FT mechanism composed of several FT MW-B&Bs organized hierarchically into groups. A fault recovery mechanism is introduced to avoid the loss of work units and to improve efficiency in terms of execution time. Indeed, work units are stored so that one can recover at each instant the subproblems assigned to failed processes using a *3-phase communication protocol*. Moreover, our approach ensures to maintain a balanced and safe hierarchy during the lifetime of the algorithm in order to guarantee a valid functioning. Finally, an efficient restart of the application is ensured by a distributed checkpointing mechanism in case of failure. This work has been published in [BMT11a].

The remainder of this chapter is organized as follows: In Section 2 we present the design of FTH-B&B, the work management, task recovery, maintenance of the hierarchy, and finally a distributed checkpointing mechanism. Section 3 details the implementation of the fault tolerance features using ProActive. The performance evaluation of the approach is studied in Section 4. We conclude the chapter in Section 5.

4.2 Architecture and Working of FTH-B&B

The proposed FTH-B&B is a fault tolerant parallel B&B algorithm based on the Hierarchical Master/Worker paradigm in order to deal with the fault tolerance issue while ensuring scalability in large scale environments. The proposed approach is composed of several fault tolerant Master/Worker-based sub-B&B algorithms (see Figure 1), where inside each sub-B&B one master manages several workers and performs failure recovery.

The sub-B&Bs are launched in parallel and act independently on different subproblems. They are organized hierarchically into several levels. The root node and the inner nodes of the hierarchy designate masters and the leaves designate workers. The masters perform a parallel recursive branching in order to decrease the size of subproblems until reaching sufficiently fine-grained subproblems which can be explored sequentially by the workers. In addition, they locally store the branched and assigned subproblems for further rescheduling. Each sub-B&B is composed of one master and a group of workers that can also be masters for another sub-B&B at the lower level. Each sub-B&B is developed using the framework AHMW [BMT12] (see Chapter 3), therefore based on (*P2PBB*) presented in [BMT09] (see Chapter 2) where one master manages a dynamic group of communicating workers. Therefore, the algorithm is composed of communicating groups of fault tolerant Master/Worker-based sub-B&B processes.

FTH-B&B is designed to run on computational Grids offering huge amount of computing resources which are highly unreliable. Therefore, for a safe execution of the algorithm, any failure must be detected and handled. The fault detection is the responsibility of all the processes of the algorithm. Both masters and workers are responsible to detect failures of the process they depend on or which depends of them. Masters send heartbeats to their children every Heartbeat Period *HP*. If a worker is dead or suspected to have failed, it is removed from its list of children. The unexplored part of its subproblem is saved and rescheduled to another free safe worker (see Section 4.3 for more details). Workers also send heartbeats to their masters every $HP \times G$, G being the size of the group forming one sub-B&B. The period of heartbeats is increased G times to avoid flooding their masters by their heartbeats.

In order to minimize the communication overhead, the masters do not inform their children about any worker failure. Within the same sub-B&B group, the workers do not heartbeat their neighbors but rather they detect failures when they broadcast their upper-bounds. When that happens, they remove the failed worker from their list of neighbors, so that they will never send them messages.

Let us recall that when executing FTH-B&B in a faulty environment, one must take into account three important points. First, one must ensure fault recovery to avoid the loss of work units and to gain in terms of execution time in minimizing the redundant work. Second, one must ensure to maintain the same topology during the lifetime of the algorithm in order to have a valid functioning of FTH-B&B. Third, one must ensure an efficient restart of a great number of FTH-B&B processes.

4.3 Work management with task recovery

Each master has its local work pool obtained by branching the subproblem assigned to it by its parent. A collegial strategy is adopted since multiple work pools are considered and each inner master has its own work pool. During the search, the local pool evolves continuously and when it is empty, the process sends a work request to the master at the upper level. A master receiving such a request consults its local work pool and asks

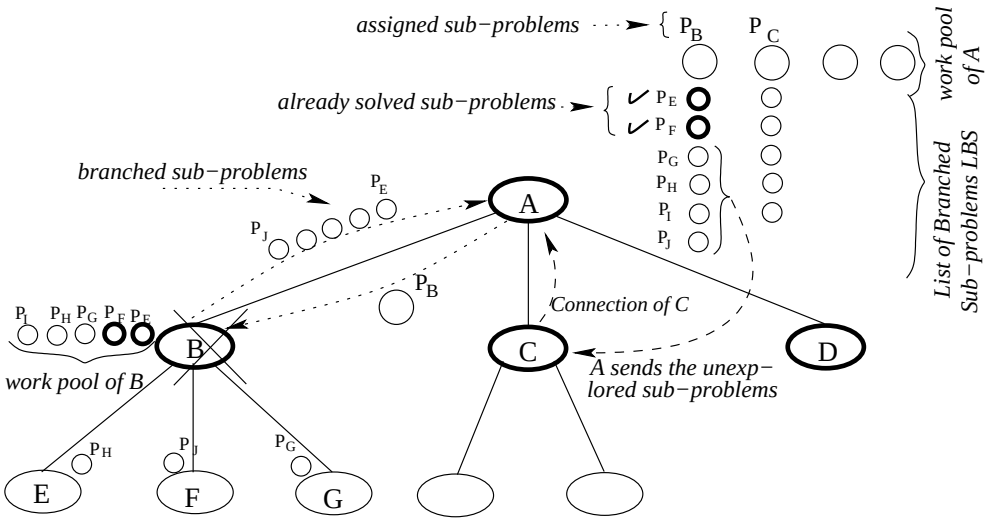

Figure 4.2: Work Management with fault recovery. The parent of a failed worker only reschedules the unexplored subproblems in order to minimize redundancy.

its upper level if its work pool is empty.

4.3.1 Fault recovery

Since no loss of work is tolerated in exact resolution of COPs, the masters manage a list of assigned subproblems, each element of the list is a mapping between the assigned subproblem and the process assigned to it. A subproblem has a unique identifier in the local work pool of the parent problem. When a failure is detected, a part of the subproblem is rescheduled to another safe worker. To avoid redundant responses, only the first result is taken into account.

In order to optimize the overall execution time and avoid redundant exploration of subproblems, when facing failures, the masters also manage a list of branched subproblems LBS. It represents the subproblems being explored by their grandchildren (see Figure 4.2). It is updated each time a grandchild finishes the exploration of its assigned subproblem. When the process exploring a subproblem fails, the master identifies the subproblem assigned to it and the branched subproblems in LBS and only reschedules the unexplored part. For example, in Figure 4.2, the subproblems $P_E, ..., P_J$ are the result of the branching of the problem P_B by the process B. B has already explored P_E and P_F (in bold in Figure 2) but not yet P_G, P_H, P_I and P_J. Therefore, when B fails, A will only reschedule P_G, P_H, P_I, and P_J to the newly connected process C. The size of LBS does not affect the masters because it is static and depends only on the size of the subproblem and the used branching method. Moreover, it is removed when the parent subproblem is totally explored and it is replaced by the new assigned one.

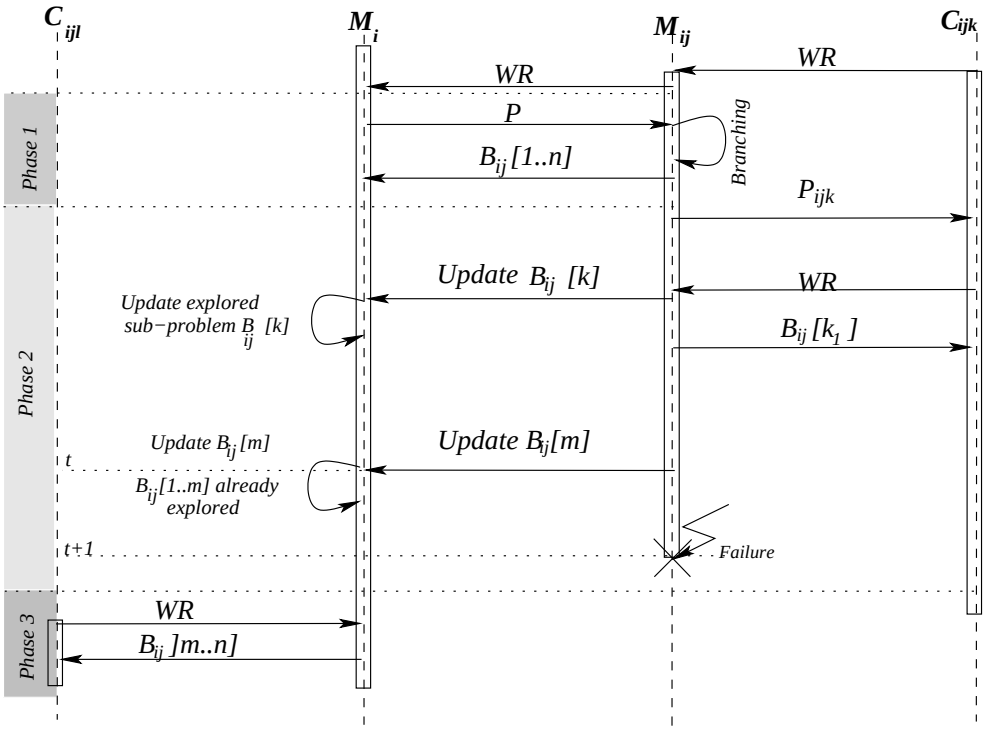

Figure 4.3: 3-phase communication sequence diagram

4.3.2 3-Phase communication mechanism

The task distribution is performed respecting a fault recovery mechanism using the proposed *3-phase communication mechanism* between the current master, its children and its grandchildren (see the sequence diagram in Figure 4.3).

- *Phase 1 (between a master and its children):* It allows a master to assign problems to its children and to receive back the branched subproblems. In the sequence diagram (see *Phase 1* in Figure 4.3), M_i assigns a problem P_{ij} to its child M_{ij}. This latter performs a branching and sends back the branched subproblems $B_{ij}[1..n]$ to its parent M_i. After that, it can assign the subproblems $B_{ij}[1..n]$ to its available children $C_{ij}[1..g]$.

- *Phase 2 (between a master, its children and its parent):* The *Phase 2* (see Figure 4.3) serves to update already explored subproblems. Each time a child C_{ijk} finishes the exploration of its subproblem $B_{ij}[k]$, M_{ij} informs M_i which updates $B_{ij}[k]$ the already explored subproblems of its grandchild C_{ijk}. Therefore, the master knows at any time the unexplored parts of a given problem

- *Phase 3 (between a master and a new free process): Phase 3* allows the master to reschedule the unexplored subproblems to another free safe worker if the process handling it has failed. For example, if after a period of time t, the children $C_{ij}[1..g]$ finish the exploration of $B_{ij}[1..m < n]$ subproblems. After $t+1$, even if M_{ij} fails,

M_i will only reschedule $B_{ij}]m..n]$ to another free safe process (see *Phase 3* in Figure 4.3).

The 3-phase communication mechanism allows the masters to minimize the execution of redundant work units, to speed up the exploration, and then to improve the overall performance. In order to obtain more precision about the already explored subproblems and to save more branched subproblems, the 3-phase communication mechanism can be extended to the i^{th} grand child. Nevertheless, new communication must be established between a master and all its i^{th} grandchildren tree at the expense of the communication overhead. Indeed, the amount of the transferred and saved subproblems grows in an exponential way when i increases. Therefore, a tradeoff must be found between the number of levels participating in the 3-phase communication mechanism and the number of saved subproblems, consequently, the amount of redundant work. In this work, only one level of masters participate in the 3-phase mechanism.

