UNIVERSITÉ DE LILLE 1

ÉCOLE DOCTORALE SPI

Sciences Pour l'Ingénieur

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université Lille 1

Spécialité : INFORMATIQUE

Soutenue par

Mathieu DJAMAÏ

Algorithmes Branch-and-Bound Pair-à-Pair pour grilles de calcul

Directeurs de thèse: Pr. Nouredine MELAB
et Dr. Bilel DERBEL

préparée à INRIA Lille Nord-Europe, Équipe DOLPHIN

soutenue le 11 Mars 2013

Numéro d'ordre : 41079 — Année : 2013

Jury :

Rapporteurs :	Daniel TUYTTENS	-	Professeur, Université de Mons, Belgique
	Didier EL BAZ	-	Chargé de Recherche HDR, CNRS/LAAS
Examinateur :	Mohamed MOSBAH	-	Professeur, LaBRI, Université Bordeaux-1
Directeurs :	Nouredine MELAB	-	Professeur, Université Lille 1, Lille
	Bilel DERBEL	-	Maître de Conférences, Université Lille 1, Lille
Invité :	Mohand MEZMAZ	-	Université de Mons, Belgique

UNIVERSITY OF LILLE 1

DOCTORAL SCHOOL SPI

Engineering Sciences

PHD THESIS

to obtain the title of

Ph.D. of Science

of the University Lille 1

Specialty : Computer Science

Defended by

Mathieu Djamaï

Peer-to-Peer Branch-and-Bound Algorithms for Computational Grids

Thesis Advisors: Pr. Nouredine Melab
and Dr. Bilel Derbel

prepared at INRIA Lille Nord-Europe, Dolphin Team

defended on March 11^{th}, 2013

Order Number : 41079 — Year : 2013

Jury :

Reviewers :	Daniel Tuyttens	-	Professor, Université de Mons, Belgique
	Didier El Baz	-	Research Director, HDR, CNRS/LAAS
Examinator :	Mohamed Mosbah	-	Professeur, LaBRI, Université Bordeaux-1
Advisors :	Nouredine Melab	-	Professor, Université Lille 1, Lille
	Bilel Derbel	-	Assistant Professor, Université Lille 1, Lille
Invited :	Mohand Mezmaz	-	Research Scientist, Université de Mons, Belgique

Contents

List of Figures

Acknowledgments

I will seize the opportunity to thank warmly both my thesis advisors, Pr. Nouredine Melab and Dr. Bilel Derbel for their continuous support during the preparation of my thesis. I wish to thank Nouredine for being deeply involved in my work, for the numerous meetings with Bilel discussing my work, the articles and the presentations i have made during the thesis. I also thank especially Bilel for his great involvement during these three years, especially when writing and reviewing articles, reviewing and improving the manuscript, and at every step of the scientific work, when thinking about the challenges to face, desiging the solutions, proving the correctness of these solutions, defining the appropriate experiments.

I am also pleased to thank Pr. Daniel Tuyttens, Dr. Didier El Baz, Pr. Mohamed Mesbah as well as my advisors and Dr. Mohand Mezmaz for honouring me by their presence as examination jury for my thesis defense.

I also thank warmly my mother for supporting me during these three years (during my entire student life, in fact :)), my father for the many discussions we had on his experience of research in mathematics.

Last but not least, I shall not forget my dearest colleagues and friends (in and out of the Dolphin team) I have known during the preparation of my thesis, some of whom I had exchanged so many useful tips and valuable ideas : Mostepha Redouane Khouadjia, Ali Khanafer, Thé Van Luong, Moustapha Diaby, Nadia Dahmani, Imen Chakroun, Marie-Éléonore Marmion, Trong Tuan Vu, Pamela Wattebled, Russel Nzekwa, Jean Decoster, Julie Hamon, Yacine Kessaci, Ahcène Bendjoudi, Khedidja Seridi, Martin Bue, Karima Boufaras, Ali Asim.

Chapter 1
Introduction

The PhD Thesis, presented in this document, deals with Large-Scale Combinatorial Optimization on Computational Grids. It has been completed within the DOLPHIN [1] research group from CNRS/LIFL , Inria Lille-Nord Europe and Université Lille 1. The presented works are part of the Inria HEMERA large-scale initiative, which involves several research teams using the Grid'5000 French experimental grid infrastructure.

Combinatorial Opimization Problems (COPs) are often NP-hard. Depending on the expected quality of the solutions, resolution methods for these problems fall into two categories: *near-optimal* methods, also referred to as heuristics, and *exact methods*. There also exists hybrid methods which combine complementary elements from multiple methods. Among heuristics, one can find *metaheuristics* which can be applied to a greater variety of problems. These methods are often used to tackle large problems and can produce good solutions within a short amount of time but these solutions are rarely optimal. On the contrary, exact methods allow to find optimal solutions along with the proof of optimality. However, their cost in terms of computation time is huge which makes them unpractical.

Large-scale parallelism, based on computational grids, appears to be a useful tool to face the processing cost issue of exact optimization methods. A grid can be seen as a set of resources located on multiple geographical sites, connected through a large-scale network and characterized by their volatility and heterogeneousness. Exploiting the potential computing power of such an environment requires the design of parallel algorithms which deal with the issues of volatility (for Fault-Tolerance purposes), heretogeneity (for Load Balancing purposes) and efficient communication management (for Scalability purposes).

During a resolution process, the irregularity of the search tree explored by exact methods as well as the heterogeneity of computational grids and their volatility, induce a huge amount of load balancing and checkpointing operations [Mezmaz 2007a]. All these operations imply high amounts of communications for storing and transferring work units between computing entities. Besides,

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

the cost of such communications may be even much more important due to the scale of a grid and its communication delays. It becomes crucial to design appropriate mechanisms to deal efficiently with these issues. Most of existing works propose approaches based on the Master-Slave paradigm [Mezmaz 2007b, Drummond 2006, Mans 1995, V.K. Janakiram 1988], where, however, the roles of the master and the slave entities can vary greatly from one approach to another. In a general way, a slave entity is in charge of applying a given optimization method on a subproblem for the problem being solved. It computes a partial result which is sent to the master entity. The latter manages global information and is in charge of balancing workload among slave entities as well as detecting the termination of the computation (the moment when all the solutions have been explored).

However, every approach based on the Master-Slave paradigm faces a major limitation, related to scalability. Indeed, harnessing a huge amount of computational power to perform a task requires an intensive process of synchronization between the computing entities. In the case of an exact method, it consists in sharing the best solution found among all entities, balance the load over the network, handle checkpointing operations for fault-tolerance purposes. These tasks generally induce a communication bottleneck on the master entity, degrading significantly the performance of the approach.

To overcome this limitation, two classes of approaches have been explored in the literature. The first one consists in introducing additional levels of centralization. Such algorithms are better known as *hierarchical* Master-Slave algorithms [Bendjoudi 2009]. Additional "*sub-master*" processes are in charge of overseeing only a part of the whole network, including communication operations. These processes send synthesized information to the global master. This kind of approach reduces the communication load upon the global Master but introduces delays in processing slaves' requests as communications have to cross multiple levels of hierarchy.

The second class includes Peer-to-Peer (P2P) approaches. The Peer-to-Peer paradigm is usually used for data sharing. In this thesis, we believe that it can be used for computation purposes, as in [Nguyen 2012].In [Di Constanzo 2007], the main idea is to provide a generic platform, so that communications induced by a parallel algorithm can be handled into a P2P fashion, using routing mechanisms. In their experiments, they validate the platform by using a Master-Slave based Branch-and-Bound algorithm, named *Grid'BnB*. Computing entities act as peers, each one having a set of neighbors. At the application level, one of these peers acts as the master entity and other peers act as slaves. Thus, whenever a slave peer needs to communicate with the master, its message is relayed through multiple peers before being delivered to the master. The main drawback of such an approach is that the communication load on the master entity is momentarily reduced,

but it is distributed through time. Indeed, for a given number of slaves, this architecture simply allows one to delay the requests from the slaves located far away from the master in the Peer-to-Peer overlay network so that the master can process incoming requests within a more reasonable amount of time. The main issue inherent to a Master-Slave architecture, that is the communication bottleneck preventing the approach from being scalable, is not overcome. Another Peer-to-Peer approach has been designed by Bendjoudi *et al.* [Bendjoudi 2009]. The application is based on a Master-Slave architecture but direct communications between the computational entities are allowed for sharing the best solution and other collaboration tasks to reduce the load on the master. The main limitation is that this kind of optimization can be performed only in specific situations by defining a sort of P2P communication layer beneath the application level. The main operation of the algorithm still relies heavily on the master entity and thus, remains an obstacle to scalability. Moreover, to the best of our knowledge, none of these works provide a formal proof of the correctness of the designed approaches.

In this thesis, we propose a new approach to overcome this limitation of scalability. The approach is completely decentralized, that is computational entities operate in a fully Peer-to-Peer fashion. Designing adequate mechanisms under such a decentralized architecture is very challenging. Indeed, there is no entity in the network which has a global view of the network. In the case of the Branch-and-Bound algorithm, no entity can determine immediately what the best solution found so far is nor if the termination of the calculation has occurred. Whereas those two tasks can be handled easily in a centralized[2] environment, they become major challenges in a fully decentralized one. Thus, to face these challenges, our approach provides the following mechanisms. Each peer is in charge of handling a local work pool and sharing it with other peers. Global information, like the best solution found so far by the optimization method, is broadcast over the network by the peers. Termination detection is handled in an innovative and decentralized way. Each peer can detect locally the presence or absence of a work unit somewhere in the network only by communicating with its neighbors and using some of the network overlay's properties. Performing all the required synchronization operations in a fully distributed manner allows to harness resources at very high scales by reducing significantly the communication load upon the computational entities. In addition, we propose a formal proof of the correctness of our approach, that is, the termination of the computation is detected in an appropriate way, the exploration process is achieved in a finite amount of time and no deadlock situations can occur during communication operations.

Moreover, we provide an extensive experimental study on the influence of the network overlay's topology on the scalability of our approach. Indeed, to the best of our knowledge, almost none of the existing works take into account the

[2] Whether Master-Slave or hierarchical architectures.

way in which computing entities communicate with each other. Most works using decentralized architectures simply conduct experiments using a predefined P2P overlay and present the subsequent results as a proof of concept of the involved distributed algorithm. We consider various topologies among the most commonly used in the literature. Experiments show that our approach can operate independently from the used topology and that best results are achieved using small-world graphs, which offer the best compromise for distributing communications over the network.

Finally, we extend our approach to *dynamic* environments, that is where resources are volatile. We design additional mechanisms for checkpointing operations, reassigning work units from failed peers, and handling efficiently the joining or the departure of peers from the network. Experiments demonstrate the robustness of the approach when facing real-case as well as "*failure-intensive*" scenarios.

This thesis is organized into 7 chapters. Chapter 2 introduces some key concepts dealing with Combinatorial Optimization as well as Grid Computing. It also provides an overview of existing works among Master-Slave based approaches and P2P-based approaches. Chapter 3 describes our fully Peer-to-Peer approach into a *static* environment, that is where all the resources are considered to be reliable, as well as a complexity study in terms of time and messages. Chapter 4 details the formal proof of the correctness of our approach. Chapter 5 provides an extensive experimental study of our approach and the impact of the network topology on its performances. Chapter 6 describes our Fault-Tolerant approach and the different mechanisms designed to handle resources volatility. We also demonstrate the robustness of our approach by simulating various failure scenarios. In Chapter 7, we summarize the major achievements of our contributions in this thesis and give some perspectives. Appendix A provides an additional study of Peer-to-Peer protocols.

Chapter 2

Parallel Distributed Branch-and-Bound Algorithms

Contents

2.1 Introduction

Many real-world problems can be modeled as Combinatorial Optimization Problems (COPs). Solving such problems in an exact manner may be computing-intensive and time-consuming and thus requires a huge amount of computational resources to be achieved. More and more computational resources can be made available from computational grids, clusters, personal computers connected through the internet, etc. Solving difficult and large scale combinatorial problems on these environments requires the design of efficient and scalable distributed algorithms that can be effectively deployed over large scale distributed environments. In this chapter, we introduce some key concepts at the crossroads of Combinatorial Optimization and Distributed Computing related to the issues underlying the design and the deployment of such algorithms. More precisely, Section 2.2 introduces the main concepts related to Combinatorial Optimization and particularly, the Branch-and-Bound algorithm. Section 2.3 introduces some concepts related to Grid Computing and presents an overview of Grid environments and some state-of-the-art works related to the parallelization of the B&B algorithm. Section 2.4 introduces more technical elements to be used in our contributions. More precisely, we introduce some elements from an existing approach and problem-encoding procedure which our contributions are based on.

2.2 Combinatorial Optimization and the Branch-and-Bound Algorithm

2.2.1 Preliminaries

Generally speaking, in *Combinatorial Optimization*, also referred to as *discrete optimization*, the goal is to find one or several configuration(s), among a finite but large set of possible configurations, optimizing a given objective function. A configuration is termed as a *solution* of the problem being considered and belongs to a mathematical space called *solution space*. Selecting a solution depends on its *cost*, which is defined by an *objective function*. Solving a combinatorial optimization problem then turns out to be the process of finding a solution with an 'accurate' value of its objective function. Depending of the expected accuracy, resolution methods can be classified into two categories (See Figure 2.1):

- **Exact methods:** Exact methods allow to obtain a set of solutions which are optimal with respect to the objective function. In other words, the output solutions are *guaranteed* to maximize (or minimize) the objective function. In this class of methods, one can cite Branch-and-Bound algorithms, constraint programming, dynamic programming, etc. Although an exact method allows to solve a given problem to optimality, it however requires to enumerate (in an implicit or explicit way) all the possible solutions for the problem being solved.