4.4 Maintenance of the hierarchy

It is important to maintain the same topology during the lifetime of the algorithm in order to get a reliable execution. Indeed, all communication and work management are based on the used topology (in our case the hierarchy). Additionally, failures can isolate some parts of processes from the rest of the hierarchy leading to loss of computing power and/or data. Moreover, at each instant, the hierarchy must be balanced.

Masters and workers are both subject to failures. A worker failure has not a great impact because it is located in a leaf of the hierarchy. In fact, no other process depends on it and its task can be partially rescheduled by its parent using the 3-phase mechanism. However, a master failure can isolate some parts of the hierarchy because the inner masters represent intermediary links. Indeed, when an inner master fails, the sub-B&B it represents crashes and the link between its descendants and the rest of the hierarchy is lost. Therefore, orphan branches may be created leading to the failure of the algorithm. Hence, it is necessary to rebuild the hierarchy by the creation of new links between the descendants and the ascendants of the failed master. In the following, every process p_i holds a list of all its ascendants $A_i[1..(l-1)]$, l is the level of p_i in the hierarchy. A_i contains at most $log_g(N)$ elements, N being the size of the computational pool and g the size of a sub-B&B group. Three rebuilding strategies are proposed: *simple connection to ascendants (SCA)*, *master election (ME)*, and *balanced hierarchy (BH)*.

4.4.1 Simple Connection to Ascendants (SCA)

In this strategy, when a master m fails, all its children $c_m \in C_m[1..g]$ connect to the closest safe ancestor A_m. In Figure 4.4, when D fails, its children E, F, and G connect to their closest safe ascendant T and consider all its children (C and B) to be their new neighbors. This strategy is simple to implement and to manage. Nevertheless, its main drawback is that the closest safe master rapidly becomes a bottleneck when

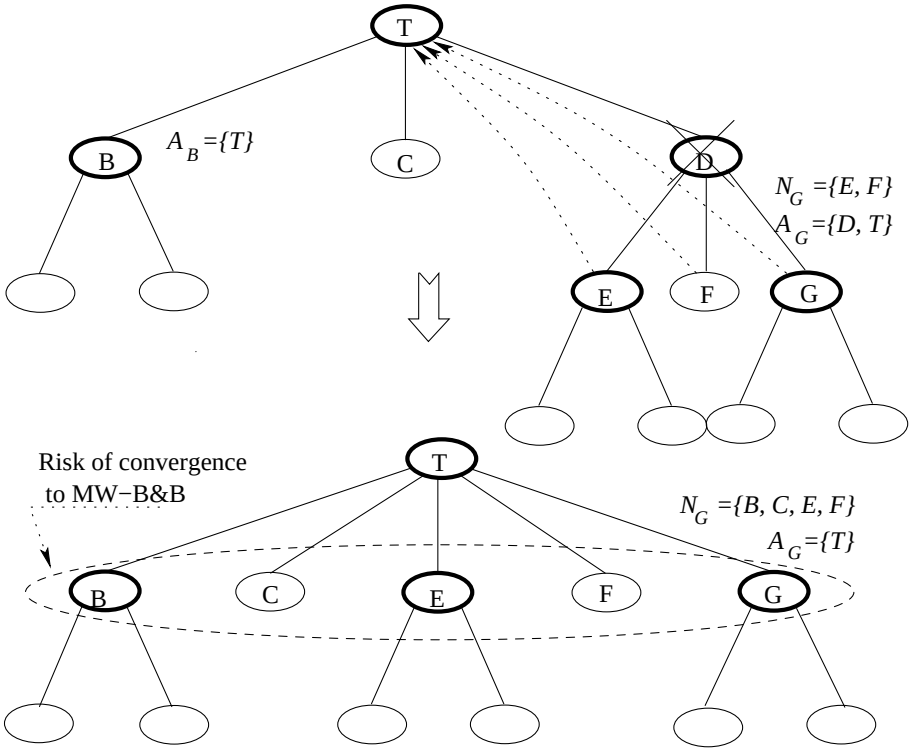

Figure 4.4: Maintenance of the hierarchy using *SCA*. When a master fails, the orphan workers connect to their closest safe ascendant. Risk of convergence to MW-B&B.

it is faced to several failures among its children. It receives an exponential number of connections in a short period of time. Indeed, if f masters (children of m) fail, it receives $g \times f$ connections from its grandchildren, g is the size of sub-B&B groups. Moreover, if one level of masters fail, it receives k^2 connections and if l levels fail, it receives g^{l+1} connections. Therefore, the FTH-B&B rapidly converges to a traditional MW-B&B over time as the number of failed masters increases.

4.4.2 Master Election (ME)

In this strategy, when a master m fails, the concerned orphan processes $p_i \in C_m$ elect a new master among of them using the bully algorithm [GAR82]. Each process has a unique identifier assigned to it at the moment of its creation. In this algorithm when the master fails, the process with the highest identifier is selected as follows: When a process p_i detects the failure of its master, it initiates an election. It sends an election message to all its neighbors with higher identifier. If no process responds, p_i becomes the master and announces its success to the other processes. If one of the processes answers, it means that there is at least a safe process which can be a master then p_i ends its election. When a process p_j receives an election message from a process p_i with a lower identifier, it answers and initiates a new election algorithm.

If the newly chosen master process p_i is a worker, it changes its behavior and becomes

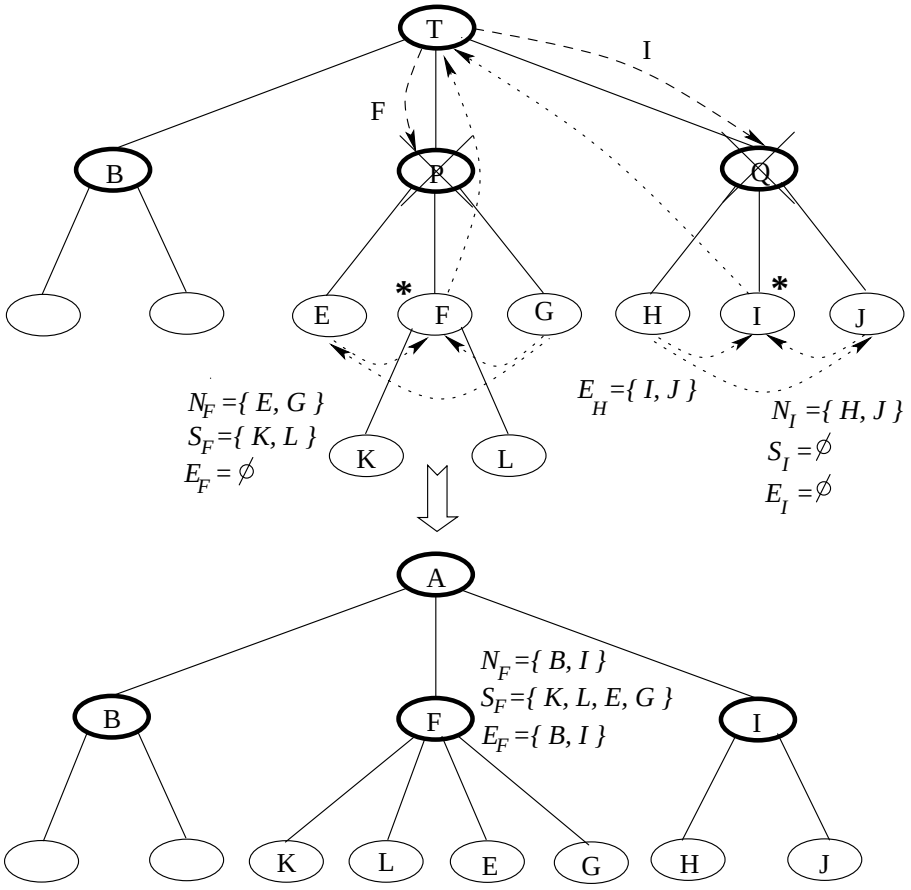

Figure 4.5: Maintenance of the hierarchy using *ME*. When a master fails, the orphan workers elect a new master which connects to its closest ascendant and considers its electors as its children.

a master and then it considers its old neighbors to be its children $C_i = C_i \cup N_i$, N_i is the list of its neighbors. C_i will contain both of its old children and its old neighbors. Each $p_j \in N_i$ and $p_k \in C_i$ update their neighbors $N_{j/k} = N_{j/k} \cup C_i$. After that, p_i connects to its closest safe ascendant $p_a \in A_i$. p_a informs p_i about its new neighbors $N_i = C_a$ and the list of the processes it will contact in case of a future election session $E_i = C_a$. For example in Figure 4.5, when P fails, E and G select F as a master and F considers them as its children $C_F = \{K, L, E, G\}$. After that, it connects to its ascendant T which informs it about its new neighbors $N_F = \{B, I\}$ and the new electors $E_F = N_F$. To avoid the same process to be always selected and consequently overloaded with the multiple orphan processes, we consider the lower identifier in the bully algorithm. Each newly elected master is assigned the highest identifier among its neighborhood.

4.4.3 Balanced Hierarchy (BH)

In this strategy, when a master m crashes, the orphan processes $p_i \in C_m$ migrate to another safe sub-B&B. Their migration is done according to a migration mechanism

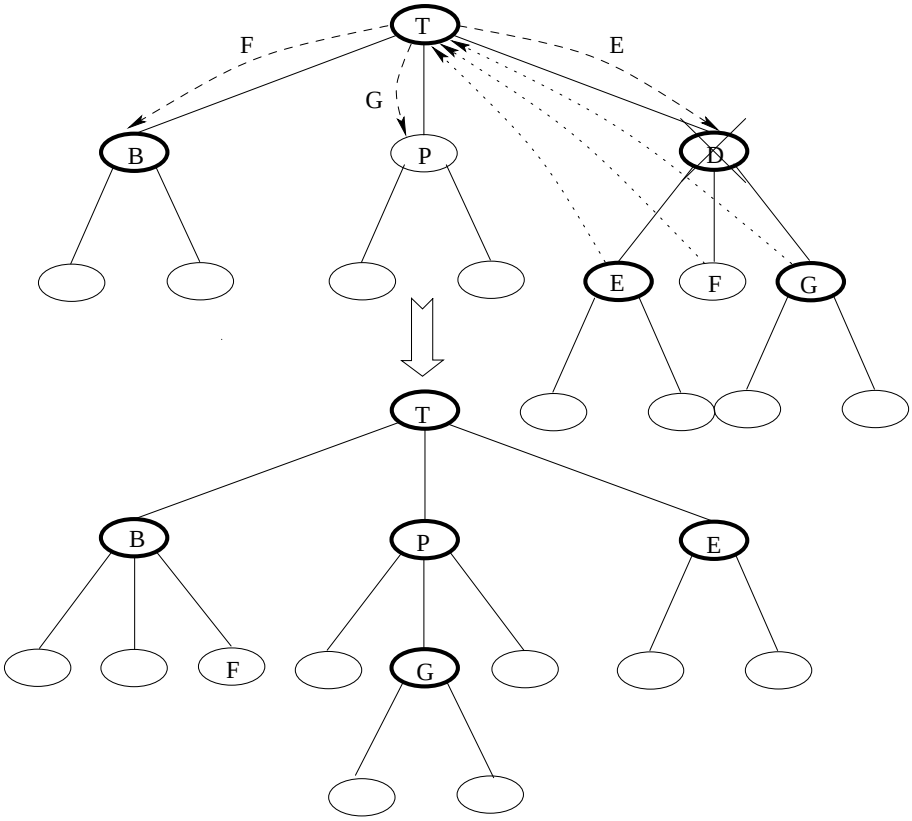

Figure 4.6: Maintenance of the hierarchy using *BH*. Orphan processes migrate to safe non-full sub-B&Bs respecting the constraint of the group size.