- **Approximate methods:** Approximate algorithms, also referred to as *heuristics*, allow one to obtain "good" solutions without strong guarantees on the optimality of computed solutions. They can be applied to problems for which the size of the solution space is subsequent. Some of these heuristics can be designed to solve a specific type of problems while others are more generic. The latter are called *metaheuristics*. These methods can operate in two different ways. They can be *single-solution based*, meaning that one solution is initially considered and then improved iteratively along with the solution space exploration process (Tabu search, simulated annealing, steep descent methods, hill climbing, etc). The other type is called *population-based*, meaning that an initial set of solutions is considered and then, all these solutions are improved simultaneously or independently, during the exploration process (Particle Swarm optimization, Ant Colonies, Evolutionary Algorithms, etc).

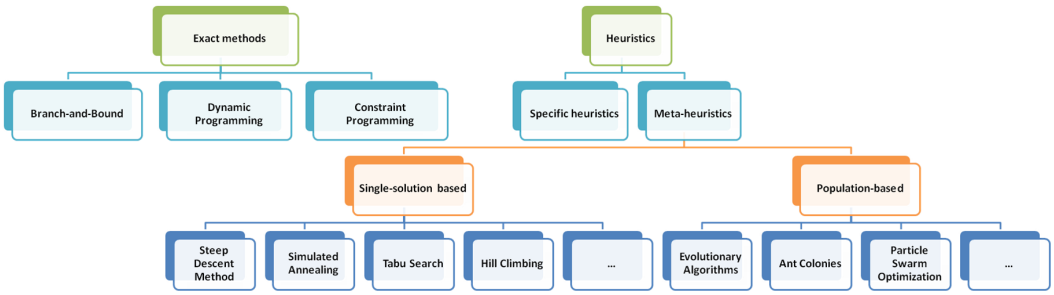

Figure 2.1: Taxonomy of resolution methods in combinatorial optimization.

The goal of this thesis is to design efficient *exact* algorithms for combinatorial optimization problems. In fact, although exact methods allow one to find a solution with optimality guarantees, they are computing intensive and very time consuming when tackling hard and large scale problem instances. In this context, parallel computing is more than just a possible alternative, it is in fact crucial to not say mandatory to solve such instances. The contribution of this thesis lies precisely in the design of parallel Branch-and-Bound algorithm in large scale distributed environments. In the next section, we give a brief overview of how a B&B algorithm operates before switching to a literature review on parallel approaches for B&B.

2.2.2 The Branch-and-Bound Algorithm

2.2.2.1 Main Operation

Let us consider a set S of all candidate solutions with respect to an optimization problem and a real-valued objective function f which associates a numerical value to every solution in S. Remark that a candidate solution is not necessary a feasible solution with respect to problem constraints, but all possible feasible solutions are assumed to be included in the set S. Without loss of generality, our goal is to find the solution(s) in S minimizing the value of the f function: $\arg\min_f(S) =$

$\left\{ s \in S \mid f(s) = \min_{r \in S} f(r) \right\}$. In the following, we assume that the set S of feasible solutions is finite, thus making our problem be a discrete optimization problem. The Branch-and-Bound algorithm [Gendron 1994, Papadimitriou 1998, Ralphs 2003] can be viewed as an implicit enumeration of all the possible solutions in S in order to find the best one(s) w.r.t function f. This algorithm takes its origins in the years 1960's: Land and Doig [Land 1960] designed an iterative algorithm to solve what they refer to as "Discrete Programming Problems". Dakin [Dakin 1965] later proposed improvements to this algorithm and proposed a tree-based algorithm for mixed integer programming. Those two algorithms were among the first one to be classified as "Branch-and-Bound" techniques, as defined by Little, Murty, Sweeney and Karel [Little 1963].

From a more technical point of view, this algorithm decomposes the original optimization problem into several subproblems of smaller size. The resulting search tree is explored by building a tree where the root node represents the entire problem to solve, inner nodes represent subproblems and leaves represent solutions for the problem being solved. This exploration process can be decomposed into four mechanisms: branching, bounding, elimination and selection.

1. First, the algorithm performs the branching operation. It is also referred to as decomposition and consists in partitioning the set of feasible solutions for a given problem into smaller subsets (subproblems) on which the same optimization problem applies. These subproblems are recursively decomposed until the solution level is reached (leaves in the search tree), or these subproblems are not likely to improve the best solution found so far.

2. Second, the algorithm bounds the generated subproblems. It computes a lower bound of the optimal solution for the problem, using data available from the generated subproblem. Different bounding functions are available in the literature. We use the well-known lower bound proposed by Lageweg et al. [Lageweg 1978] for the Flow-Shop Scheduling problem, based on the Johnson's algorithm [Johnson 1954].

3. Third, the B&B proceeds with an elimination process. The purpose is to avoid exploring subproblems that will likely not lead to the discovery of the optimal solution. This process is achieved in two steps. The algorithm (i) determines whether the solutions are feasible or not (and then discards non-feasible solutions) and (ii) compares for the remaining subproblems their lower bound with the best solution found so far[1] and discards subproblems whose bound is greater (for minimization problems).

4. The last step of the B&B exploration process is to determine a policy for exploring the remaining subproblems to be explored. More precisely, one has to determine which problem will be selected for exploration. Multiple

[1] Solution which can be seen as an upper bound of the optimal solution.

strategies exist in the literature and are described in detail in the following section.

Therefore, the Branch-and-Bound algorithm consists in recursively branching and bounding (sub)problems aside eliminating some of these and exploring the remaining ones according to a predefined strategy.

The previous exploration steps are repeated until all solutions are explicitly or implicitly visited. The B&B is then guaranteed to output the optimal feasible solution(s) for the problem being solved. It should be clear from the previous discussion that the B&B exploration process can be seen as a tree-based process. In fact, the original problem (whole set S) can be seen as the root of the tree and the subproblems resulting from its decomposition as its direct children nodes. The subproblems resulting from the decomposition of a subproblem can be recursively considered as children nodes, and so on until we end up with single solutions being the leaves of the tree.

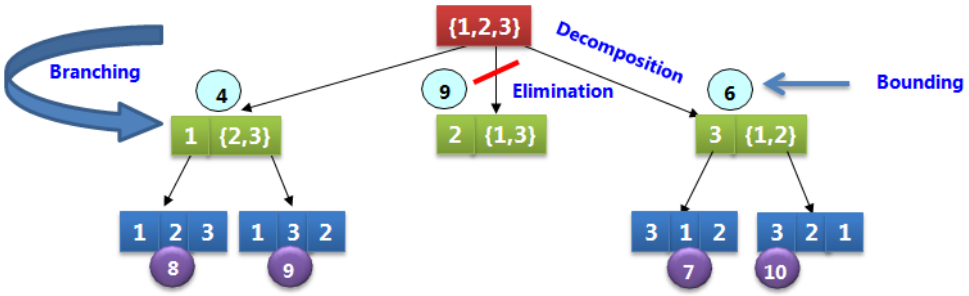

Figure 2.2: General illustration of the operation of the Branch-and-Bound algorithm for a 3-element permutation-based problem.

2.2.2.2 Exploration Strategies

After a problem is decomposed into subproblems, it is necessary to determine which subproblem will be explored next, according to a specified strategy. Three main exploration strategies can be distinguished:

- **Depth-First Strategy:** This policy assumes the existence of an arbitrary order among subproblems. When a problem is decomposed into several subproblems, this strategy consists in always exploring those subproblems according to the so-defined order. Let us consider a set of n subproblems ordered from S_1 to S_n at some iteration of the B&B algorithm. This strategy then begins exploring subproblem S_1 first. If this subproblem can be decomposed further, then the first (with respect to the considered order) of its subproblems will be explored first, then the second subproblem and so on recursively until the subtree rooted at S_1 is completely explored. When the exploration is done with subproblem S_1, then and only then the second subproblem S_2 can be

explored. The procedure is then recursively applied until the whole tree is explored. Note that this exploration process is independent from the numbering of the subproblems. Here, problems are labeled according to the order they were generated by the problem decomposition mechanism, that is S_1 is the first subproblem generated during the decomposition.

- **Best-First Strategy:** This strategy relies on the lower bounds obtained when evaluating generated subproblems. It assumes that, when a set of subproblems is generated, the optimal solution has a greater probability to be contained in the subproblem having the "*best*" (lowest) lower bound. When this subproblem is explored, the algorithm explores the subproblem whose lower bound is immediately higher than the previous one, and so on. This policy implicitly assumes that the bounding function is *discriminating*, meaning that, all the subproblems resulting from the same decomposition have different lower bound values. Nevertheless, this limitation can be overcome by defining a *sub-policy* to deal with *equal priority* subproblems.

- **Breadth-First Strategy:** This policy does not introduce any arbitrary order in the exploration of subproblems. When a problem is decomposed into several subproblems, all the generated subproblems are first evaluated then some may be discarded by the B&B algorithm. The others are again decomposed into new subproblems. More technically speaking, let us consider a problem B. The strategy consists in generating its $B_{i\in[1,n]}, n \in \mathbb{N}$ subproblems at once, evaluating them all, possibly branching them and then, for all the remaining B_i subproblems, their subproblems $B_{i,j\in[1,n']}$ are generated and so on. The main advantage of this strategy is that it generates many large subproblems at the beginning of its execution. In the case of the Branch-and-Bound algorithm, it enables the branching operator to discard potentially large sets of solutions and speed up the resolution process. One can note that this exploration strategy requires to store a potentially sizeable list of subproblems in memory.

Now that we reviewed the main generic components of a Branch-and-Bound algorithm, we give an overview on which parts of the algorithm can be parallelized and how this parallel strategies can be effectively deployed. This is the goal of the next Section where we start providing a general overview of grid-based environments.

2.3 Parallel Branch-and-bound literature overview

In this section, we give an overview of the different works existing in the literature as well as the parallel models of Branch-and-Bound algorithms. Deploying parallel B&B algorithms requires to take into account the characteristics of the target execution environment. Therefore, we shall first study these parallel environments as well as their characteristics.

2.3.1 Large scale Grid Environments

A *Grid* can be viewed as a set of computational resources scattered over several geographically distributed sites and connected through a wide-area network. In the following, we present the main properties defining more precisely a grid environment [Melab 2005].

2.3.1.1 Multiple administrative domains

First, the computing resources in a grid environment are typically distributed among multiple administrative domains and managed by different organizations. With respect to this property, one major issue appearing in grid computing is security. In fact, very often, users and resources providers can be clearly identified, which allows one to improve security. Nevertheless, communicating between multiple sites typically through *firewalls* can be an obstacle and pauses challenging issues. In global computational systems like XtremWeb [Fedak 2003], which are based on Internet-wide cycle stealing, this issue can be overcome quite easily as these systems are based on voluntary computing, meaning that computational units initiate communications from inside the administrative domain. In Condor [Litzkow 1988], multiple approaches for harnessing multi-domain resources as if they all belong to a unique domain, have been designed such as *Condor-G* and *Condor Glide-in* [Frey 2002] or *flocking*. The two first approaches are coupled with the Globus platform [Foster 1997] whereas the latter solution consists in implementing a resource manager for each administrative domain. These resources managers exchange the tasks which can not be processed locally.

2.3.1.2 Heterogeneity

A grid is by essence heterogenous. Resources heterogeneity, whether at the hardware level or the software level, can be viewed as the consequence of the large variety of equipments composing the grid and the large variety of software tools that could be run on them. A grid can integrate hardware from multiple vendors, run various operating systems, and use different network protocols for remote communication, etc. In contrast, a single site of the grid is often composed of homogeneous resources, mostly aggregated into clusters.

2.3.1.3 Large-Scale systems

Due to the number of available computational units and the wide-area network interconnection infrastructure, a computational grid is a large-scale system. Depending on the considered scale, a distinction is generally made between computational grids on the one hand, and global/P2P computing systems on the other hand. Those latter are usually believed to enable the aggregation of much more computational resources and are dedicated to voluntary computing. One of the most famous voluntary computing systems is *SETI@Home* [Milojicic 2008, Anderson 2002]. We can

also cite XtremWeb [Fedak 2003] which can be considered as a voluntary computing system but serves more general purposes as it aims to convert any set of resources into a fully operational grid-like environment. In such large scale systems, computing intensive applications are not guaranteed to scale with the system. This is typically due to communication latency and load imbalance issues occurring in such heterogenous and large scale networked systems. Designing distributed protocols which are both scalable and efficient is one of the most challenging tasks to achieve high performance using grid environments.

2.3.1.4 Dynamic environment

The last major property characterizing a grid environment is resources' volatility. By volatility, we mean the fact that computing resources are not expected to be always available for the application. This is due to hardware crash, software issues or any other system variance. Volatility should not be viewed as unlikely or as an exception in a grid environment. In fact, the probability that resources are operational and available for the application is usually observed to decrease as the amount of aggregated resources increases. Volatility poses several challenging issues such as: dynamic resource discovery, fault tolerance, data recovery, synchronization, etc. These issues are difficult to deal with at a hardware level as they are mainly dependent on the nature of the application being executed. They are instead tackled mostly at the application level.

2.3.2 Parallel models for the B&B

The Branch-and-Bound algorithm can be parallelized according to multiple models, each one focusing on a specific element of the algorithm. Here, we give an overview of these different models based on the classification proposed by [Gendron 1994, Cung 1994, Melab 2005]. Four models have been identified: *multi-parametric parallel model, tree-based exploration parallel model, parallel bounds evaluation model* and *the parallel evaluation of one bound*.

2.3.2.1 Multi-parametric parallel model

The multi-parametric parallel model, not often studied in the literature, consists in considering multiple B&B algorithms (See Figure 2.3), which is a coarse-grained model. Multiple variants of this model can be derived by modifying one or several parameter(s) of the algorithms. These algorithms may only differ through the separation operator (used to split a problem into sub-problems) in [Miller 1993], the selection operator (determining the order in which sub-problems are explored) in [V.K. Janakiram 1988] where a variant of the *depth-first* strategy is used for exploration. Each algorithm randomly selects the next subproblem to explore among the last generated subproblems. In [Kumar 1984], each algorithm uses a different upper bound in the experiments. The main idea is that only one algorithm uses the best solution found so far while others use this value increased by a value $\varepsilon > 0$. Another

version of this parallel model consists in splitting the interval composed of the lower bound for the problem and the best solution found so far into sub-intervals. Each sub-interval is assigned to one of the algorithms.

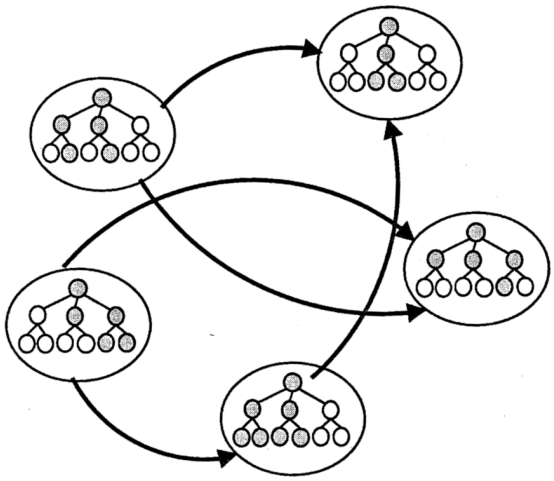

Figure 2.3: Illustration of the Multi-Parametric parallel model.

The main advantage of the multi-parametric parallel model is its genericity, enabling its use transparently for the user. Its main drawback is the additional cost of exploration as some nodes of the exploration tree are visited multiple times. Besides, as the number of algorithms involved is low, this model shall be used on grids along with other parallel models.

2.3.2.2 Tree-based exploration parallel model

The tree-based exploration parallel model consists in exploring in parallel multiple subtrees corresponding to subsets of the search space for the problem. (See Figure 2.4). This implies that the separation, selection, evaluation and branching operations are executed in parallel, (a) synchronously by different algorithms exploring these sub-spaces. In the synchronous mode, the Branch-and-Bound algorithm contains multiple phases. During each phase, algorithms perform exploration independently from each other. Between the phases, algorithms synchronize to exchange information, like the best solution found so far. In the asynchronous mode, algorithms communicate unpredictably.

In comparison with other models, the tree-based exploration parallel model is more attractive and is the main focus in many works for two main reasons. On the one hand, the degree of parallelism of this model can be very high for large-scale problems, which could justify by itself the use of a computational grid. On the other hand, the implementation may be faced to multiple parallel programming challenges. Among these issues, one can cite the localization and the management of the list of sub-problems to solve, the load balancing, the communication of the best solution found, the detection of the algorithm's termination, and the fault tolerance.

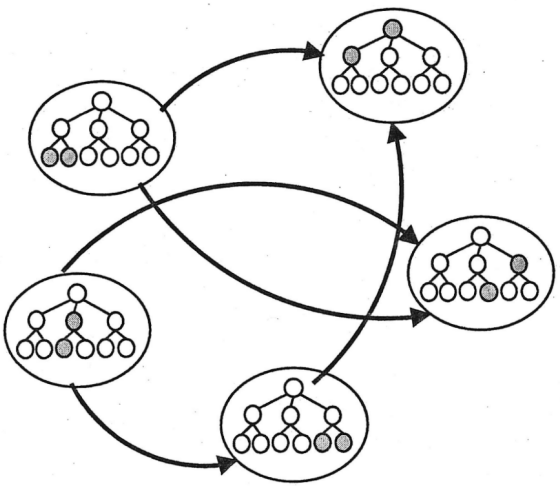

Figure 2.4: Illustration of the Tree-Based Exploration parallel model.

In [Gendron 1994], one can find an analysis of synchronous and asynchronous Branch-and-Bound algorithms. It appears that synchronization comes to be useless and inefficient on homogeneous machines with message passing features, consisting of less than 32 processors. Its usage on computational grids (heterogeneous and volatile) is not recommended. Still in this article, a study has also been conducted on the problem of localizing and managing the remaining sub-problems. Two categories of algorithms have been identified: those using a unique list of subproblems, shared by all processes and those using multiple lists. In the conclusions of this study, the authors claim that the use of a single list can be interesting for problems where the calculation of the bound is not trivial, and for machines (especially with shared memory) with a low number of processors. Therefore, the use of a single list is not recommended when using grids. In our works, we will focus only on asynchronous algorithms using multiple lists to store generated subproblems as they can be seen as more adapted for grids. Three approaches exist for managing these lists: *collegial, grouped* and *mixed* [Gendron 1994]. In the collegial approach, each process manages its own list where it stores the subproblems it generates. The grouped organization consists in defining sets of processes. A global list is defined for these processes to store all the subproblems they generate. The mixed approach is a collegial approach where each process manages its own list and simultaneously shares a global list with all other processes. As the collegial approach is fully distributed, it can be expected to have a greater scalability. However, the mixed approach can be efficient for grids with multiple administrative domains: processes belonging to the same domain can share a common list.

The load balancing issue consists in minimizing the number of situations where processes have many sub-problems to explore while others have empty lists. A load balancing policy is necessary. Its aim is to initiate, at some appropriate times, transfers of sub-problems from overloaded machines to under-used machines. Three agents define a load balancing policy: an information agent, a transfer agent and

a localization agent [Melab 1996, Melab 1997]. The first agent collects information about the load of processors. It provides a load indicator (number of sub-problems, CPU usage, ...), which allows to define the load of a machine, and an information exchange strategy (centralized, distributed or hierarchical). The transfer agent can use a passive, active or mixed strategy. In the passive strategy (respectively active), transfers are initiated by under-used (resp. overloaded) processors. The mixed strategy combines both. The localization agent allows to determine the receiving process (often the least used) for the sub-problems in excess in the overloaded machines. In a grid-like environment, to take into account the machines' heterogeneity, their power should be included into the definition of the load indicator. Gathering information at large scales can induce a communication bottleneck when the global load increases rapidly. In the large-scale cycle-stealing model, used in our works, heterogeneity is considered in a more natural way: powerful machines ask for more work to weaker machines.

The best solution found by a process can be communicated to other processes whether through a shared storage space or by broadcasting mechanisms. The first approach consists in the use of a global variable to store the best solution found so far. This variable is updated each time a process finds a solution better than the current one. The communication of this value to other processes can be achieved through broadcasting or using any approach based on propagation through neighbors. The broadcasting method shall not be considered in large scale environments. The issue of termination detection is not specific to B&B algorithms. In our works, a process detects termination through gathering information from other processes.

The fault tolerance issue for exact exploration methods is crucial. Indeed, the loss of one or more sub-problems can prevent the processes from discovering the optimal solution for the problem being solved. On a volatile computational grid, the issue can be handled whether at the middleware level of at the application level. At the middleware level, the proposed solutions are often independent from applications. The failed process is restarted from scratch, which can be inefficient for processes with long lifetimes. At the application level, the issue is handled through a *checkpointing* mechanism. It allows to restart a failed process from its last saved state. Thus, it is crucial to determine which data must be saved. The data saved by each process usually contains the best solution found so far, the current problem being explored and the list of pending sub-problems. Other data can be saved if the current parallel model is combined with other models.

2.3.2.3 Parallel evaluation of solutions/bounds model

The Parallel evaluation of solutions/bounds Model can be compared to the Parallel Population Evaluation Model, often used with metaheuristics. This model is data parallel and allows the parallelization of the generated subproblems. It can be used when the evaluation of the bounds is performed entirely after the generation of the sub-problems. The B&B algorithm is not changed: only the evaluation phase becomes faster. The main advantage of this method is its genericity (See Figure 2.5).

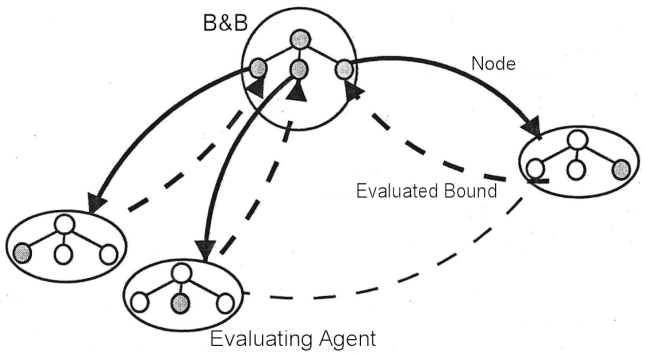

Figure 2.5: Illustration of the Parallel evaluation of solutions/bounds model.

However, in a grid environment, it can be inefficient for the following reasons: (i) the model is intrinsically asynchronous, which makes it costly in terms of CPU time within a volatile and heterogeneous environment; (ii) its granularity (the cost of the bound evaluation function) can be fine and thus, inefficient in large-scale environments. For instance, in the case of the Flow-Shop Problem, the processing cost of the bound's evaluation may not be high enough to justify its parallelization. (iii) The degree of parallelism of this model depends on the problem being solved. It is often limited and decreases along with the exploration process: the number of generated sub-problems decreases when the number of constraints for the problem splitting operation increases. Combining this model with the tree-based exploration model can induce a massive parallelism which can be efficiently employed only within grid environments.

2.3.2.4 The Parallel Evaluation model for a single bound/objective function

This model can be used for the resolution of real-world problems where the evaluation of the objective function/lower bound requires to access to massive amounts of data that can not be handled on a single machine. Because of this hardware constraint, those data are distributed among multiple sites. The evaluation of the objective function takes advantage of the data parallelism. The parallel evaluation of the objective function can be interesting when it is costly in terms of time.

This model requires the definition of new specific elements for the problem being processed, like partial objective functions and a function to aggregate these partial results. As the implementation of this model is naturally synchronous, it is crucial to memorize all the partial evaluation value for the solution being evaluated, to manage fault tolerance and variable availability of the resources in a grid. As its scalability can be very limited, this model shall be combined with other models.

2.3.3 Deploying parallel B&B

Having these different parallel strategies in mind, two main paradigms can used to make a parallel B&B algorithm effective and to deploy it over a distributed computing environment, namely, the Master-Slave model and the P2P model. In the next sections, we give a literature review on these models while focusing on their main characteristics with respect to B&B.

2.3.3.1 Master-slave approaches

In [Aida 2002, Aida 2005], a Master/Slave-based parallel B&B algorithm is proposed and deployed on a grid. This approach is aimed to avoid performance degradation caused by the communication overhead between the master process and worker processes. Processes are organized into sets, where each set comprises a group of worker processes and one master process to coordinate them. In addition, a process called *Supervisor* is in charge of controlling and coordinating all the sets of processes. One set of worker processes explores a given part of the search tree. The supervisor assigns a subset of solutions to the master of the set and this master dispatches the work to its worker processes. One can see the supervisor as an entity performing load balancing operations between the sets of processes. This supervisor, as well as the master process of each set of processes, is in charge of gathering and broadcasting the best solution found so far, thus accelerating the exploration process. The latter approach shows a limited scalability as it may create a bottleneck on the Master processes and the Supervisor process. 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 [Seymour 2002], Ninf-G [Tanaka 2003], and Ninf [Sato 1997].

The granularity issue is studied in [Di Constanzo 2007] but yet a hierarchical Master/Slave model is used therein. The architecture of the approach, named *Grid'BnB* is quite similar whereas the communication layer is a bit different. The application's design is to fit a grid environment. It is composed of four different types of entities: *master*, *sub-master*, *worker*, and *leader*. The master has the same role as the supervisor in the previous approach. The sub-master is in charge of coordinating one set of worker processes. The difference comes from the *leader* role. The approach assumes that the physical architecture is cluster-based. Each set of processes comprises several workers and is deployed on a physical cluster. The cluster running the master process also hosts the sub-master processes which are in charge of communicating with other clusters (see Figure 2.6). In those other clusters, one worker is chosen to be the *leader*. It is given a specific role, which is to handle communications with its sub-master. Thus, when a worker discovers a new best solution for the problem being solved, it broadcasts it to all the workers belonging to the same cluster, including the leader. This leader sends it to its sub-master process, which broadcasts it to the whole network through the master

process.

A middleware has been designed from this application. It enables to emulate Master-Slave based parallel B&B techniques on P2P networks or platforms. The main difference with a classical hierarchical Master-Slave model is that messages are relayed through peers towards the master and the sub-masters. This approach is less exposed to a communication bottleneck issue but it may end up with huge communication delays as these communications can be relayed through a great number of entities before reaching their destination point. The main goal of [Di Constanzo 2007] is in fact to design a middleware that hides the network architecture/topology for computational applications.