which assigns new orphan processes to a safe non-full sub-B&B. A sub-B&B is full if the number of processes composing it reaches the threshold group size a master can manage. The migration mechanism aims at avoiding the convergence of FTH-B&B to a traditional MW-B&B and to maintain the hierarchy balanced at each instant. That is achieved with the readjustment of the sub-B&Bs group sizes each time a master fails. An orphan process p_i connects to its closest safe ascendant $p_a \in A_i$ which holds p_i only if the group it manages is not full in order to maintain the hierarchy balanced. A process p_i receiving orphan processes from its parent also respect the constraint of the group size. Indeed, it dispatches the orphan processes among its children if the group it manages is full and so on until a non-full group is found. In Figure 4.6, when D fails, its children E, F, and G connect to T which holds only one node E and dispatches the others F and G respectively to its children B and P. Therefore, the hierarchy of FTH-B&B changes over time as soon as masters fail. This allows one to avoid creating new bottlenecks after several failures and prevents from the risk of convergence to a standard MW-B&B during the lifetime of the system. Moreover, at each instant the hierarchy is balanced whatever the number of failures since the new orphan processes are dispatched over all the hierarchy.

4.5 Distributed checkpointing

In a traditional MW-B&B, reliability can be achieved through checkpointing operations performed by the master process. However, this approach assumes that the master is reliable. In our approach, there exists several levels of checkpointing. Each master in the hierarchy performs distributed asynchronous checkpointing operations independently from others. This makes the whole algorithm more reliable even if some inner masters fail. Each master process manages a back-up file and saves a sequence of unexplored branches represented by a vector of partial solutions. When a master fails, the newly designated master reads the back-up file, reconstitutes the locally unexplored subproblems, and then the exploration process carries on from the last consistent local state of the sub-B&B. After a crash of the root master, a new instance of the algorithm must be launched. Then the new processes must reconstitute the unexplored subproblems from the different found back-up files and the exploration process cannot begin until a consistent global state is found. The reconstitution operation of the subproblems is illustrated in the following.

4.5.1 Reconstitution of subproblems

Even though a consistent local state of a sub-B&B is easy to find, finding a global one is not trivial. The processes must collaborate to determine a consistent global state of the system. At each instant of the execution of FTH-BB, the graph formed by the dependencies between the subproblems being explored must match with the one formed by the processes (the masters and workers). A consistent global state of the system means that the processes must reconstitute the subproblems respecting their dependencies before the crash. Therefore, the same graph of dependencies (hierarchy) must be reconstituted. Let I be the set of processes of FTH-B&B, p_i a problem being explored by the process i, C_i the children of i, and δ_{p_i} the set of subproblems obtained by the branching of the problem p_i. At each instant, the consistent global state means that the processes of the algorithm and the dependencies of the subproblems must satisfy the implication:

$$\forall \;\; i, j \in I, j \in C_i \Rightarrow p_j \in \delta_{p_i} \tag{4.1}$$

This implication means that whatever a process j being a child of i, it explores subproblems obtained by the branching of problems belonging to the work pool of its parent i. After a failure of all processes, a new instance of the system is launched and then a new hierarchy is formed. The masters retrieve the unexplored subproblems from the distributed back-up files. Some of the new processes (i and their children j) are launched on processors different from those on which they were launched before the failure. They retrieve subproblems belonging to other failed processes. Consequently, the dependencies between the new retrieved subproblems p_j form a random graph and they do not match with the initial hierarchy. Therefore, (4.1) can not be satisfied because $\exists i, j \in I, j \in C_i \mid p_j \notin \delta_{p_i}$. Moreover, there is no global information about the retrieved work. Therefore, the global set of unexplored subproblems cannot be known before reconstituting the subproblems according to their dependencies. To address this issue, we define 3 operators: *Gather*, *Reduce*, and *Deduce*.

4.5.2 Reconstitution operators

Gather allows one to gather the distributed subproblems and to aggregate them into several centralized points. Each process sends the retrieved subproblems to its parent which collects all its children subproblems. Let y_i be the subproblem retrieved by the process i and C_i the set of its children. The set of unexplored subproblems of i can be formulated as:

$$G_i = \bigcup_{j \in C_i} (y_j) \cup y_i \tag{4.2}$$

Reduce allows one to reduce several subproblems into their smallest root problem $p \neq P$ such that they can be obtained by the branching of p according to (4.3). P is the original problem. The reduction of two subproblems q and r is achieved using the equation:

$$R_i(q,r) = \begin{cases} p & \text{if} \quad \exists\, p \mid q, r \in \delta_p \wedge \\ & \quad \nexists\, p' \mid p' \in \delta_p \wedge q, r \in \delta_{p'} \\ r & \text{if} \quad q = \phi \\ q & \text{if} \quad r = \phi \\ \{q, r\} & \text{else.} \end{cases} \tag{4.3}$$

Each process uses *Deduce* described in (4.4) to deduce the set of the unexplored subproblems for each sub-B&B.

$$D_i = \prod_{p,q \in G_i} R_i(p,q) \tag{4.4}$$

The product ($\prod$) represents the application of the operator *Reduce* to all couples of subproblems (p, q) including the newly obtained root problems.

Using these 3 operators, the processes perform the reverse operation of the 3-phase communication mechanism described in Section 4.3.2. In the 3-phase communication mechanism, a master assigns a problem to its child and then the child sends back the branched subproblems. Here, a child sends subproblems to its parent and then the parent reconstitutes their root problem using the *Gather*, *Reduce*, and *Deduce* operators. After that, it reconstitutes its local work pool and the list of branched subproblems. The subproblems are sent as a list of couples (p, φ_p) such that $p \in D_i$ and $\varphi_p = \{G_i \cap \delta_p\}$. Each couple represents the root problem p in the set D_i and its corresponding set δ_p of retrieved subproblems obtained by the branching of p. From the point of view of the 3-phase communication mechanism, D_i represents the local work pool of the process i and G_i represents the list of the branched subproblems (LBS).

4.5.3 Consistent global state

The 3 operators allow to build gradually a consistent global state from the local ones. At each instant and after applying *Gather, Reduce*, and *Deduce* in this order by a

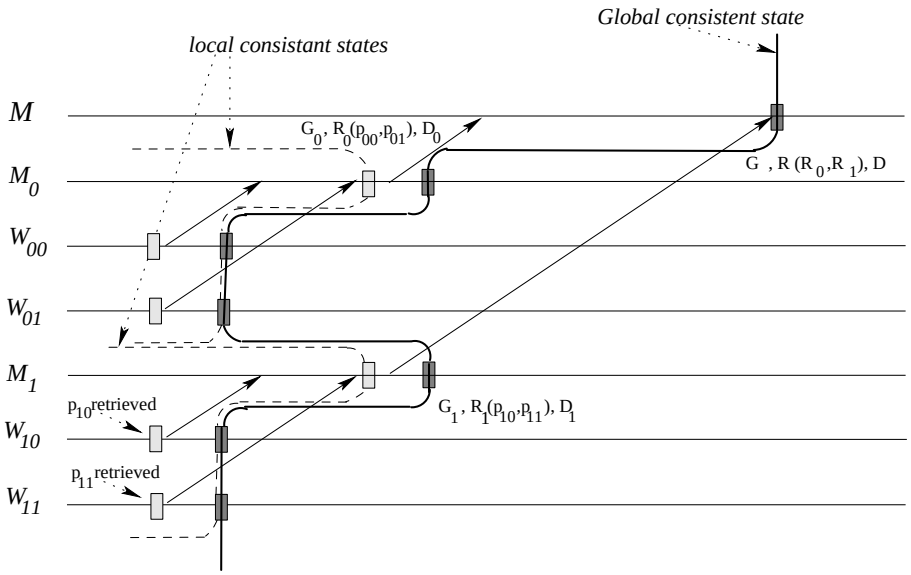

Figure 4.7: Consistent global state. Masters perform their checkpoints after they receive all the couples (p, φ_p).

process i:

$$\begin{aligned} \forall j \in C_i, p_j \in pool(j) &\Rightarrow p_j \in G_i \\ &\Rightarrow \exists p_i \in D_i, \exists p'_i \in G_i \ | p_i = R_i(p_j, p'_i) \\ &\Rightarrow p_j \in \delta_{p_i} \end{aligned} \tag{4.5}$$

These operators are applied on the processes from the leaves to the root master. All the reduced subproblems (roots of the retrieved subproblems) are obtained and the global unexplored work is deduced. The consistent global state can be built gradually (see Figure 4.7) until reaching the root master. Therefore, for the root master $\forall i, j \in I, j \in C_i, \Rightarrow p_j \in \delta_{p_i}$, i.e., Equation (4.1) is satisfied. In Figure 4.7, the checkpoints are represented by gray squares. They are performed when a master receives the couples (p, φ_p) from all their children.

A complete example of the use of the 3 operators is presented in Figure 4.8 where a Flow Shop scheduling problem is considered. A subproblem is represented by a vector $V[1..n]$ of placed jobs. The root problem of two subproblems $V_1[1..l]$ and $V_2[1..m]$ is a vector $V[1..k \leq l, m]$ such that $V_1 = VV'$ and $V_2 = VV''$, $V' = V_1]k..l]$ and $V'' = V_2]k..m]$. In the figure, the workers check their back-up files and then send the subproblems to their parents. For each master, the operators are applied and the couples $(p \in D_i, \{G_i \cap \delta_p\})$ are computed. The global unexplored subproblems are located at the level of the root master. For example, the workers W_{100} and W_{101} retrieve respectively the subproblems $sp_{100} = [7, 8, 2]$ and $sp_{101} = [7, 8, 4]$. sp_{100} and sp_{101} are sent to the parent master M_{10} using *Gather*. M_{10} reduces them using *Reduce* and their root problem $rp_{10} = [7, 8]$ is found. After that, the locally unexplored subproblems are deduced using *Deduce* and the couple of the root problem and the branched subproblems

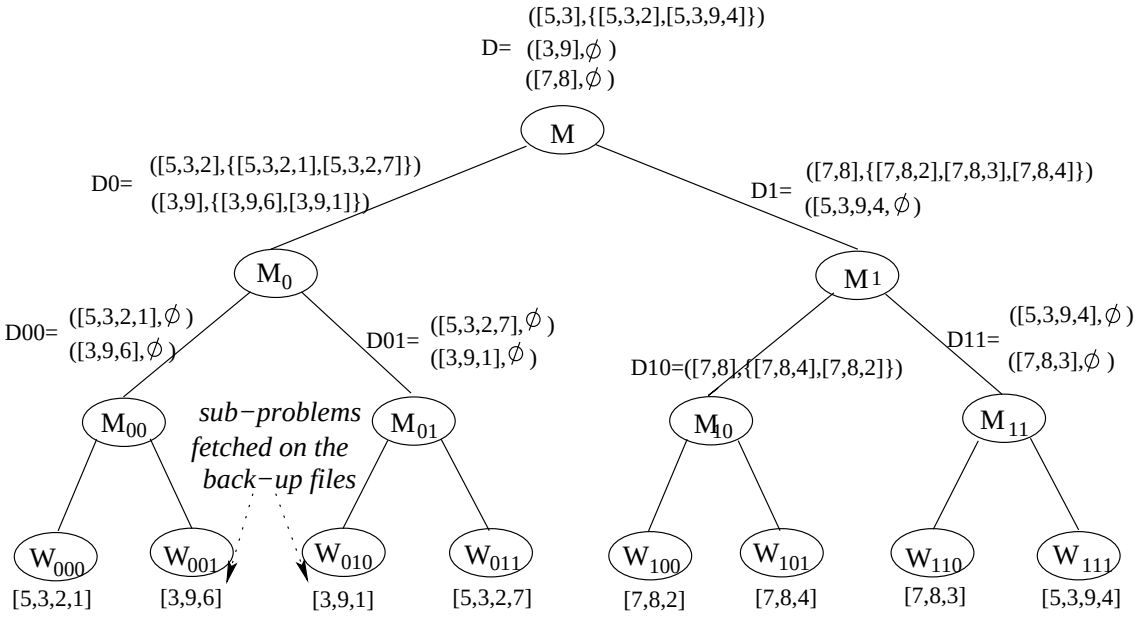

Figure 4.8: An example of subproblems recovery and reduction mechanism. Masters apply the three operators to deduce the global unexplored subproblems.

$D_{10} = ([7,8]\ ,\ \{[7,8,4],[7,8,2]\})$ is found.

The main drawback of this approach is the large number of generated backup files when the number of launched sub-B&Bs increases. That can degrade the overall performance of the approach. In fact, when many masters are launched on the same grid cluster, the multiple disk accesses can slow down the cluster. One can remedy to this drawback by uniformly launching sub-B&Bs on different grid clusters, or by limiting the number of masters that can perform checkpointing. Therefore, two types of masters can be distinguished: masters that perform checkpointing (active masters) and those that do not (passive masters). The active masters are locate in the upper levels of the hierarchy and the passive ones in the lower levels. We can decide dynamically if a master can wether or not participate in the checkpointing process.

4.6 Implementation of FT mechanisms

To take into account FT, new methods have been implemented by the different components of the AHMW framework (see Chapter 3). Therefore, they introduce new features to ensure fault detection and hierarchy maintenance.

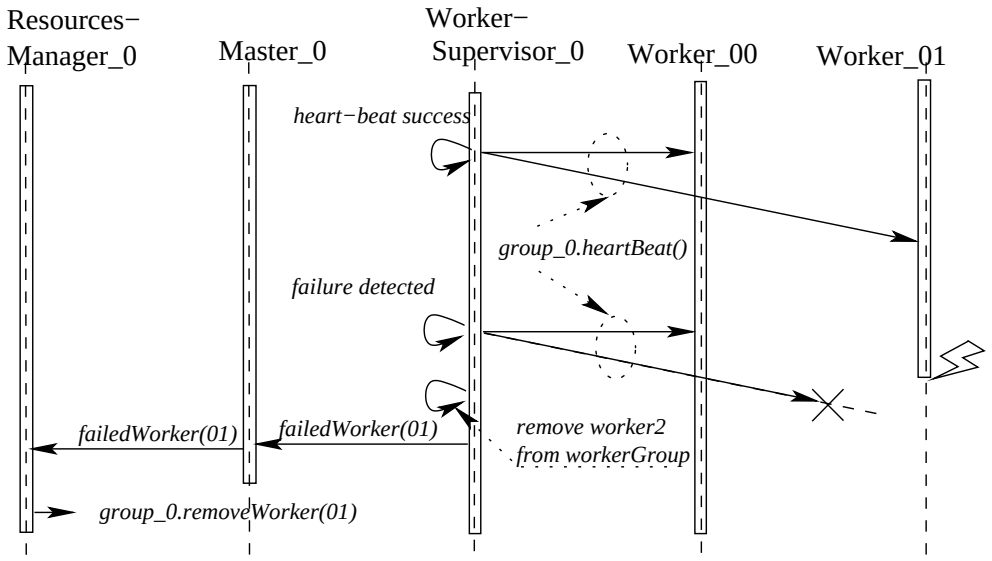

Figure 4.9: Fault detection sequence diagram.

4.6.1 Fault detection

The fault detection is performed by the *Worker-Supervisor* process (see the sequence diagram on Figure 4.9). It sends heart-beats to the group of workers it supervises every *heart-beat* period by calling the *heartBeat()* method on the ProActive typed group *group0* representing the children of the current master. The *group0* on its side, forwards the message to the whole workers. Each worker implements the *heartBeat()* method which is an empty method allowing the *Worker-Supervisor* to know whether or not they are alive using ProActive lower level mechanisms. It informs the master about the failed workers by calling the method *failedWorker()* and removes the failed worker from the group communication using *removeNeighbor()*.

4.6.2 Implementation of the hierarchy maintenance algorithms

4.6.2.1 Simple connection to ascendants (*SCA*)

The workers use the *heartBeat()* method implemented on the master to know whether it is alive or not (see the sequence diagram in Figure 4.10). When they detect the master's failure, they connect to the closest master by calling the *connectToAscendantsCSA()* method on the safe ascendant master. The considered master then informs its *Resources-Manager* through the method *newWorker()* and transfers the identity of the orphan worker. The *Resources-Manager* updates the old typed group communication and involves the orphan workers in the computation by calling the method *addNeighbor()* implemented at the level of the workers.

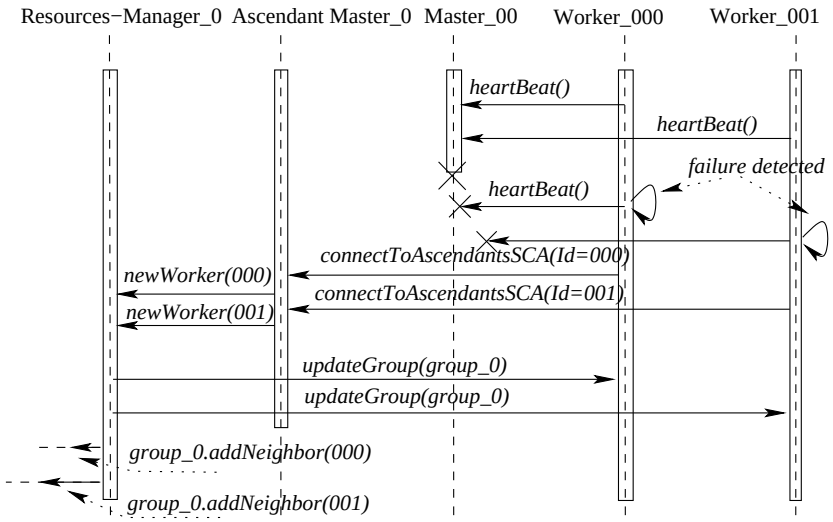

Figure 4.10: *SCA* sequence diagram.

4.6.2.2 Master Election (*ME*)

When a group of workers detect the failure of their master, they initiate the election by sending their identifiers to the workers involved in the bully algorithm using the method *bullyAlgorithm()*. After the algorithm converges, the selected worker changes its behavior and moves into master and creates its own *Resources-Manager*, which is also an Active Object, in order to manage its neighbors. This new master connects to the closest ascendant master according to the previous method (*CSA*).

4.6.2.3 Balanced Hierarchy (*BH*)

The orphan workers connect to the closest safe ascendant master using *connectToMasterBH()*. The considered master informs its *Resources-Manager* about this new worker through *newWorkerBH()*. The *Resources-Manager* then holds the workers if the group not yet full and handles it as a simple new worker through *newWorker()*, otherwise, it dispatches them among its children through *newWorkerBH()*. The *newWorkerBH()* is called among the children until a non full group is found.

4.7 Performance evaluation

FTH-B&B has been experimented on the Flow-Shop scheduling problem (*FSP*) considering the total completion time *(C_{Max})* as cost function. In all the reported results FTH-B&B is experimented at large scale. Indeed, between 1900 and 8900 FTH-B&B processes are deployed in the different experiments. In order to obtain deeper hierarchy, for some experiments multiple processes are launched on the same processor.

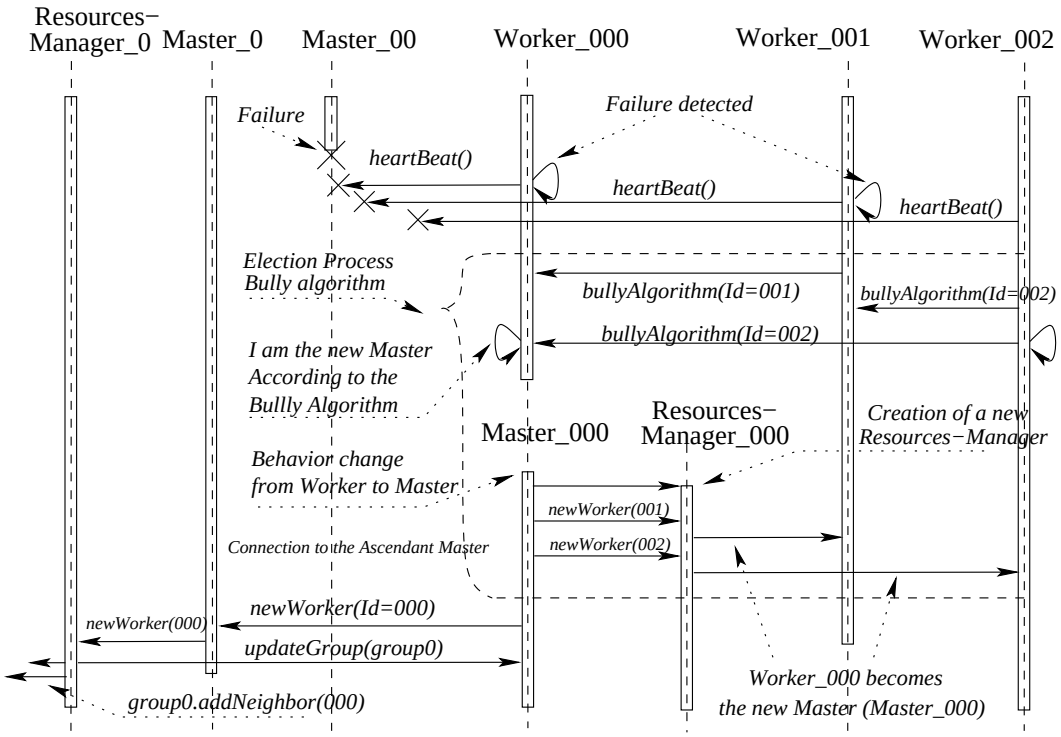

Figure 4.11: *ME* sequence diagram.