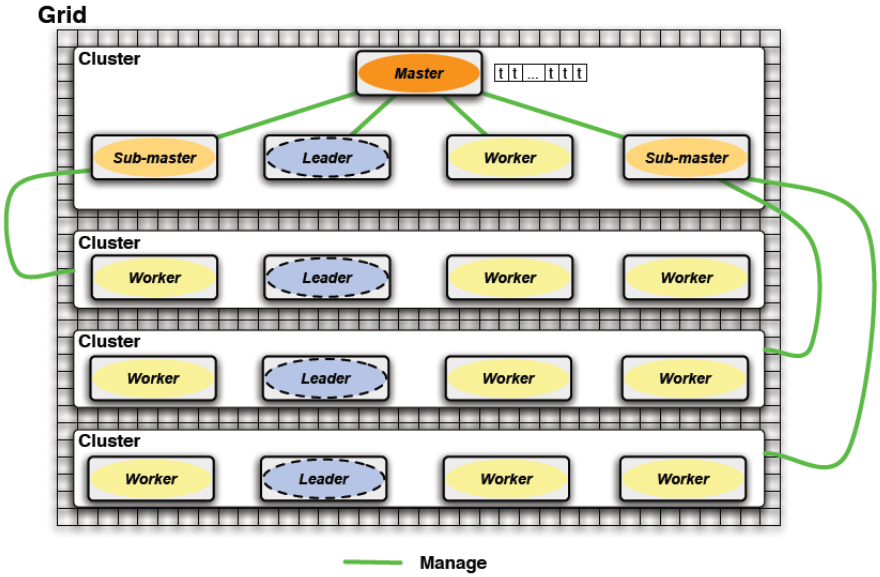

Figure 2.6: The Grid'BnB architecture

Xu *et al.* [Y. Xu 2005] and Eckstein *et al.* [Eckstein 2000] proposed respectively *ALPS* and *PICO* which are parallel B&Bs based on Master-Hub-Worker 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 worker(s). 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.

In [Drummond 2006], Drummond *et al.* propose another hierarchical B&B algorithm designed for grid environments. This approach is applied to the *Steiner* problem in graphs. In this problem , the branching operation creates two new sub-

problems. The approach performs the following operations: a master is run on the processor of one given cluster. This entity runs and monitors a set of *leader* processes, one on each cluster. Those *leaders* represent the processors of the cluster on which they are running. A leader splits its subproblem into two *left* and *right* subproblems. The *right* one is assigned to another leader. This process is repeated until no leader is available. Then, each leader processes its subproblem by sharing it with the workers being run on its own cluster.

In [Bendjoudi 2009], the authors suggest a hierarchical architecture to allow workers to communicate directly together after receiving a task from the master, the redundancy induced by [Mezmaz 2005]'s approach when exploring the search space can be significantly reduced.

Bendjoudi *et al.* [Bendjoudi 2011] have proposed a fault tolerant hierarchical B&B (See Figure 2.7), named FTH-B&B, in order to deal with the fault tolerance and scalability issues in large scale unreliable environments. Their algorithm is composed of several fault tolerant M/W-based B&Bs, organized hierarchically. Some other works, e.g., Cabani *et al.*'s PHAC ([Cabani 2007]), Saffre *et al.*'s Hypergrid [Saffre 2003], aim to dynamically redesign the topology in order to face communication bottleneck issues.

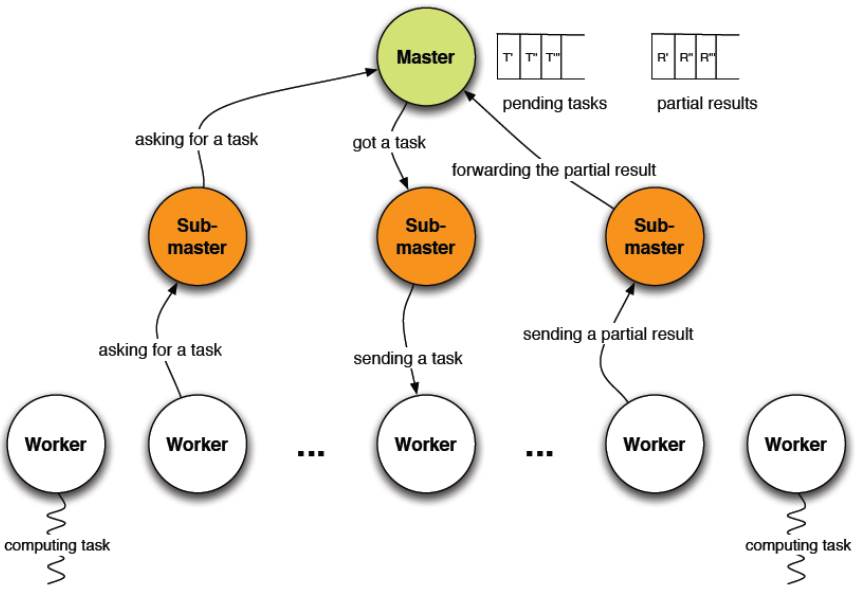

Figure 2.7: Illustration of a Hierarchical Master-Slave based architecture

In [Mezmaz 2005], an efficient encoding of the search space is proposed to reduce the size of exchanged messages. The overall parallel efficiency is then improved compared to previous solutions. The approach of [Mezmaz 2005] can be considered as the best parallel Master-Slave B&B approach that can be applied in a large scale computational environment such as grids [Mezmaz 2007b]. In particular, it was successfully applied to find the optimal solution of an unsolved Flow-shop hard instance, namely the Ta056 instance [Taillard 1993, Mezmaz 2007b]. The latter

approach is also based on the Master-Slave model. We describe it more in details later.

2.3.3.2 Peer-to-Peer approaches

Parallel applications designed under the Master-Slave paradigm may often face scalability issues due to bottlenecks on the master. To overcome this limitation, some works consider the use of the Peer-to-Peer (P2P) paradigm for parallel B&B as in *DIB* by Finkel *et al.* [Finkel 1987] and in the work proposed by Iamnitchi *et al.* in [Iamnitchi 2000]. The latter proposes a fully decentralized approach for the Branch-and-Bound algorithm. The role of each process is to manage a local work pool and share it with other processes whenever they receive a request. The best solution is broadcast over the network through the most frequently sent messages. By distributing the communication load among multiple processes, this approach gains in terms of scalability.

Instead of dealing with the scalability issues at the applicative level, Di Constanzo *et al.* [Di Constanzo 2007, Caromel 2007] propose a more generic approach operating at the communication layer. The approach consists in defining a fully P2P infrastructure and providing an information routing mechanism. Processes are organized in a P2P fashion: one of them acts as the master and all others as slaves. Whenever a slave needs to communicate with the master, its messages are routed/relayed by multiple peers before reaching the Master. While this approach enables to provide a better scalability, it only distributes the communication load of the master through time and introduces additional delays for processing slaves' requests.

More generally, P2P systems are used in wider fields of application, like voluntary computing as in SETI@HOME [Anderson 2002] where a central server collecting radio signals to be processed dispatches the tasks among personal machines provided by individuals around the world. Other systems were designed for content sharing purposes like Bittorent [Bittorrent 2005], Kademilia [Maymounkov 2002], Pastry [Rowstron 2001a], for massively distributed computing like Javelin [Cappello 1997], for data storage and retrieval like OceanStore [Kubiatowicz 2000], Freenet [Clarke 2001], NaradaBrokering [Fox 2005], PAST [Druschel 2001].

With respect to the large body of literature dedicated to Master-Slave based parallel B&B, there is relatively few works dealing with parallel B&B in P2P networks. In this thesis, we propose to take benefit from the scalability properties of P2P network in order to design new distributed B&B algorithms that are able to handle a huge amount of computational resources. More precisely, we leverage the tree-based Master slave approach of [Mezmaz 2005, Mezmaz 2007b] by providing a fully distributed alternative. Although, the approach described in this thesis and the one of [Mezmaz 2005, Mezmaz 2007b] share the same B&B work encoding, they are fundamentally different, since the underlying distributed protocols are different. In the next section, we give a more detailed overview of the parallel approach

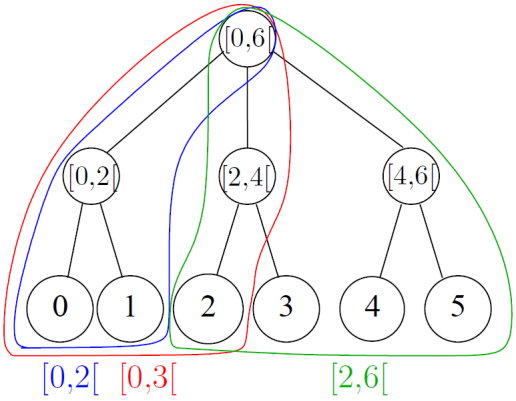

Figure 2.8: Example with a 3-variable permutation.

of [Mezmaz 2005, Mezmaz 2007b] which allows the reader to better appreciate our contributions.

2.4 The B&B@Grid approach

In this section, we analyze more thoroughly the Master-Slave based parallelization of the Branch-and-Bound Algorithm designed by Mezmaz *et al.* [Mezmaz 2005, Mezmaz 2007b], named B&B@Grid, which is, to the best of our knowledge, the most scalable 'Master-Slave'-based approach in the literature. This approach was designed for grid environments and parallelizes the B&B algorithm according to the tree-based exploration parallel model presented in Section 2.3.2.2. Some of the mechanisms of this approach are reused or referred to in our works.

2.4.1 Tree encoding

For the sake of clarity let us assume that we are given a permutation-like optimization problem, that is a problem where solutions can be encoded by mean of a permutation of size N. Hence, a basic B&B strategy can be represented by a tree where the root designates the problem to be solved, a leaf represents a solution (a permutation) and a node inside the tree represents a partial solution (equivalently, a sub-problem) where only some variables in the permutation are fixed. From a parallel/distributed point of view, many B&B algorithms are based on depth-first strategy when exploring the search space [Zhang 2000, Prieditis 1998, Mans 1995]. A sequential depth-first strategy explores the tree in a depth-first manner branching and bounding according to some branching and bounding policies. A parallel version of this strategy is to run several depth-first explorations in parallel on different parts of the search tree. As soon as a new best solution for the problem is found, it is communicated to other processes which allows them to update their own local best solution and thus to speed up the overall search process.

One major difficulty when setting up this general idea in a distributed environment is to carefully encode the nodes of the B&B tree in order to reduce the cost of exchanging messages between computational nodes and decide which part of the tree should be explored by which computational node. A trivial approach would be to encode a sub-problem of the B&B tree in a fully-comprehensive way by providing all necessary information. In [Iamnitchi 2000], a tree node (a subproblem) is encoded as a path from the root to the node itself which is still expensive. In [Mezmaz 2005, Mezmaz 2007b], an extremely simple and efficient encoding is proposed. Roughly speaking, the tree is labeled in such a way a sub-set of nodes (corresponding to a subproblem) can be encoded by an interval, i.e., two integers. A centralized model is then adopted to distribute intervals among computational nodes. More specifically, the approach described in [Mezmaz 2005, Mezmaz 2007b] consists in defining a central computational entity (the master) which is in charge of controlling intervals (work units) assignment to other computational nodes (the slaves).

In this thesis, we adopt the interval-based encoding of [Mezmaz 2005] while removing the need of the central coordinator. We briefly give an example to illustrate the tree encoding. For the sake of clarity and to make our results more comprehensive, we also recall the basic ideas of the Master-Slave approach and the main challenges we are addressing. Figure 2.8 gives a simple example with a p-variable permutation problem, (with $p = 3$) i.e., the optimal solution of our problem is one of the $3! = 6$ possible ordering of our variables. More precisely, a label (integer) is assigned to each leaf of the tree, corresponding to a specific permutation. Then, an interval $[x, y[\subseteq [0, p![$ refers to a subtree of the whole search tree. For instance, one can see in Figure 2.8 the subtrees corresponding to intervals $[0, 3[$, $[0, 2[$, and $[2, 6[$. Having this labeling, two operators, defined in [Mezmaz 2005], allow us to switch from the tree representation to the interval-based representation and conversely. Notice that exploring intervals $[0, 3[$ and $[2, 6[$ in parallel implies exploring the same region of the search space (permutation 3) twice, while exploring only interval $[0, 2[$ and $[2, 6[$ may fail to produce the optimal solution since permutation 2 is not explored.

2.4.2 Master-Slave tree exploration

In [Mezmaz 2007b], this tree encoding is coupled with a Master-Slave model. More precisely, the master owns initially the whole interval to be explored, i.e., the whole tree. Then, slaves ask the master for a sub-interval (a piece of the tree) to explore. The master continuously removes from the list the subintervals that are already explored and distributes those not explored yet. However, since many slaves having different computational power can ask for some work at the same time, some Slaves may end exploring the same part of the search tree inducing a so-called "*redundancy*". Redundancy issue is highlighted in [Mezmaz 2007b] and work dealing with this issue can be found in [Kumar 1984]. We also remark that each time a slave finds a new solution better than the best solution found so far, i.e., an ordering of

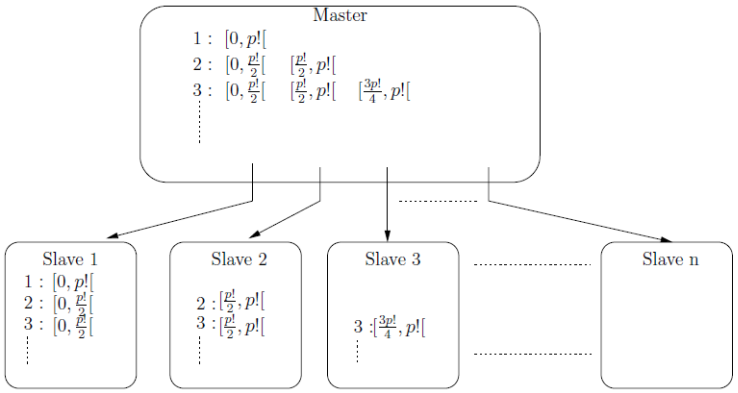

Figure 2.9: Operation of a Master-Slave based architecture.

variables, it directly informs the master which broadcasts this information to other slaves. Hence, sharing the best known solution is straightforward and does not induce any technical difficulty from a pure distributed/parallel point of view. In addition, termination of the whole distributed search process can be easily detected by the master, i.e. when the list of remaining sub-intervals maintained by the Master becomes empty. Load balancing is also quite simple since the master can control slaves progress and reassign work unit directly to under-used slaves.

A concrete scenario is proposed in Figure 2.9. Assume that we have initially n slaves. Assume that a first slave wants to start participating in the exploration process. Then, it sends a work request to the master. When receiving that request, the master answers with the whole problem to process (since until now the master has not received any request). In the second step, suppose that the master receives a work request from a second slave. To balance the load, the master splits the original interval into two parts and shares it equally between both slaves. Now, suppose that in the third step, a new request coming from slave 3 reaches the master. The master shall choose one of the already assigned intervals, split it into two parts and assign one of them to Slave 3. In the scenario of Figure 2.9, the Master chooses the interval assigned to slave 2 and shares it with slave 3. However, at this point of the execution, slave 2 is not aware of its newly assigned sub-interval $[\frac{p!}{2}, \frac{3p!}{4}[$. As depicted in Figure 2.9, it may explore the solutions belonging to interval $[\frac{3p!}{4}, p![$, i.e. the interval assigned to slave 3, leading to "*redundancy*". This issue is handled in [Mezmaz 2007b] as following. The master and the slaves continuously checkpoint together and update their intervals each fixed period of time. Depending on search speed and on whenever the checkpoint is done, redundancy may or may not occur. In our example, slave 1 has updated its work unit whereas slave 2 has not yet. However, in [Mezmaz 2007b] this solution was shown to be efficient. More precisely, experimental results have shown that the redundancy rate is of 0.39%. While this rate may appear to be low, as we are dealing with large-scale instances, it represents a huge amount of processing time. The associated total computation cost could be

highly significant if the exploration time of each node is high.

2.5 Conclusion

In this chapter, we provided a comprehensive description of the Branch-and-Bound algorithm and gave an overview of the main contributions related to the parallelization of this algorithm. We introduced the interval-based encoding of parallel B&B tasks which we use through this thesis. We also highlight the main characteristics of existing Master-Slave approaches for B&B. The rest of this thesis is devoted to provide a fully distributed P2P approach to Master-Slave approaches and assessing its efficiency.

Chapter 3

B&B-P2P : a P2P approach for scalable Branch-and-Bound on fully distributed environments.

Contents

3.1 Introduction

Most of the works related to the parallelization of the B&B algorithm are based on a Master-Slave architecture. A Master-Slave architecture can be viewed as a star graph where the center of the star is the Master and the leaves are the Slaves. Although it allows one to have a global view of the network, this topology fails to scale efficiently when a huge number of computational entities is involved. In this chapter, we propose a new fully decentralized approach to distribute the computation of parallel B&B among possibly a huge number of computing entities. Our goal is to gain in scalability while handling more and more distributed resources. For that purpose, we propose to view computing resources as *peers* and to organize them into a peer-to-peer logical overlay, thus getting rid of the centralized bottleneck induced by the master-slave organization of resources. However, in order to evolve in such a fully distributed environment, many challenges and issues have to be tackled and solved.

Compared to a master-slave approach, information about search and system state can not be handled in a centralized manner when dealing with a peer-to-peer structured network. It is thus necessary to rethink the mechanisms of the B&B algorithm to adapt them carefully. In particular, a peer should manage global information (like the best incumbent solution for the B&B or the termination of the computation) with only a limited view of the network — by communicating only with its immediate neighbors. Besides, when designing a parallel algorithm under the Peer-to-Peer paradigm, every peer has to manage its own local work pool and eventually to share it with its neighbors in the network. Setting up such an architecture implies ensuring two important properties:

- First, we want to gain not only in scalability but also in Parallel Efficiency. Indeed, the entities will have to self-coordinate the B&B search process and share work efficiently while having a local view of intervals being explored. Handling an increasing number of computational nodes organized according to a pure Peer-to-Peer overlay implies more communications and message exchanges which could lead to efficiency loss if not managed very carefully.

- Second, we want to guarantee that the parallel search process *terminates correctly* by effectively computing the optimal solution, which is not straightforward without any global view of the network.

In this chapter, we describe the peer-to-peer approach we designed to handle these two major issues in a static environment where computational nodes are fully available and do not crash. In the *description* of our approach, we shall first assume that the computational nodes are organized according to any logical overlay. In other words, each computational node can communicate with only a subset of other computational nodes called its neighbors. We assume that the considered overlay can be any connected graph $G = (V, E)$, i.e., a node $v \in V$ designates a computational node and an edge (u, v) designates the fact that u and v are neighbors in the overlay. For the sake of simplicity, we also assume that we are given a permutation optimization problem to be solved in parallel by a B&B algorithm. We then use the interval-based encoding[1] introduced in [Mezmaz 2005], and assume that the problem to be solved is initially pushed into a unique peer in the system. Having these assumptions in mind, we shall first provide a general and generic description of the main algorithmic ingredients embedded in our framework by describing the main operations performed by a peer during the whole execution. Our goal is to first highlight the main algorithmic difficulties we are facing and the way we are solving them when designing a generic parallel B&B running in a fully distributed environment. The described approach is in fact overlay *independent*, in the sense that it operates correctly without taking into account any specific property of the considered overlay. However, the choice of the overlay is important for such an approach to perform efficiently. Therefore, after describing our approach, we shall

[1] Notice that any other B&B work encoding could be plugged in our approach as far as a mechanism for splitting work is provided

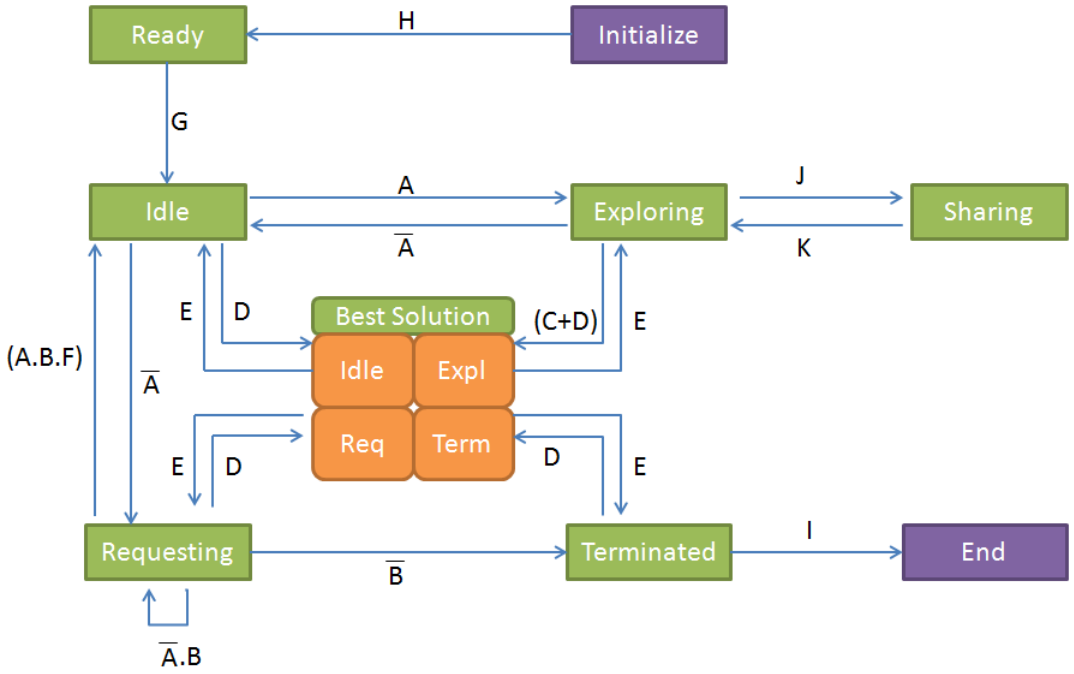

Figure 3.1: State automata representing the behavior of a Peer.

highlight how it is expected to perform according to the properties of the chosen overlay. As we will argue, choosing a scalable overlay is crucial in order to obtain a good balance between communication cost and computing efficiency.

The rest of this paper is organized as following. In Section 3.2, we give a general overview of the approach. In Section 3.3, we give a description of the mechanisms dedicated to sharing the tasks among peers and to managing the information related to the best incumbent solution found during the B&B exploration. In Section 3.4, we provide technical elements about the design of the mechanism related to Termination Detection. Finally, in Section 3.5, we discuss the scalability allowed by the peer-to-peer overlay and its expected impact on the overall performance.

3.2 Overview of the approach

We use the state automaton of Figure 3.1 to summarize the main operations performed by a peer. This mainly include the exploration task, the best solution sharing task, the work sharing task and the termination detection task. The logical propositions related to the state automata in Figure 3.1 are as following:

- $A = [x, y] \neq \varnothing$
- $B =$ Number of requests strictly lower than the network's diameter.
- $C =$ A new improving solution is found.

- D = A new improving solution is received from a neighbor.
- E = The new improving solution has been broadcast to neighbors.
- F = Received all replies (The current round of requests is finished).
- G = Neighbors are ready to communicate. (They are in the "Ready" state.)
- H = Initialization procedure complete.
- I = Neighbors have all claimed the termination of the calculation.
- J = Received Work Request.
- K = Sent back Work Reply.

As depicted in Figure 3.1, in our approach a peer first **initializes** itself by determining its neighbors according to the input logical overlay. When it is **ready**, it needs to ensure that its neighbors are also ready to communicate so it waits for them to be ready as well. Once its neighbors are ready, a peer becomes **idle** if no initial work is initially pushed into it (recall that we are assuming that only one peer owns the B&B work initially). From this point, every idle peer starts **requesting** for a work unit from its neighbors following a **round-based communication pattern**. As long as the rounds of requests are unsuccessful (that is, no interval has been obtained during a communication round), the peer keeps requesting work in subsequent rounds until a work is received or termination is detected. If a peer detects **termination**, that is it detects that no work is remaining in the whole network, it waits for its neighbors to claim termination as well before **ending** the execution of the algorithm. If a round of work request allows a peer to obtain a new work unit, then the peer starts **exploring** it. While a peer is exploring some work, it may receive requests from other neighboring peers and thus it has to serve them by **sharing** the locally available work. Once work is shared, the peer resumes the exploration of its interval and so on. Moreover, during its execution, a peer may either find or be informed about a new **best solution** found by another peer in the network. In this case, a peer pauses its current task, broadcasts the new solution to its neighbors and resumes its previous task.

Based on the state automaton of Figure 3.1, we derive a message passing distributed algorithm corresponding to our peer-to-peer approach. This is depicted in the high level code of Algorithm 1 (See Page 42), where the peer initialization phase is omitted for the sake of clarity. Generally speaking, the distributed algorithm is organized into two parts executed concurrently in parallel by *every* peer.

Through lines 1 to 26, we give the main set of instructions executed by a peer in order to coordinate the search process with its neighbors in a local manner. This is the core of our distributed P2P framework.

In lines 5 to 7, a peer performs the B&B algorithm on an interval it has previously received from a neighbor and, whenever it discovers a solution whose score is better

than the best solution found so far (stored into variable sol_v), it broadcasts it over the network through sending a New_Sol(sol_v) message to all its neighbors.

In lines 8 to 23, a peer performs a round of requests in order to obtain a new work unit or to claim termination. This mechanism is based on the use of a counter (stored in variable $count_v$). Its operation is described more in detail later. This counter is used to determine the distance between the current peer and the closest peer in the network holding a work unit. If a peer explores an interval, this counter is set to -1. Upon finishing its exploration, it is put back to 0. When requesting a new work unit, a peer first sends a Count_Request to its neighbors to retrieve the value of their respective counters. It waits for all the replies to be received. If a neighbor claims to have a work unit (it replied with a Count_Reply(-1) message), the current peer can send a Work_Request message to this neighbor to obtain a new work unit (received through a Work_Reply message). Note that it is useful to check the content of the received work unit. Indeed, between the moment when the neighbor receives the Count_Request message and when it receives the Work_Request message, it may have finished exploring its interval. If so, the current peer may resume its round of requests. The last part of a round of requests deals with synchronization. Indeed, as we evolve in an asynchronous environment, one has to make sure that during the Count_Request/Count_Reply procedure, the value of the retrieved counter remains coherent, that is they are not altered while the current peer performs its round of requests. Thus, we added a synchronization mechanism based on a *α-synchronizer* ([Awerbuch 1985],[Shabtay 1994]), using Safe messages. When a peer receives a Safe message from all its neighbors , it means that it can start a new round of requests and possibly change its counter without disturbing the operation of other peers. Note that when a neighbor replied with a Count_Reply(-1) message, it is exploring an interval and thus it is useless for the current peer to synchronize with this neighbor. Such neighbors are stored into the Q_v variable.

In lines 24 to 26, a peer synchronizes with its neighbors upon claiming termination. This part of the algorithm aims to keep the current peer active so it can process incoming requests from its neighbors until they detect termination too.