4.7.1 Fault Injection

The simplest and most obvious method to stress a grid-based application in terms of failures is to emulate a fault on a randomly chosen processor every fixed period of time. In the literature, several approaches have been proposed for fault injection in distributed systems. Hoarau *et al.* have summarized some approaches in [HT05]:

ORCHESTRA [DJM96] is a fault injection tool. It allows the user to test the reliability and the liveliness of distributed protocols. A fault injection layer is inserted between the tested protocol layer and the lower layers, and allows to alter and manipulate messages exchanged between the protocol participants. Messages can be delayed, lost, reordered, duplicated, modified and new messages can be spontaneously introduced into the tested system to bring it into a particular global state. This approach cannot be applied to our work and nowadays grid-based applications since the real challenge in fault tolerance is not the validity of the exchanged messages but at the level of the reliability of computing processes.

NFTAPE [SFB00] provides mechanisms for fault-injection, triggering injections, producing workloads, detecting errors, and logging results. Unlike other tools, *NFTAPE* separates these components so that the user can create his own fault injectors and injection triggers using the provided interfaces. *NFTAPE* introduces the notion of Lightweight Fault Injector (*LWFI*). *LWFIs* are simpler than traditional fault injectors, because they don't need to integrate triggers, logging mechanisms, and communication

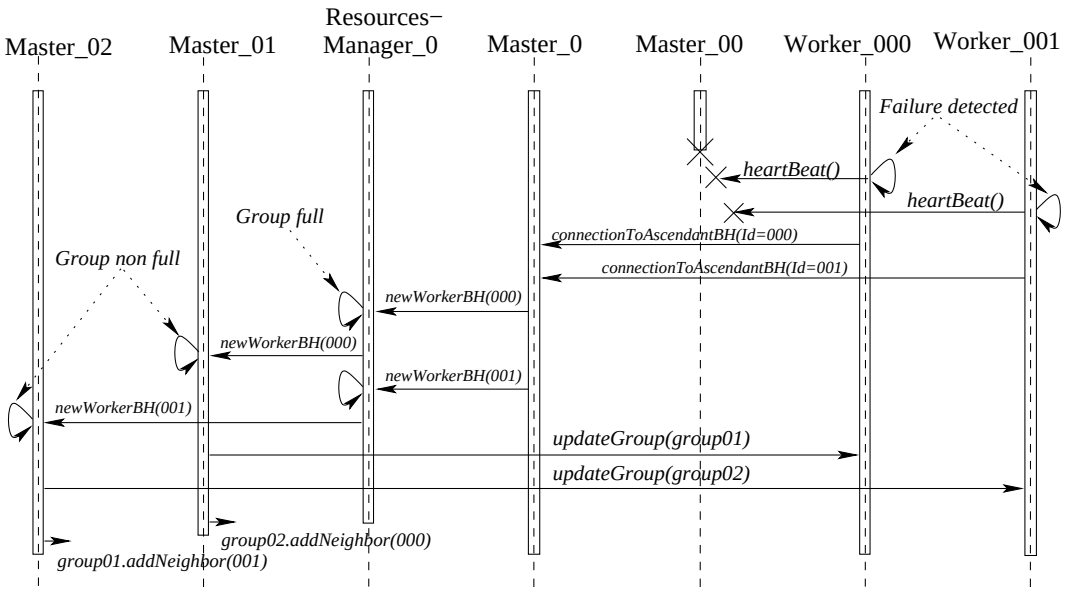

Figure 4.12: *BH* sequence diagram.

support. This way, *NFTAPE* can inject faults using any fault injection method and any fault model. Interfaces for the other components are also defined to facilitate portability to new systems.

LOKI [CLCS00] is a fault injector dedicated to distributed systems. It is based on a partial view of the global state of the distributed system. The faults are injected based on a global state of the system. An analysis a posteriori is executed at the end of the test to infer a global schedule from the various partial views and then verify if faults were correctly injected according to the planned scenario.

FAIL-FCI [HTV07, HT05, HTV05] (*FAult Injection Langage*) which is a language allowing to easily describe fault scenarios and (*Cluster Implementation*) which is a distributed fault injection tool using FAIL language. FAIL-FCI is an interesting tool for distributed fault injection, however, the library is written in C++ reducing than its portability. Moreover, even though the generated failures are qualitative, but they are still static and they cannot reflect the realism of the scenarios and they do not take into account the lifetime of the processes.

However, none of these approaches has considered the lifetime of the processors. To give more realism to our experiments in an unreliable environment, we consider that the lifetime distribution function of the processes follows an exponential distribution [XDP04]. The reliability of a process i is considered as the probability that it performs its function for a specified period of time. The reliability function $R_i(t)$ is the probability that the process i will be successfully operating without failure within the interval $[0, t]$, $R_i(t) = P(T > t), t \geq 0$, where T is a random variable representing the failure time. Therefore, the failure probability is given by: $F(t) = 1 - R(t) = P(T \leq t)$.

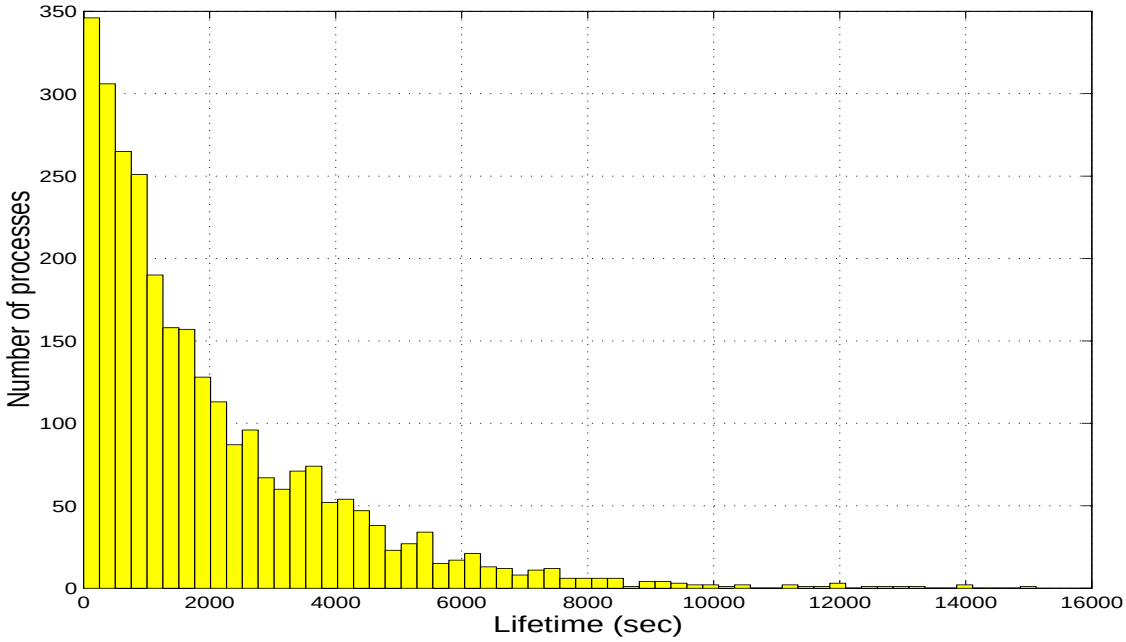

Figure 4.13: Lifetime distribution of the launched processes.

If the lifetime distribution function follows an exponential distribution with parameter λ, $F(t) = 1 - e^{-\lambda t}$. In our experiments we only consider the failures of node. Given constant failure rates of resources, one can obtain the conditional probability of a process success as $p(t) = e^{-\lambda t}$ [DLT07], where λ is the failure rate of nodes. In all the experiments we fixed $\lambda = 0,5$ in order to expose the processes to extreme failure situations and to evaluate the robustness of the application. The lifetime distribution of the processes involved in the experiments is presented in Figure 4.13. The height of the bars indicates the number of processes and their width represents intervals of lifetime duration. The whole bars show the number of processes that live a given time duration. According to the exponential distribution, the number of processes decreases as their lifetime increases.

4.7.2 Experimental Results

In the following, we experimentally evaluate the ability of FTH-B&B to deal with the fault tolerance issue in large scale unreliable environments. Indeed, we measure the efficiency of FTH-B&B which is the effective CPU time taken by the workers solving their tasks taking into account fault tolerance (the time of storing and updating subproblems). We also evaluate the benefit of the task recovery using the 3-phase mechanism on the efficiency of the algorithm in terms of the number of recovered and rescheduled subproblems and the gain in terms of execution time. Finally, we measure the impact of failures on the overall topology of FTH-B&B using the three rebuilding strategies *SCA*, *ME*, and *BH*.

Bench	Execution T_{Exec}	3-Phase $T_{3-Phase}$	Delay $T_M + T_C$	Efficiency
Ta021	46996,603	120,507 (0,25%)	367,501	98,97%
Ta022	30843,960	126,265 (0,40%)	401,372	98,31%
Ta023	47180,395	130,493 (0,27%)	381,944	98,92%
Ta024	57050,001	128,745 (0,22%)	384,204	99,10%
Ta025	60370,203	129,536 (0,21%)	421,354	99,09%
Ta026	50753,190	117,435 (0,23%)	423,435	98,94%
Ta027	54855,053	128,871 (0,23%)	362,219	99,11%
Average	50840,405	125,187 (0,25%)	385,206	98,92%

Table 4.1: Efficiency of FTH-B&B

4.7.2.1 Efficiency of FTH-B&B

In Table 4.1, we report experimental results obtained on Taillard instances defined by 20 jobs and 20 machines published in [TAI93]: Ta021 – Ta027. In this experiment, we evaluate the delay induced by the 3-phase communication mechanism on the performance of the algorithm. We calculate the ratio R between the effective execution time t_{Exec} and the idle time t_I recorded on the workers. The idle time includes the communication time t_C and additional time of internal management (local overhead) t_M, and the time masters take to perform the 3-phase communication mechanism $t_{3-Phase}$ which includes: time of branching, time of storing subproblems received from children and times of updating the subproblems explored by the grandchildren.

$$R = 100 \times \frac{t_{Exec}}{t_{Exec} + t_I}$$
$$t_I = t_{3-Phase} + t_C + t_M$$

The columns 1 to 5 designate, respectively, the name of the instance, the effective execution time, the 3-Phase communication time, the communication and local management time, and the parallel efficiency. Times are expressed in seconds. As shown in Table 4.1, the use of the 3-phase communication mechanism is not costly in terms of execution time. Indeed, it takes between 0,20% and 0,40% of the execution time which is insignificant compared to the total execution time on all the experimented instances. As shown in the last column, the workers spend on average $98,92\%$ of their time solving subproblems on most of the instances.

The task recovery mechanism adopted by FTH-B&B aims at minimizing the redundant work (rescheduled subproblems) when the processes handling them fail. Table 4.2 represents a comparison between two strategies of work management: with and without task recovery in presence of failures. We measure the number of rescheduled subproblems during 40 minutes of execution solving the FSP instances Ta021 - Ta027 described above using 1120 grid nodes (112 masters and 1008 workers). The hierarchy is rebuilt at each instant using *BH*. Table 4.2 shows respectively, the percentage of dead processes, the number of rescheduled subproblems (redundant work) using the 3-phase communication mechanism and without using it, and finally the speedup between the

Bench	(%) Dead processes	Redundant work		Speedup
		Task Recovery (TR)	Without TR	
Ta021	70,98%	27868835188	41478715872	01,48
Ta022	52,76%	2056690549	19667608092	09,56
Ta023	42,58%	5550936476	31977434544	05,76
Ta024	58,39%	4235795214	18402793860	04,34
Ta025	53,66%	7049773500	48082138104	07,07
Ta026	50,98%	5543751206	32421508416	05,84
Ta027	51,16%	1195024644	12529597080	10,48
Average	54,89%	7642972397	29222827995	6,36

Table 4.2: Redundant work with and without task recovery

two strategies.