Through lines 28 to 42, we define the behavior of a peer upon receiving a message. These behaviors are quite simple as the goal is to send back a piece of information previously requested by a neighbor. Four different requests can be processed: a request for a counter (Count_Request), a request for a work unit (Work_Request), a new solution has been found by another peer (the solution is forwarded to other neighbors if it improves the local best solution) and a termination message (Terminated).

In the following sections, we describe the main issues tackled by Algorithm 1 and give the details of the different distributed techniques used there-in with respect to the B&B parallel computation.

3.3 Best solution and Work Sharing

Let us recall that we are assuming that B&B work is encoded following the interval-based approach of [Mezmaz 2005]. Hence, there are two main issues to handle when exploring the B&B tree intervals in parallel. First, each computational node should be assigned an interval to explore. To guarantee that the optimal solution will be effectively computed, we must ensure that an interval is assigned to at least one computational node. In addition, two intervals assigned to two computational nodes should not intersect to avoid redundancy and thus efficiency loss. Second, each computational node should be aware of the best solution found so far during the search process to avoid exploring unwanted branches, i.e., branches not leading to the optimal solution. The faster a computational node is aware of the best solution the more the B&B branching is efficient. In the following two subsections, we describe how these two issues are addressed in the framework of Algorithm 1.

3.3.1 Best solution sharing

During the B&B exploration process, a peer may compute a solution which is better than the best solution found so far. To make sure that other peers do not explore *unwanted* problems,(i.e. problems whose evaluation's lower bound is higher than the newly found best solution) it is necessary that the whole network becomes aware of this new solution. As we evolve in a fully distributed environment, this piece of information has to be broadcast among the peers. In Algorithm 1, the following simple rules are used:

- If a solution sol_v improving the previous best known solution is found locally by computational node v, then v sends a New_Sol(sol_v) message to its neighbors (line 7).
- If a node w receives a New_Sol(sol_u) message from neighbor u then it updates its current best solution sol_w. If sol_u is better than sol_w, then w also forwards New_Sol(sol_u) to its neighbors (line 40).

In Figure 3.2, we give illustration of a simple scenario involving 3 peers P_1, P_2 and P_3 connected through a path overlay: P_1 is a neighbor of P_2 and P_2 a neighbor of P_3. Let us assume that these peers are exploring (respectively) the following intervals : $[x, y], [x', y'], [x", y"]$? On step (1), peer P_1 discovers a new solution while exploring its interval. On step (2), peer P_1 informs peer P_2 about this solution by sending it a message. On step (3), as peer P_2 has been informed of a new improving solution found by another peer (here, P_1), it broadcasts it again to its neighbor P_3. Thanks to this process, peers P_2 and P_3 may be able to branch partly or entirely their respective intervals.

One may think that forwarding new solutions to neighbors could slow down the B&B search from two perspectives. First, the network may be flooded leading to congestion problems. Second, the time needed to forward the best found solution may be non negligible for an increasing number of computational nodes, hence

slowing down the B&B pruning process. We latter argue, in Section 3.5, that the above simple rules are sufficient to handle these two issues.

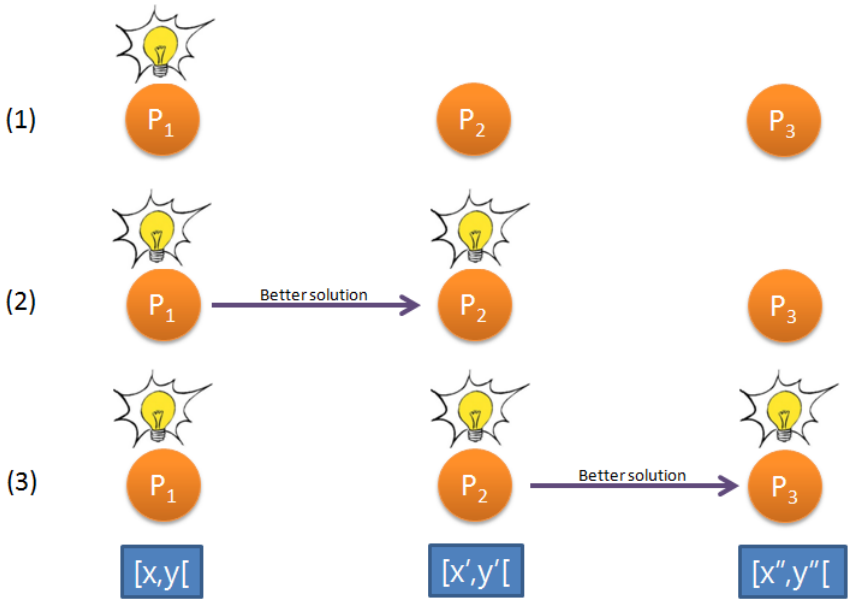

Figure 3.2: Scenario of best solution broadcasting.

3.3.2 Work Sharing

In Algorithm 1, a work unit corresponds to an interval $[x, y[$ where $x < y$, which encodes a B&B subtree as introduced in [Mezmaz 2005]. We recall that initially there exists one computational node in the network holding the interval $[0, N[$ encoding the whole problem to be solved. At any step of the algorithm, idle peers having no interval to explore ask their neighbors for a piece of interval. This is done by sending a Work_Request message (line 14). Upon receiving a Work_Request message, a node v holding an interval $[x, y[$ halts the local B&B search process and sends a Work_Reply($[x', y'[$) message answering the neighbor request (line 36). Here, subinterval $[x', y'[$ is chosen in such a way $[x', y'[\subset [x, y[$ and $[x', y'[$ has not been explored by node v yet. More precisely, if node v is exploring the branch corresponding to integer $z \in [x, y[$ when receiving a Work_Request message, then it replies with interval $[\frac{z+y}{2}, y[$[2] and continues its local exploration process using interval $[z, \frac{z+y}{2}[$. Notice that our work sharing mechanism guarantees that no kind of redundancy can occur during the search (an interval cannot be explored by two processors at the same time) which is to be contrasted with the Master-Slave approach of [Mezmaz 2005]. However, it may happen that at the time a computational node v sends a Work_Request message, none of its neighbors has an interval to share

[2] If many requests are received simultaneously from more than one neighbor, then node v could share its interval equally depending on the number of requests.

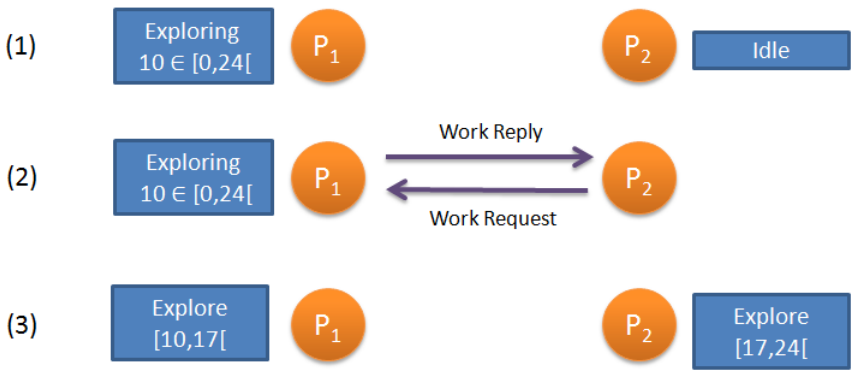

Figure 3.3: Scenario of work units sharing.

with him. For the sake of clarity, the discussion of this issue is delayed to later when addressing the termination detection of the B&B search process.

In Figure 3.3 represents a simple scenario with two neighboring peers P_1 and P_2, where peer P_1 is processing interval $[0, 24[$ and peer P_2 is idle. On steps (1) and (2), we consider the time where peer P_1 is exploring solution 10 in interval $[0, 24[$ while receiving a work request sent by peer P_1. As peer P_1 has already explored the interval $[0, 10[$, it would split the remaining part of its interval and thus share subinterval $[10, 24[$. Consequently, peer P_1 will send a work reply to P_2 containing the interval $[17, 24[$. We end up in Step (3) where peer P_2 begins exploring interval $[17, 24[$ and peer P_1 exploring interval $[10, 17[$.

Our work sharing policy can be compared to the steal-half work-stealing mechanism [Blumofe 1996] where work-stealing occurs *locally* among neighboring peers. Dividing an interval equally when receiving neighbor's requests may appear inefficient from a dynamic load balancing point of view. However, one should keep in mind that the length of an interval does not reflect how difficult it is to explore. In fact, we may have a long interval which will be explored very quickly by the B&B process, e.g., typically when the B&B pruning mechanism works very efficiently. Conversely, we may have a short interval that actually needs an extensive exploration and thus an increasing processing time. This is a well understood problem due to the irregularity of the B&B search tree. Thus, one cannot say in advance which interval is harder to explore than an other. In other words, different policies taking into account the length of intervals according to the power of nodes will fail to give strong guarantees on the load. However, we argue that the simple policy described above still enables a good dynamic load balancing. In fact, we can reasonably assume that the more a computational node is powerful, the more it will go fast when exploring an interval and the more it will send work requests to neighbors. Therefore, even if some powerful nodes are assigned some easy-to-explore intervals at a given time, they will perform fast and get more difficult intervals quickly. Nodes with powerful computational resources actually act like a black hole where work

intervals are spontaneously attracted.

3.4 Termination Detection

3.4.1 Basic concept

As pointed previously, a Work_Request message sent by node v could not always be satisfied since v's neighbors could not have any interval to explore. From a local point of view, all what v can learn at this point is that its neighbors (in the logical overlay) have finished exploring their own intervals. Nevertheless, v could not say yet whether (i) some other intervals are being explored by other nodes farther away in the overlay or (ii) all nodes have finished exploring their intervals. In the former case, we would like node v to get a sub-interval and enter again the exploration process. In the latter case, we would like node v to stop computing and to detect locally that the B&B exploration is terminated.

To handle this termination detection issue, we make use of the following key observation which reflects the local nature of our work sharing mechanism: If a node v sends a request message to its neighbors and none of them holds an interval, then this means that neighbors are symmetrically requesting a work interval from their own neighbors. Thus, a simple solution would be as follows: (i) every node having an interval being explored and receiving a request answers with a sub-interval as explained previously, and (ii) every other node requesting a work from neighbors waits until effectively receiving an interval from one neighbor. However, this solution will lead to a deadlock. To see this, consider for instance the situation where all nodes finish exploring their intervals simultaneously and hence no more intervals remain to explore. Then, all nodes will send request messages to their neighbors. Hence, no node will respond since the search process is actually terminated but no node can locally detect this global situation.

To correctly detect the global termination of the B&B search process, we use another more advanced technique. In fact, suppose that node v has made a first request to all its neighbors without receiving any interval. Then, node v resubmits its request. If node v does not receive an interval in this second round then node v concludes that its neighbors did not receive an interval during their first round. Thus, no node at distance two from v (in the overlay graph) owns an interval. Roughly speaking, by repeating the local request/response process t rounds, then a node can detect that no intervals are available in the ball of radius t around it. Therefore, after $\Theta(D)$ unsuccessful rounds[3] where D is the diameter of the overlay graph, a node can locally conclude that no interval is being explored by any node in the network and termination can thus be detected.

Remark 1. *At this point of the chapter, we recall that a large amount of literature is devoted to termination detection [Dijkstra 1980, Mittal 2004] in distributed*

[3] That is after $\Theta(D)$ local requests without getting an interval to explore.

systems. All existing termination detection techniques (wave-based, tree-based, snapshot, channel counting, etc.) have a cost [Mattern 1987]: delays and/or message overhead. Roughly speaking, distributed detection in our approach mimics a snapshot-based one (coupled with wave techniques) while using the computation proper messages. In fact, to detect termination we only have to detect that in a ball of radius D (the diameter) there remain no work intervals. Conversely, if in a ball B_t of radius $t < D$, the property $\mathcal{P}$='some work remains' holds, then termination cannot be announced. Counting (in a consistent manner) the number of times a node had performed unsuccessful work requests to neighbors is then sufficient to answer whether property $\mathcal{P}$ holds or not. In addition, this idea is both symmetric[4] *and topology independent (meaning for instance that when switching to a dynamic environment, there is a hope that it could still work under additional assumptions regarding overlay connectivity). For the sake of scalability, we thus use the basic messages of our algorithms, and we deliberately reduce the amount of required control/signal information and simplify at most the detection process to avoid congestion and delays. In other words, termination detection is carefully embedded in the work sharing mechanism. However, for our termination detection to work correctly, we need a knowledge about the diameter D of the overlay. Actually, even a rough upper bound will do correctly and efficiently. In Remark 2 in section 3.5, we argue why in practice the knowledge of D is not a requirement.*

3.4.2 Technical details

Technically speaking, we use an adaptation of the termination detection idea sketched in previous section in order to take into account the specificity of our B&B exploration process. More specifically, every node v owns a variable $count_v$ counting the number of times v asks its neighbors for an interval. This request is performed by sending Count_Request messages and receiving Count_Reply messages containing neighbors counters values. Initially, if node v is exploring an interval, $count_v$ is set to -1. If v does not hold any interval or has just ended exploring an interval, $count_v$ is set to 0 and a work request round is started (lines 2 and 21). An unsuccessful round, meaning that none of v's neighbors holds an interval, makes v increment $count_v$ by one and start a new request round until reaching $\Theta(D)$. However, since a node can possibly terminate exploring the whole piece of work interval before being able to share it with a neighbor[5], incrementing $count_v$ can lead to a situation where v increments its counter in an inconsistent manner, i.e., too fast. To illustrate this issue, let us consider the simple example of Figure 3.4 where the overlay is a simple path of four nodes[6]. Suppose that initially node v_0 owns an interval, i.e., all nodes but v_0 have their counters set to 0. After a first work request

[4] It is initiated symmetrically by all nodes

[5] For instance, this may happen when a request message takes a long time to be delivered

[6] We choose a line topology for this example for only a sake of simplicity. The reasoning can be applied for *any* overlay topology. In particular, the path of the example can be thought to be a subgraph of a more general overlay. An example with a grid topology is given in Appendix B to summarize our work sharing algorithms.

round, only node v_1 gets an interval. Nodes v_2 and v_3 increment their counters to 1. Suppose that during the second work request round, node v_1 terminates exploring its interval before receiving the request of its neighbor v_2. We term this situation as an *"Interval Vaporization"*. Then, after the second request round, node v_2 does not receive any work interval. Thus, if node v_2 increments its counter, as depicted in Figure 3.4, then after the third request round, nodes v_2 and v_3 will set their counters to 3 reaching the diameter of overlay path. This interval vaporization situation can then occur further in future rounds as we can imagine that the intervals that node v_1 gets from node v_0 are always explored/prunned very quickly.

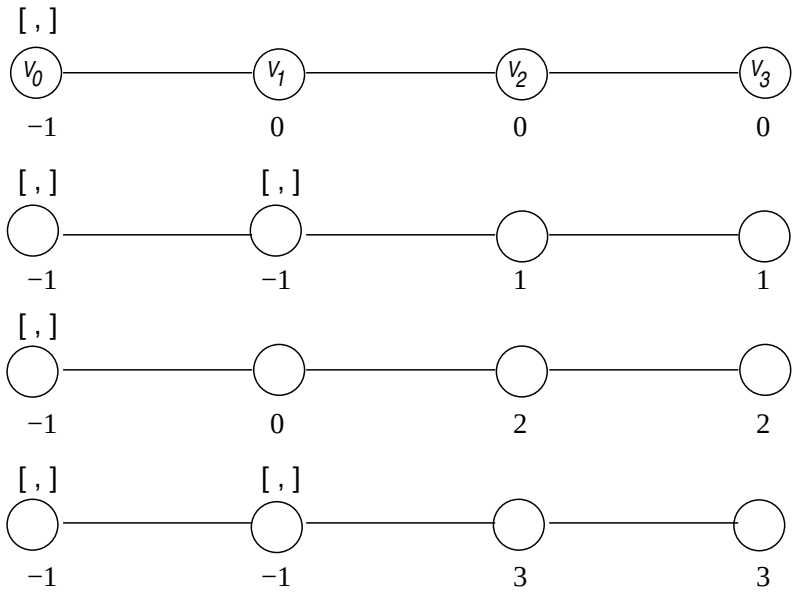

Figure 3.4: Interval Vaporization

When an interval vaporization occurs at some node v, the counter of v is first set to -1 then to 0. Thus, a neighbor u of v, having a counter set to some value $x > 0$ in previous rounds, can detect that actually an interval vaporization has occurred at node v. Therefore, when detecting such a situation, node u does not increase its counter until v's counter reaches value x. This rule is implemented by lines 39 and 40 using the received Count_Reply(*count*) messages.

3.4.3 Asynchrony issues

The previous work sharing and termination detection mechanisms are based on request/response *rounds*. At each round, every node asks for neighbors' counters and possibly asks for some work interval if any. This clearly implies a kind of synchronism between nodes in order to perform efficiently. However, in a fully *asynchronous* distributed environment, we cannot have any guarantees about the time it takes to deliver a message from a node to its neighbors, i.e., link latencies could be arbitrary but finite. To illustrate these issues, let us consider the simple

example of Figure 3.5 given below.

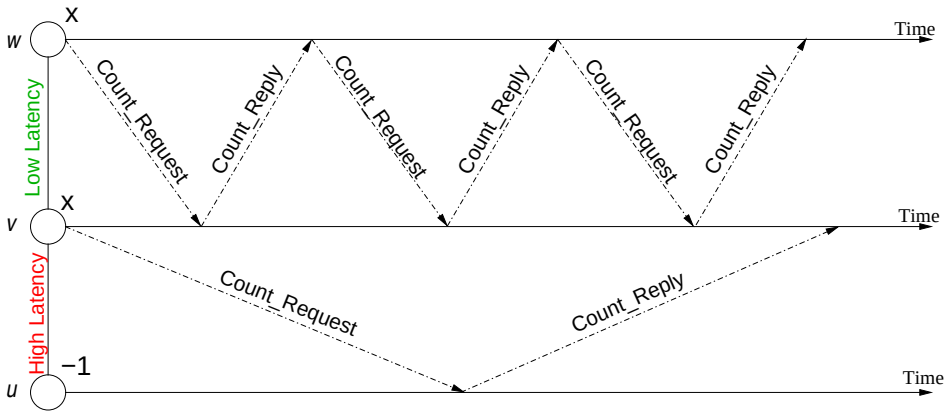

Figure 3.5: Synchronization issues

Assume that we have three nodes u, v and w connected by a path somewhere in the network overlay. Assume that the network latency is high between v and u whereas it is low between v and w. Suppose that node v and w have set their counters to x while node u has executed a successful round and thus has a counter set to -1. Then, the request message sent from v to u will take much time, while node w will by the meanwhile perform a large number of requests. These requests are obviously useless as node v has not yet received the piece of information that can make it update its counter.

In addition, we remark that due to network latencies, whenever a piece of information (e.g., a counter value) is exchanged between two nodes, this information may become incoherent with the sender status before it is actually received. For instance, using the example of Figure 3.5, one can imagine the situation where node v performs a request round by sending Count_Request messages to its neighbors, including node w. Node w shall receive this message and send back a Count_Reply($count_w = x$) message to v. Meanwhile, node w may have symmetrically done the same process. Thus, during the time the Count_Reply(x) message is transmitted from node w to v, node w may set its counter to -1 (if it gets a work from another neighbor), 0 (if it detects an interval vaporization), or $x + 1$.

To avoid these situations and let nodes operate correctly and efficiently under unknown network latencies[7], we use an adaptation of a well known technique, called α-synchronizer [Awerbuch 1985, Moran 2000, Shabtay 1994]. This technique allows us to abstract away the problems induced by network latencies and to efficiently coordinate request rounds. Generally speaking, each time a work request round is started by some node v, node v performs the following local actions. First, it waits for neighbors responses, i.e., Count_Reply messages. This allows node v to detect that its requests were actually received by neighbors and to receive neighbors

[7] Actually, in our distributed algorithm, we only assume that messages are received in the same order they are sent, i.e., asynchronous FIFO model.

counters. If node v detects that some neighbor owns an interval then it asks to share it, i.e., Work_Request and Work_Reply messages. After finishing this work request round, a node then sends a Safe message to neighbors (line 20) and symmetrically waits for their Safe messages before starting a new round (line 21) and so on until termination is detected or a piece of interval is received. Notice that nodes being exploring some intervals do not send any Safe messages (line 12). Symmetrically, a node that has detected that a neighbor is exploring an interval (by receiving a Count_Reply(-1) message) does not send any Safe message to it (line 20). The same reasoning is applied for a node that claims termination (lines 24-26).

3.5 Complexity issues and overlay properties

In this section, we give a complexity analysis of the distributed mechanisms designed for our P2P Branch-and-Bound approach. We first remark that since our distributed algorithms were designed to work under a fully asynchronous environment and due to irregularity/non-determinism of the B&B tree exploration process, it is very difficult to give a rigorous complexity analysis of our P2P approach. To our best, no previous studies have neither been able to give any theoretical analysis of distributed B&B algorithms, mainly due to the hardness of mathematically predicting how the pruning process can act in function of the tackled instance. Therefore, the goal of this section is to highlight the main complexity issues of our approach while stressing on the scalability of the overlay and the congestion that may be created at the network level. For that purpose, we shall make the following assumptions:

- A *message* is an infrangible piece of information which is sent along a logical link between two nodes (peers) in the overlay, independently from the number or the nature of physical equipments required to transmit it over the physical network.
- During one *time unit*, a node in the network may send to each neighbor at most one message. Equivalently, this means that a message sending induces a delay of one time unit. We emphasis on the fact that time units are introduced for the sake of analysis only. In other words, our algorithms do not assume any shared global clock nor any known bound on the time needed to transmit a message over the network.

Let us recall that D denotes the diameter of the network overlay $G = (V, E)$ connecting n computing peers and d_v denotes the degree of peer $v \in V$. From a message complexity point of view, the two following properties can be claimed:

- Whenever a new best solution is found by a node during the B&B local search process, it costs at most $O(|E|)$ messages until all other nodes in the network update their local solutions. In addition, assuming that sending a message to neighbors costs one time unit, it takes at most D time units to update the local solutions of all nodes.

- Consider that, at a given time, a node v has no work interval to explore. Assume that *no interval vaporization occurs* during any work request round. In the worst case, node v sends at most $O(d_v \cdot D)$ messages within at most $O(D)$ time units until either it receives a work unit or it detects the termination of the B&B algorithm.

Since our approach can properly operate whatever the logical network topology is, it becomes clear from the previous analysis that it requires specific properties for the overlay graph to perform efficiently. Before pushing the discussion further, let us remark that when tackling large instance problems, the B&B process is initialized with some good solution (computed for instance using a given heuristic[8]). Thus, it is rather rare that many new solutions are found by many peers at the same time. Equivalently, most of the time peers just prune the explored intervals without finding new solutions. Therefore, we can reasonably state that the communication complexity of our approach is dominated by the work sharing mechanism. Thus, to gain in efficiency, the overlay graph should have a good balance between node degrees (i.e., each peer must have a low number of neighbors) and network diameter (i.e., peers must stay relatively close in the logical overlay). In fact, spreading a piece of information (intervals) over the network must be performed quickly while keeping the communication load of the peers at a low level to avoid bottlenecks around peers.

To satisfy these constraints, one may use several kinds of overlay constructions. Let us consider a toric hypercube overlay[9] constructed in the following way.

Considering N vertices in the graph, the dimension of the hypercube is set to $\lceil \log(N) \rceil$ i.e. the toric hypercube is defined as a $\lceil \log(N) \rceil$-dimensional lattice. Figure 3.6 provides an example of a toric 2-dimensional 5x5 lattice. Peers a_1 to a_5 are linked to peers e_1 to e_5. Peers a_1 to e_1 are linked to peers a_5 to e_5.

More precisely, to generate the overlay topology, we proceed the following way. Let N be the number of peers in the network. Two values are calculated: $k = \lceil \log(N) \rceil$ and $m = \lceil N^{\frac{1}{\lceil \log N \rceil}} \rceil$. Those two values correspond to the number of dimensions and the length of the edges of the hypercube. Consider a peer whose identifier is $p \in [0, N-1]$.

For a given integer b, we define p_b the representation of the number p in base b: $p_b = (x_c, x_{c-1}, \ldots, x_2, x_1)$ with $c = \lceil \log_b(p) \rceil$ and $\sum_{i=1}^{c} (x_i \cdot b^{i-1}) = p$ and $\forall i \in [1, c], x_i \in [0, b-1]$.

Still considering the peer identified by p, we consider its representation in base m : $p_m = (x_k, x_{k-1}, \ldots, x_2, x_1)$.

$\forall u \in [1, k]$, we define the following numbers in base m:

$$q_m^u = (x_k, x_{k-1}, \ldots, f^+(x_u), \ldots, x_2, x_1)$$

[8] It is generally admitted that starting a B&B from scratch is rather very costly.

[9] The implementation and deployment in Grid'5000 of the overlay is presented in Section 5.2.3.

and

$$r_m^u = (x_k, x_{k-1}, \ldots, f^-(x_u), \ldots, x_2, x_1)$$

where

$$f^+(x_u) = \begin{cases} x_u + 1 \mod m, & if q_{10} = q \leq N; \\ 0, & else. \end{cases}$$

and

$$f^-(x_u) = x_u - \alpha_u \mod m$$

where α_u is the smallest strictly positive integer such that $r_{10} = r \leq N$. When converted back into base 10, $q_{10} = q$ and $r_{10} = r$ represent the identifiers of two other peers that are neighbors of the current peer p.

Note that, to ensure that we do not miss any neighbors during the procedure, the number of digits (the number of x_i elements in p_m) has to be set to k. Indeed, consider the peer whose identifier is set to 0, p_m would have only one digit x_1 and we would forget considering neighbors defined along the $k-1$ other dimensions of the hypercube ! This process is applied to all the N peers.

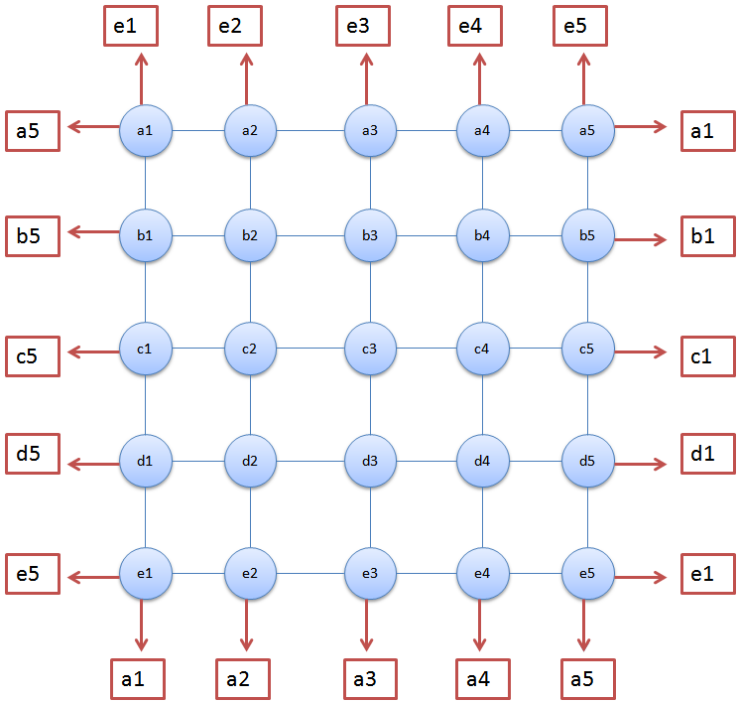

Figure 3.6: Illustration of a 2-dimensional toric lattice whose edges length is 5.