We note that without using the 3-phase communication mechanism for task recovery, the masters reschedule more subproblems than when they use it. Therefore, more redundant work is done on all the tested instances, slowing down the algorithm. The speedup between the two strategies is then calculated and shows that it is on average equal to 6,36. That means that when using the task recovery mechanism, FTH-B&B is 6 times more efficient than another approach which does not adopt a task recovery mechanism in an unreliable environment.

4.7.2.2 Evaluation of the hierarchy maintenance strategies

As mentioned in Section 4.4, it is important to maintain the hierarchy balanced during the lifetime of the algorithm in order to obtain a valid execution and to avoid isolating some parts of the hierarchy when inner masters are subject to failures. In the following, we experiment this aspect according to the three proposed approaches: *SCA*, *ME*, and *BH*. During the experiments, we measured the degrees (number of children) of the different masters. The degrees inform us about both the shape of the hierarchy and whether the nodes are subject of bottlenecks or not. Indeed, when the average degrees are constant on all the masters, the hierarchy remains balanced. However, when the degrees differ and diverge from a master to another, the hierarchy is unbalanced. Moreover, if the degree of a master is high, it presents a risk to become a bottleneck. To obtain a deeper hierarchy, the initial size of a group is fixed to 10 and the number of launched processes on the same processor is duplicated to get a larger scale in terms of processes. For each strategy, the algorithm is executed 10 times during 60 minutes.

Figure 4.14 shows the degrees recorded on the four most loaded masters when using *SCA* to determine wether or not failures create new bottlenecks in the hierarchy. The curve represents the evolution of the degrees as a function of time. First, we note that the degrees of the masters increase over time (with the number of failed processes), the fact that makes them slow because of the huge number of new workers to manage. Second, the root master process rapidly becomes bottleneck passing from a degree equal to 10 at the beginning of the execution, to 1368 at the end. This is due to the fact that in

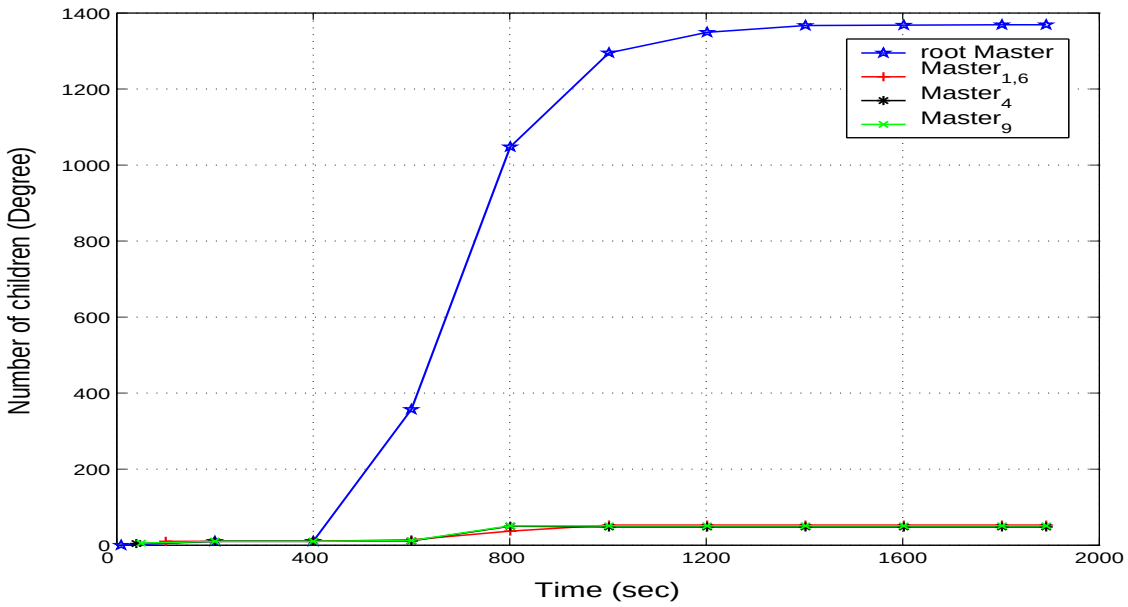

Figure 4.14: Average degree using *SCA*. The root master becomes rapidly a bottleneck over the time.

this strategy, when an inner master fails, all its children connect to its closest safe parent. Hence, the algorithm rapidly converges to a standard Master/Worker paradigm. Although *SCA* is easy to implement and to manage, it is not efficient in presence of failures since new bottlenecks are created and it rapidly converges to a standard Master/Worker.

Figure 4.15 shows the evolution of the degrees of four most loaded masters using *ME*. We note that the masters are less loaded compared to *SCA* and with a medium standard deviation because when a master is designated, it receives the highest identifier and then it will not be designated in the next election session. Nevertheless, the degrees have doubled and have reached their maximum value after killing 25% of processes. This is due to the fact that when an inner master fails, the newly designated master will manage the children of the failed master in addition to its own children.

As mentioned in Section 4.4.3, there is no risk of the creation of new bottlenecks and no risk of convergence to a standard Master/Worker paradigm when using *BH*. Therefore, we compute the average degree of masters on each level instead of the most loaded masters. The degrees are computed as follows:

$$D_{tl} = \frac{\sum_{i=1}^{|\Gamma_{tl}|} d_{it}}{|\Gamma_{tl}|}$$

D_{tl} is the average degree of the masters located at the level l of the hierarchy at the instant t, Γ_{tl} is the set of masters located at the level l at the instant t, and d_{it} the degree of the master m_i at the instant t.

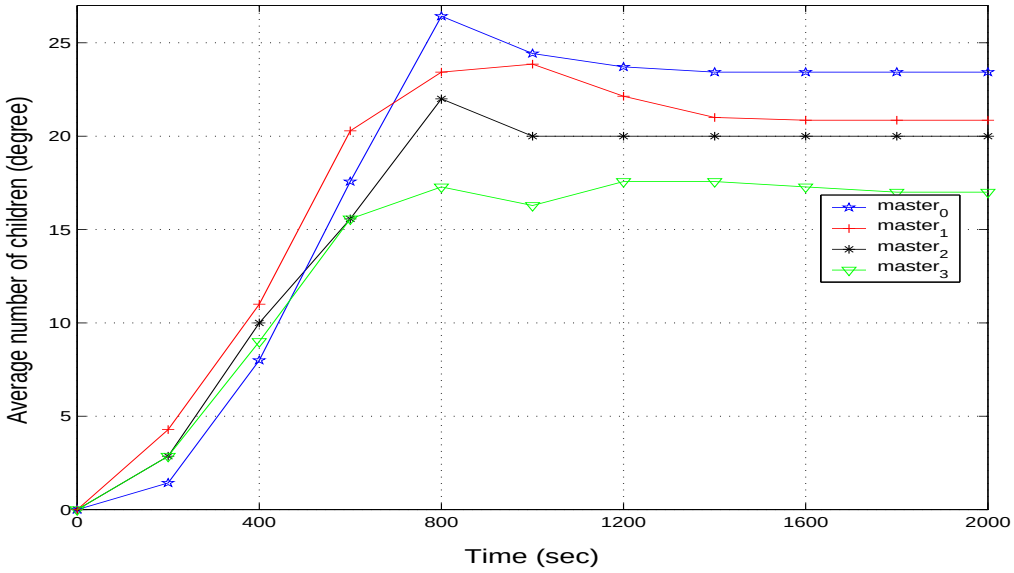

Figure 4.15: Average degree using *ME*. Masters are less loaded than when using *SCA*

Figure 4.16 shows the computed average degrees of the different masters sorted by their levels. First, we note that the masters resist well to failures and do not become bottlenecks during all the experiments. Second, the root master maintains the same degree during all the lifetime of the algorithm. Finally, the average degree decreases softly for the three first levels and more rapidly for the fourth one. That can be explained as follows: at the beginning, all sub-B&Bs are initially full because of the balanced construction of the hierarchy. After the first failures, no master can acquire orphan processes. Therefore, it uniformly dispatches them among its children, and so on until achieving the leaves of the hierarchy (workers). The adaptability of FTH-B&B allows the worker receiving an orphan worker, to change its behavior and becomes a master. This newly converted master will have only one child (the migrated worker), therefore its degree=1. Consequently, the overall degree of the masters at the down levels of the hierarchy is affected. After many failures, the new masters acquire more and more migrated orphan processes. Therefore, the size of sub-B&B groups converges to the maximum size of a group stabilizing the global degree.

4.8 Conclusion

Solving to optimality combinatorial optimization problems using parallel B&B algorithms requires a huge amount of computing resources. That can be achieved with their execution at large scale on computational grids. The scalability can be ensured using hierarchical Master/Worker-B&B overcoming the limits of the traditional Master/Worker paradigm. However, the resources offered by such grids are volatile and highly unreliable. Therefore, fault tolerance must be taken into account to ensure no loss of data during the execution of the algorithm which is intolerable for exact solving

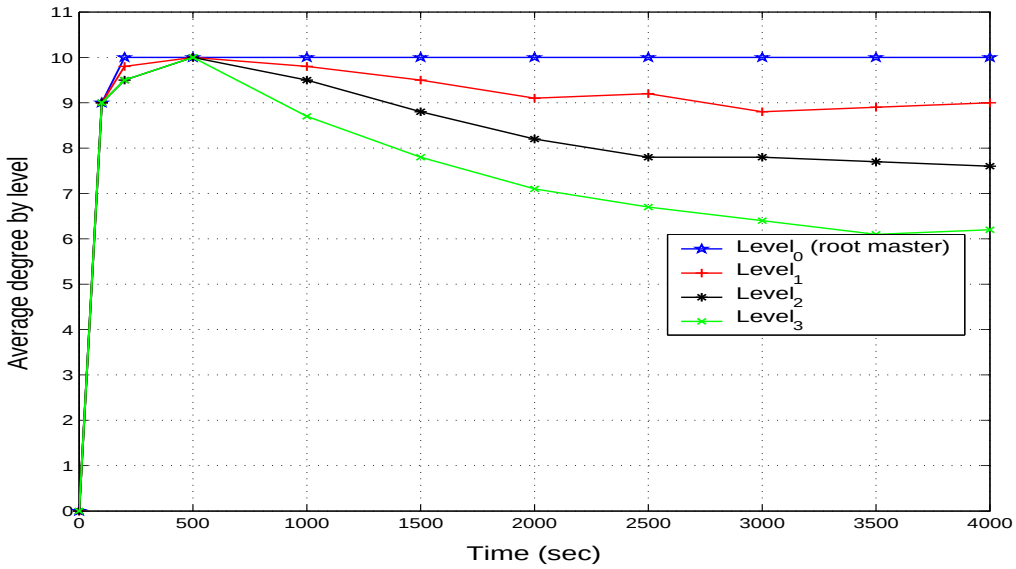

Figure 4.16: Average degree using *BH*. The masters resist well to failures and do not become bottlenecks.

of COPs.

In this chapter, we have proposed FTH-B&B (Fault Tolerant Hierarchical B&B algorithm) in order to deal with the fault tolerance issue of grids. The fault tolerance is introduced at the application level. Indeed, it is composed of several fault tolerant Master/Worker-based sub-B&Bs launched in parallel and organized into a hierarchy so that each sub-B&B is composed of a unique master managing several workers and performing locally a set of fault tolerance mechanisms. A 3-phase communication mechanism between a master, its children, and its grandchildren is proposed to distribute, store, and recover work units in case of failures. This mechanism allows one to gain in terms of execution time because the master does not reschedule the entire subproblems but only the unexplored parts. In addition, three strategies are proposed to maintain the hierarchy during the lifetime of the algorithm when it is faced to failures. Finally, a distributed checkpointing mechanism is proposed in which each master performs its checkpointing independently from others. Three operators are defined: *Gather*, *Reduce*, and *Deduce*, allowing the reconstitution of the unexplored subproblems.

We have implemented our approach on top of ProActive and we have experimented it on the Grid'5000 real nation-wide grid hardware infrastructure. To give more realism to our experiments in an unreliable environment, the failures are generated according to the exponential lifetime distribution function of the processes. The different experiments demonstrate the ability of FTH-B&B to ensure fault tolerance while maintaining high execution efficiency. Indeed, the approach enables to achieve an efficiency of 98,92%. Moreover, the task recovery allows to speedup the approach in average more than 6 times compared to the same approach but without using the 3-phase mechanism. Moreover, the experiments show that the hierarchy can be maintained safe and

balanced, thanks to the balanced hierarchy *BH* maintenance strategy which is more efficient than the master election *ME* and the simple connection to ascendants *SCA* strategies.

Conclusions and Perspectives

IN this thesis, we presented our contributions to grid-based Branch-and-Bound algorithms for exact resolution of combinatorial optimization problems. We particularly addressed some challenging issues related to scalability, fault tolerance, heterogeneity of resources and load balancing, and design and implementation of grid-aware applications.

In the first contribution, we proposed a P2P MW-based B&B framework (*P2P-B&B*) aiming at facilitating the development of grid-based B&Bs, hiding the complexity of the grid to the users, and developing a complete grid-based B&B dealing with scalability. In this framework, the scalability is achieved by reducing the task request frequency towards the master process and enabling direct communication between workers. The task request frequency is minimized by proposing a solution to handle coarse-grained tasks without losing performance. This is achieved by performing multiple executions of an atomic fine-grained task rather than single execution of a compact coarse-grained task without overloading the master. Direct communication allows the workers to share their upper bounds at real-time and to perform other collaboration tasks alleviating the master process. The reported large scale experiments showed the benefit of the direct communication between the workers in terms of speedup. However, P2P-B&B is shown limited in terms of deployment time which is relatively high when considering larger number of computing resources.

In the second contribution, we proposed a new HMW-based B&B (*H-B&B*) aiming at improving the scalability of the conventional MW-based B&B by eliminating the bottlenecks created at the level of the central master process. H-B&B is based on (*AHMW*) framework on its side based on the *P2P-B&B* framework. Unlike the literature approaches, H-B&B is fully dynamic as it is composed of several levels of masters, and evolves over time according to the dynamic acquisition of new computing nodes. Masters perform decentralized branching on subproblems using a new proposed exploration strategy and workers perform a complete exploration of the received subproblems. The different components of H-B&B handle tasks of different grain sizes according to their roles (master or worker) and to their position in the hierarchy. Accordingly, bottlenecks likely to be created at centralized points in the hierarchy can be controlled.

Different large scale experiments of H-B&B demonstrate its efficiency compared to existing single-level HMW-based B&B (1-H-B&B) and more significantly compared to the classical MW-based B&B in terms of scalability. In fact, it deploys nodes much faster than the two other approaches, thanks to the participation of all the launched processes in the parallel deployment. This makes the number of deployed nodes increasing exponentially in time remaining insensitive to the large scale number of resources. The adaptive feature of masters behavior demonstrates its benefits on the efficiency of this approach. Indeed, the contribution of masters in the exploration processes allows to gain in terms of computing power. The average CPU-load on masters is minimized and prevents from the creation of bottlenecks among the masters. In the approach, the masters are assisted in the management of work requests and in the decomposition to achieve acceptable grains. This allows them to distribute tasks much faster without blocking the workers requesting tasks. Therefore, the workers spend less than 1% of the total time waiting for tasks from their parents. From the masters' point of view, this time (1%) represents mainly the decomposition time to achieve medium and small task grains, which facilitates the work control and minimizes the waiting time of workers.

The last contribution concerns Fault Tolerance. We proposed a Fault Tolerant Hierarchical B&B (*FTH-B&B*). FTH-B&B, also based on *AHMW* and then *P2P-B&B*, is an application-level distributed FT mechanism. A fault recovery mechanism is introduced to avoid the loss of work units and to improve efficiency in terms of execution time. Moreover, our approach ensures to maintain a balanced and safe hierarchy during the lifetime of the algorithm in order to guarantee a valid functioning. Finally, an efficient restart of the application is ensured by a distributed checkpointing mechanism in case of failure. The different experiments demonstrate the ability of FTH-B&B to ensure FT while maintaining high execution efficiency. Indeed, the approach enables to achieve an efficiency of 98,92%. Moreover, the task recovery allows to speedup the approach in average 6,36 times compared to other approaches without task recovery. The experiments show that the hierarchy can be maintained safe and balanced.

The works we presented in this thesis can be extended further in several directions. First, improve the efficiency of H-B&B by the exploitation of the different grains spawned at the different levels of the hierarchy. Consequently, to develop a grid-aware adaptive load balancing mechanism, and to produce several forms of granularities according to the processor power. Second we intend to develop an enhanced version which takes into account a best effort mechanism that allows to exploit a much larger number of resources on the used computational grid. We also intend to run our application on real production grids such as EGI/EGEE [EGI][EGEE] that has a high rate of nodes failure. Finally, we project to extend our hierarchical algorithms and frameworks to actually challenging infrastructures such as multi-core architectures.

Publications

Journal Papers

A. Bendjoudi, N. Melab, and E-G. Talbi, "***An adaptive hierarchical master-worker (AHMW) framework for grids-Application to B&B algorithms***", Journal of Parallel and Distributed Computing (JPDC), 2012.

A. Bendjoudi, N. Melab, and E-G Talbi. "***P2P design and implementation of a parallel branch and bound algorithm for grids***". International Journal of Grid Utility and Computing, Vol. 1, No 2, issn 1741-847X, pages 159–168, 2009.

Book Chapter

A. Bendjoudi, S. Guerdah, M. Mansoura, N. Melab, and E-G. Talbi. "***P2P B&B and GA for the Flow-Shop Scheduling Problem***". Book chapter in Metaheuristics for Scheduling: Distributed Computing Environments, Studies in Computational Intelligence, Edited by F. Xhafa and A. Abraham, Springer-Verlag Berlin Heidelberg, ISBN 978-3-540-69260-7, pages 301-321, September 2008.

Conferences and Workshops

A. Bendjoudi, N.Melab, and E-G Talbi. "***H-B&B: A Hierarchical B&B for large scale environments***". The Fourth IEEE International Scalable Computing Challenge IEEE/ACM CCGRID/SCALE'2011, Newport Beach, USA, May 23-26, 2011.

A. Bendjoudi, N.Melab, and E-G Talbi. "***Fault-Tolerant Mechanism for Hierarchical Branch and Bound Algorithm***". In Proc. of IEEE IPDPS'2011, Woks. on Large-Scale Parallel Processing (LSPP), May 16-20, Anchorage (Alaska), 2011.

A. Bendjoudi, N.Melab, and E-G Talbi. "***A Parallel P2P Branch-and-Bound Algorithm for Computational Grids***". 7th International Workshop on Global and Peer-to-Peer Computing, International Symposium on Cluster Computing and the Grid

2007 IEEE/ACM CCGRID 2007. Rio de Janeiro - Brasil, May 14-17, 2007.

E-G Talbi, **A. Bendjoudi**, and N.Melab. "***Parallel Branch and Bound on P2P Systems***". International Conference on Complex, Intelligent and Software Intensive Systems, IEEE/CISIS-2007, Vienna, Austria, 10-12 April 2007.

Bibliography

[AND86] S. Ahuja, C. Nicholas, and G. David. Linda and Friends. *Computer*, 19(8):26–34, 1986.

[AFO06] K. Aida, Y. Futakata, and T. Osumi. Parallel Branch and Bound Algorithm with the Hierarchical Master-Worker Paradigm on the Grid(Grid). *IPSJ Trans. on High Performance Computing Systems*, 47(12):193–206, 2006.

[ANF03] K. Aida, W. Natsume, and Y. Futakata. Distributed Computing with Hierarchical Master-worker Paradigm for Parallel Branch and Bound Algorithm. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:156, 2003.

[ALL05] G. Allen *et al.* The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid . In *Proceedings of the IEEE,* Vol. 93, pages 534–550, March 2005.

[ACK+02] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[BBC+06] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, Composing, Deploying for the Grid. In *in Grid Computing: Software Environments and*. Springer Verlag, 2006.

[BBL02] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide area distributed computing. . In *Software-Practice & Experience, 32(15), 1437–1466, 2002* .

[BCMH02] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssiere. Interactive and Descriptor-based Deployment of Object-oriented Grid Applications. In*In Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing,* pages 93–102, IEEE Computer Society, Edinburgh, Scotland, July 2002.

[BCM03] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications (DOA*, pages 1226–1242. Springer-Verlag, 2003.

[BMT07] A. Bendjoudi, N. Melab, E-G. Talbi. A Parallel P2P Branch-and-Bound Algorithm for Computational Grids. *7th International Workshop on Global and Peer-to-Peer Computing*, IEEE/ACM International Symposium on Cluster Computing and the Grid 2007 IEEE/ACM CCGRID 2007: 749–754. Rio de Janeiro - Brasil / May 14-17, 2007.

[BGMM08] A. Bendjoudi, S. Guerdah, M. Mansoura, N. Melab, and E-G. Talbi. P2P B&B and GA for the Flow-Shop Scheduling Problem. Book chapter in Metaheuristics for Scheduling: Distributed Computing Environments, Studies in Computational Intelligence, Edited by F. Xhafa and A. Abraham, Springer-Verlag Berlin Heidelberg, ISBN 978-3-540-69260-7, pages 301-321, September 2008.

[BMT09] A. Bendjoudi, N. Melab, and E-G. Talbi. P2P design and implementation of a parallel branch and bound algorithm for grids. *International Journal of Grid and Utility Computing*, 1(2):159–168, 2009.

[BMT11a] A. Bendjoudi, N. Melab, E-G. Talbi. Fault-Tolerant Mechanism for Hierarchical Branch and Bound Algorithm. In *Proc. of IEEE IPDPS'2011, Woks. on Large-Scale Parallel Processing (LSPP)*: 1806–1814, May 16-20, Anchorage (Alaska), 2011.

[BMT11b] A. Bendjoudi, N. Melab, E-G. Talbi. H-B&B: A Hierarchical B&B for large scale environments. The Fourth IEEE International Scalable Computing Challenge IEEE/ACM CCGRID/SCALE'2011, Newport Beach, USA, May 23-26, 2011.

[BMT12] A. Bendjoudi, N. Melab, and E-G. Talbi. An adaptive hierarchical master-worker (AHMW) framework for grids – Application to B&B algorithms. *J. Parallel Distrib. Comput.* 72(2): 120–131, 2012.

[BDLP08] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical Master-Worker Skeletons. In *LNCS*, Vol. 4902, pages 248–264. Springer-Verlag Berlin Heidelburg, 2008.

[BOI] BOINC: Berkeley Open Infrastructure for Network Computing, http://boinc.berkeley.edu/.

[BDS03] L. Bote-Lorenzo, A. Dimitriadis, and E. G. Sanchez. Grid Characteristics and Uses: A Grid Definition. In *European Across Grids Conference, LNCS 2970, Lecture Notes in Computer Science,* pages 291–298, 2003.

[BCG00] B. Bourbeau, T. G. Crainic, and B. Gendron. Branch and Bound Parallelization strategies applied to a depot location and container fleet management problem. In *Parallel Computing,* Vol. 26 (2000) 27–46.

[CAR93] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[CDCL06] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. ProActive: an integrated platform for programming and running applications on Grids and P2P systems. *In Computational Methos in Science and Technology 12(1)*, pages 69–77, 2006.

[CDHQ03] D. Caromel, C. Delb, L. Henrio, and R. Quilici. Asynchronous and automatic continuations of results between communicating objects. In *French patent FR03 13876 - US Patent Pending* .

[CLCS00] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. LOKI: A state-driven fault injector for distributed systems. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, June 2000.

[CON] Condor, http://www.cs.wisc.edu/condor/.

[COS07] A. Di-Costanzo. Branch and Bound with Peer to Peer for Large Scale Grids PhD thesis, Sophia Antipolis, France, 2007.

[CBCM07] A. di Costanzo, L. Baduel, D. Caromel, and S. Matsuoka. Grid'BnB: A Parallel Branch and Bound Framework for Grids. In *Procedings of the 13th International Conference on High Performance Computing*, Goa, India, December 2007.

[CCR06] T.G. Crainic, B.L. Cun, and C. Roucairol. Parallel Combinatorial Optimization. In *chapter Parallel Branch-and-Bound Algorithms,* pp 1–28. Wiley, 2006.

[DLT07] Y.S. Dai, G. Levitin, and K.S. Trivedi. Performance and reliability of tree-structured grid services considering data dependence and failure correlation. *IEEE T. Computers*, v. 56, pp. 925–936, 2007.

[DVC+09] Z. Dai, F. Viale, X. Chi, D. Caromel, and Z. Lu. A Task-Based Fault-Tolerance Mechanism to Hierarchical Master/Worker with Divisible Tasks. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:672–677, 2009.

[DJM96] S. Dawson, F. Jahanian, and T. Mitton. Orchestra: A fault injection environment for distributed systems. In *26th International Symposium on Fault-Tolerant Computing (FTCS),* pages 404–414, Sendai, Japan, June 1996.

[DDM11] M. Djamai, B. Derbel, and N. Melab. Distributed B&B: A Pure Peer-to-Peer Approach. In *Proc. of 25th IEEE LSPP/IPDPS*, Anchorage, (Alaska) USA, Mai 16th-20th 2011.

[DAK65] RJ. Dakin. A tree-search algorithm for mixed integer programming problems. In *The Computer Journal,* 8(3):250, 1965.

[DUGS06] L.M.A. Drummond, E. Uchoa, A.D. Gonçalves, J. M. N. Silva, M. C. P. Santos, and M. C. S. de Castro. A grid-enabled distributed branch-and-bound algorithm with application on the steiner problem in graphs. *Parallel Comput.*, 32:629–642, October 2006.

[EPH00] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An Object-Oriented Framework for Parallel Branch and Bound. Technical report, Rutgers University, Piscataway, NJ, 2000.

[EGEE] EGEE, http://www.eu-egee.org.

[EGI] EGI, http://www.egi.eu.

[EAWJ96] M. Elnozahy, L. Alvisi, Y. Wang, D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, 1996.

[FM87] R. Finkel and Udi Manber. Dib—a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9(2):235–256, 1987.

[FOS02] I. Foster. What is the Grid ? A Three Point Checklist. Grid Today, 1(6), July 22 2002.

[FOS05] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing,* in LNCS 3779. Springer-Verlag, 2005.

[FK98] I. Foster, and C. Kesselman. The grid: blueprint for a new computing infrastructure. In *Morgan Kaufmann Publishers Inc* SanFrancisco, CA, USA, 1998.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling Scalable Virtual Organizations. In *Int. J. High Perform. Comput. Appl.,* 15(3) 200–222, 2001.

[GBF+02] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Anan-Thakrishman, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. In *Cluster Computing,* 5(3):325–336, 2002.

[GAR82] H. Garcia-Molina. Elections in a Distributed Computing System. In *IEEE Transactions on Computers*, Vol. C-31, No. 1, 48–59, 1982.

[GAR79] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Commpleteness. W. H. Freeman & Co., ISBN 0-7167-1045-5, p. 208–209, New York, NY, 1979.

[GC94] B. Gendron and T.G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(06):1042–1066, 1994.

[GSDB09] M. Ghasemi-Gol, M. Sabzekar, H. Deldari, and A-H. Bahmani. A Linda-based Hierarchical Master-Worker Model. *International Journal of Computer Theory and Engineering*, 1(5):1793–8201, December, 2009.

[GJK05] T. Goodale *et al.* SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid. http://www.cs.vu.nl/kielmann/papers/saga-sc05.pdf, 2005.

[GKLY00] J. Goux, S. Kulkami, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. *IEEE Symposium and High Performance Distributed Computing (HPDC9)*, 9:43, August 2000.

[GLY00] J-P. Goux, J. Linderoth, and M. Yoder. Metacomputing and the Master-Worker Paradigm. In *Preprint MCS/ANL-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne*, 2000.

[GRIDa] Grid'5000, https://www.grid5000.fr.

[GLOB] Globus, http://www.globus.org.

[GRIDb] GridBus, http://www.gridbus.org.

[GRI02] A.S. Grimshaw. What is a Grid ? Grid Today, 1(26), 2002.

[HT05] W. Hoarau, S. Tixeuil. A language-driven tool for fault injection in distributed applications, In *Proceedings of the IEEE/ACM Workshop GRID*, November 2005.

[HTV05] W. Hoarau, S. Tixeuil, and F. Vauchelles. Easy Fault Injection and Stress Testing with FAIL-FCI. Technical Report 1421, Laboratoire de Recherche en Informatique, Universit Paris Sud, October 2005.

[HTV07] W. Hoarau, S. Tixeuil, and F. Vauchelles. FAIL-FCI: Versatile fault injection. Future Generation Computer Systems, Vol. 23, 913–919, 2007.

[IAM00] A. Iamnitchi. A problem-specific fault-tolerance mechanism for asynchronous, distributed systems. In *Proceedings of the International Conference on Parallel Processing 2000*, pages 4–14, 2000.

[JOH54] S.M. Johnson. Optimal two and three-stage production schedules with setup times included. Naval Research Logistis Quarterly,1:61–68. 1954.

[KBM02] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing . In *Software-Practice & Experience,* 32(2) 135–164, 2002.

[LD60] AH. Land, and AG. Doig. An Automatic Method of Solving Discrete Programming Problems. In *Econometrica,* 28(3):497–520, 1960.

[LFGL01] G. V. Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. In *Concurrency and Computation: Practice and Experience,* 13:643–662, 2001.

[LHPB04] E. Laure, F. Hemmer, F. Prelz, S. Beco, S. Fisher, M. Livny, L. Guy, M. Barroso, P. Buncic, P. Kunszt. Middleware for the next generation Grid infrastructure. In *Proceedings of CHEP,* Interlaken, Switzerland, 2004.

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

[MEL05] N. Melab. Contribution a la resolution de problemes d'optimisation combinatoire sur grilles de calcul. Habilitation a Diriger des Recherches de l'Université Lille1, 2005.

[MEZ07] M. Mezmaz. Une approche efficace pour le passage sur grilles de calcul de methodes d'optimisation combinatoire PhD Thesis, Université Lille1, 2007.

[MMT07a] M. Mezmaz, N. Melab, and E-G. Talbi. A Grid-based Parallel Approach of the Multi-Objective Branch and Bound. In *Fifteen Euromicro Conference on Parallel, Distributed and Network-based Processing*, IEEE Computer Society Press. Naples, Italy, February. 7-9 2007.

[MMT07b] M. Mezmaz, N. Melab, and E-G. Talbi. A Grid-enabled Branch and Bound Algorithm for Solving Challenging Combinatorial Optimization Problems. In *Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symposium*, Long Beach, California, March 26th - 30th 2007.

[NORD] NorduGrid, http://www.nordugrid.org.

[OAR] OAR, http://oar.imag.fr/

[OGCE] Open grid computing environment (ogce). http://www.ogce.org.

[PS98] C.H. Papadimitriou, and K. Steiglitz. Combinatorial Optimization: Algorithms and Complexity. In *Prentice-Hall, Inc.* ISBN: 0-13-152462-3, 1982.

[PROA] ProActive, http://proactive.objectweb.org.

[PL96] J. Pruyne and M. Livny. Interfacing Condor and PVM to haress the cycles of workstation clusters. *Journal on Future Generation of Computer Systems*, (12):56–61, 1996.

[RLS03] T. Ralphs, L. Ladanyi, and M. Saltzman. Parallel Branch, Cut, and Price for Large-Scale Discrete Optmization. In *Mathematical Programming* 98 (2003), 253–280.

[RENA] Renater, http://www.renater.fr/.

[SNS+97] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *HPCN Europe,* pages 491–502, 1997.

[SNMD02] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *3rd International Workshop on Grid Computing,* November, 2002.

[SFB00] D.T. Stott *et al.* Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium,* pages 91–100, March 2000.

[TAI93] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.

[TNS+03] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. In *Journal of Grid Computing,* 1(1) 41–51, 2003.

[TTL05] D. Thain, T. Tannenbaum, and M. Livny. Distribute computing in practice: the Condor experience. In *Concurrency-Practice and Experience,* 17(2-4) 323–356, 2005.

[TRI89] H.W.J.M. Trienekens. Parallel Branch and Bound on an MIMD System. Report 8640/A, Econometric Institute, Erasmus University, Rotterdam, Netherlands, 1986.

[TB92] H.W.J.M. Trienekens and A. Bruin. Towards a Taxonomy of Parallel Branch and Bound Algorithms. In *Erasmus University,* 1992.

[UNIC] Unicore Forum. http://www.unicore.org.

[WISD] WISDOM: Initiative for grid-enabled drug discovery against neglected and emergent diseases. http://wisdom.eu-egee.fr/.

[XDP04] M. Xie, Y.S. Dai, and K.L. Poh. Computing Systems Reliability: Models and Analysis. *Kluwer Academic Publishers: NewYork, NY, USA*, 2004.

[XTRE] XtremWeb, http://www.xtremweb.net.

[XRL05] Y. Xu, T. K. Ralphs, L. Lada'nyi, and M. J. Saltzman. Alps: A framework for implementing parallel search algorithms. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 319–334, 2005.