Having this overlay constructed, one can show that the diameter of the underlying graph and its average degree is order of $\log(n)$ with n being the number of peers. Therefore, it holds from the previous discussion that the combined work sharing and termination detection in our approach running over a hypercube overlay costs roughly $O(\log(n)^2)$ messages per node. To illustrate how this can improve the scalability of the Master-Slave approach, let us assume that the limitation of the

Master-Slave approach is only due to the bottleneck created around the master, that is to the number of slaves or equivalently to the degree of the star overlay induced by such an architecture. Let n be the maximum number of slaves that can be handled simultaneously by the Master, i.e., n is now the maximum number of Slaves that leads to a bottleneck around the Master. Then, we can reasonably assume that n would be also the maximum number of neighbors before a communication bottleneck around a given peer can be observed. Thus, considering a P2P overlay with size $n' = \Theta(\exp(n))$, we obtain a mean node degree of $\Theta(\log n') = \Theta(n)$. This means that while the Master-Worker model can handle only up to n computational nodes before breaking down, the P2P approach scales up to $\Theta(\exp(n))$ nodes. Of course, exponentially more computational resources does not straightforwardly mean exponential gain in load balancing/parallel efficiency nor in search speed-up, since handling more resources has a communication cost. In the Master-Slave approach, in order to share work or to announce termination, the Master sends $O(1)$ messages to each Slave. Overall, the number of messages sent is thus $\Theta(n)$ with the Master having a degree equal to n. Assume now that it can be proven that $\Theta(n)$ is the maximum number of messages that can be sent simultaneously over the network before network delays/congestion can lead to the system breaking down. Thus, under the assumption that no work vaporization occurs which is a reasonable assumption for large scale and difficult instances, our P2P approach can still handle up to $n' = \Theta(n/\log^2(n))$ peers which is asymptotically better than $\Theta(n^{1-\varepsilon})$ for an *arbitrary small* positive constant $\varepsilon < 1$. Thus, overall, we obtain a better balance between scalability in terms of number of peers and scalability in terms of communication cost that is in Parallel Efficiency. As our experimental results show in chapter 5, we even obtain lower communication overheads than expected theoretically while being more efficient in terms of space exploration and in parallel efficiency.

Remark 2. *Notice that for the sake of termination detection, we have assumed that the diameter D of the graph is known. This may appear as a strong and penalizing assumption. However, we argue that an upper bound on D is sufficient. In addition, using an upper bound instead of the exact value of D does not affect the communication cost as long as some work interval remains somewhere in the network, i.e., message overhead will only occur when the search space is fully explored. According to the previous discussion, the overlay graph should be chosen such that $D = \Theta(\log n)$. Thus, in practice, for n around, let us say, 10^{10}, even an over estimated upper bound of, let us say,* 100 *is perfectly plausible.*

3.6 Conclusion

In this chapter, we presented our P2P-B&B approach for parallelizing the Branch-and-Bound algorithm in a large-scale distributed environments. Our approach does not undergo any centralized mechanism. From a logical point of view, computational nodes are peers that can communicate with only a subset of other computational nodes called neighbors and the network overlay can be any connected graph

$G = (V, E)$. On the algorithmic side, a peer is in charge of managing its own local work pool, to share its work units with other peers through the use of the Work Sharing Mechanism defined in Section 3.3.2, to broadcast any new improving solution discovered during the exploration process to other peers through the Best Solution Sharing Mechanism described in Section 3.3.1 and detect the termination of the calculation in a distributed way based on a local vision of the network thanks to the Termination detection mechanism defined in Section 3.4. We also provided a complexity analysis of our approach in terms of time and messages showing how the choice of the logical overlay connecting peers could impact the overall performances. The latter analysis is however conducted assuming a simplified model of communication and B&B search process. In Chapter 5, we shall provide a throughout experimental study of the performances of our approach as well as its scalability when effectively implemented and deployed over a real large-scale experimental grid.

One should remark that the informal description given in this chapter allows one to have a general idea of our approach; however, it does not guarantee the correctness of the designed distributed protocols. Therefore, and before turning to an experimental evaluation of our approach, the next chapter of this thesis shall provide a formal analysis of our approach.

Algorithm 1: General overview of our $B\&B$ distributed algorithm

1 **High level code executed by node** v **(in parallel with receive triggers below):**
2 **while** $count_v < X$ **do**
3 ;
4 $increment_v := true$; $ready_v := true$; $Q_v := \varnothing$;
5 **while** $[x, y[\neq \varnothing$ **do**
6 Explore $[x, y[$ using the B&B algorithm;
7 **As soon as** a better solution sol_v is found **Send** New(sol_v) to all neighbors;
8 **if** $count_v = -1$ **then** $count_v := 0$;
9 **Send** Request **to all neighbors**;
10 **Wait Upon Reception of** *Reply*($count_w$) **from every neighbor** w **:**
11 **if** $count_w = -1$ **then**
12 $increment_v := false$; $count_v := 0$; $Q_v := Q_v \cup \{w\}$;
13 **if** $[x, y[= \varnothing$ **then**
14 **Send** Work_Request **to** w;
15 **Upon Reception of** *Work_Reply*($[z', y'[$) **from a neighbor** w **do :**
16 **if** $[z', y'[\neq \varnothing$ **then** $[x, y[:= [z', y'[$;
17 **if** $count_w \geq X$ **then** $Q_v := Q_v \cup \{w\}$;
18 $ready_v := false$;
19 **if** $[x, y[\neq \varnothing$ **then** $count_v := -1$;
20 **Send** Safe($count_v$) **to every neighbor** $w \notin Q_v$;
21 **Wait Upon Reception of** *Safe*($count_w$) **from every neighbor** $w \notin Q_v$ **:**
22 **if** $(count_w < count_v)$ **then** $increment_v := false$;
23 **if** $increment_v$ **then** $count_v := count_v + 1$;
24 **Send** Terminated **to all neighbors**;
25 $ready_v := true$;
26 **Wait Upon Reception of** Terminated **from all neighbors**;
27 **High level code for receive triggers executed by node** v **(in parallel):**
28 **Upon Reception of** *Request* **from a neighbor** w **do :**
29 **if** $ready_v \wedge w \notin Q_v$ **then Send** Reply($count_v$) **to** w;
30 **else**
31 `/* Inqueue the Request and delay the response (v waits until it is ready for the next round) */`
32 **Send** Reply($count_v$) **to** w **later as soon as** $ready_v \wedge w \notin Q_v$ **is verified**;
33 **Upon Reception of** *Work_Request* **from a neighbor** w **do :**
34 **if** $[x, y[$ *is not yet entirely explored* **then**
35 Let $z \in [x, y[$ be the integer corresponding to the solution being explored last;
36 $[x, y[:= [z, \frac{z+y}{2}[$; **Send** Work_Reply($[\frac{z+y}{2}, y[$) **to** w;
37 **else Send** Work($\varnothing$) **to** w;
38 **Upon Reception of** *New*(sol_w) **from a neighbor** w **do :**
39 **if** sol_w *is better than* sol_v **then**
40 $sol_v := sol_w$; **Send** New(sol_v) **to all neighbors** $w' \neq w$;
41 **Upon Reception of** *Terminated* **from a neighbor** w **do :**
42 **if** $ready_v = false$ **then** $Q_v := Q_v \bigcup \{w\}$;

Chapter 4

Correctness and Complexity Analysis of B&B-P2P

Contents

4.1 Introduction

The goal of this chapter is to provide a formal analysis of the correctness of our fully distributed approach. In fact, the work-sharing, solution exchange, and termination distributed protocols embedded in our approach are inter-dependent which may lead to hidden and unpredictable deadlocks issues. In order to guarantee that such issue cannot occur in our P2P approach, we provide a complete correctness analysis that guarantees several properties of our distributed protocol. More precisely, we shall state different lemmas proving the following main properties: (i) there is no redundancy during the exploration process (that is, the P2P network processes the tasks to be performed in a finite time), (ii) there is no deadlock in the communication protocol (that is, two peers are blocked waiting for a message from each other), (iii) a peer cannot claim termination if there still exist a peer in the network which is processing a task and (iv) once the last task is processed, a peer can detect and claim termination within a finite time and. Besides proving the correctness of our approach, the analysis we conduct provides insights into the properties of our approach and highlights the challenging issues we had to deal with in designing and deploying our approach.

4.2 Correctness Analysis (Termination Detection)

4.2.1 Overview of the proof

In the following, we give a step by step proof assessing the correctness of Algorithm 1, depicted and described in the previous chapter. We consider the whole execution process and distinguish between two phases. The first phase starts with peers deployed and beginning the computations and ends when the very last B&B solution is explored by some peer. The second phase starts from the time the last B&B solution was explored, i.e., no work remains in the system and ends with all peers having detected that indeed, the computation is finished. Notice that from the peer local view, no information is available about those two phases. In other words, these phases are introduced only for the sake of analysis. During those two phases, we proceed by proving the following crucial properties:

- During Phase 1, one has to prove that:
  - All the solutions will be explored within a finite amount of time, i.e. no solution can be explored at least twice (**Lemma 1**) and thus Phase 2 can effectively start.
  - A peer synchronizes its rounds of requests with its neighbors, to avoid continuously asking for the same piece of information repeatedly and uselessly. This is to make sure that the information being communicated remains identical and coherent during the communication process. (**Lemma 2** and **Lemma 3**)
  - A peer cannot detect termination while another peer is still exploring solutions. (**Lemma 4**)
- Once Phase 2 starts:
  - Every peer may detect termination within a finite amount of time. (**Lemma 6**)
- During both phases:
  - Any round of requests is achieved within a finite amount of time. (**Lemma 5**)

Figure 4.1 represents the logical relationships between the properties that are to be proved. We start with the basic observation that no redundancy can occur in Algorithm 1. From the algorithm pseudo-code, a node shares its interval with at most one neighbor, i.e., Work messages. Whenever an interval is shared, it is split into disjoint pieces. Thus, the following is straightforward.

Lemma 1. *Algorithm 1 guarantees that two peers never explore two intersecting intervals, i.e., there is no redundancy.*

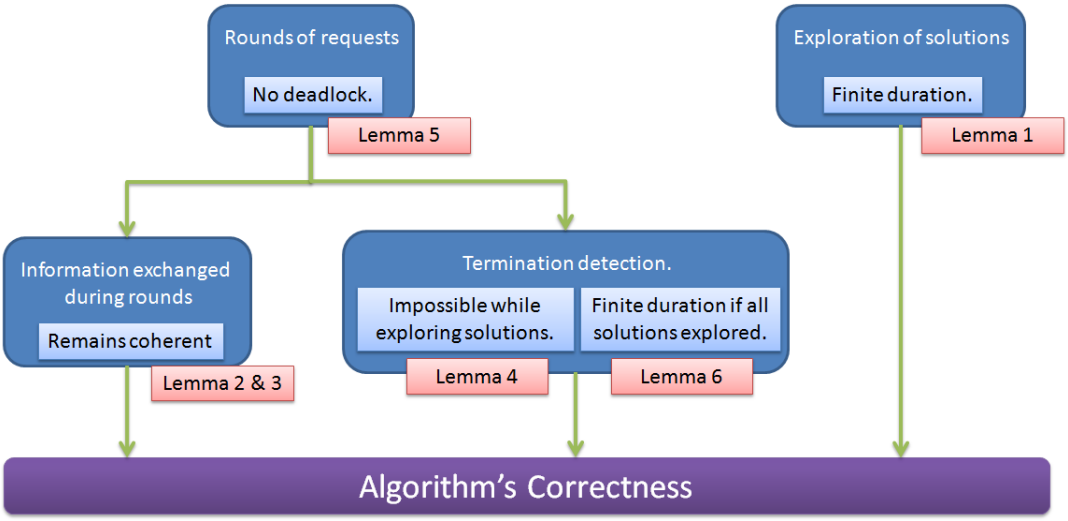

Figure 4.1: Visual representation of precedence constraints between the properties to be proved.

4.2.2 Properties of communication rounds

Given a node v executing Algorithm 1, we denote by $count_v(t)$ the counter of v at time t, where a global an unique time clock is assumed for all peers. We recall that the counter of a peer v is used in order to handle communication round local synchronization and termination detection. Hence, it is crucial in proving the correct behavior of our protocols. Suppose that $count_v(t) = x$ for some integer x verifying $x \geq 0$ and at some fixed time point $t \geq 0$. From the algorithm description, it is clear that v must have successively incremented its counter starting from 0 until reaching x. We then denote by $t_i(v)$ the time when the counter of node v reached value $i \in \{0, 1, 2, \cdots, x-1, x\}$. More precisely, for $i \in \{1, \ldots, x\}$, $t_i(v)$ is the time node v incremented its counter by 1 in line 23 of algorithm 1. For $i = 0$, $t_0(v)$ is defined as the last time node v sets its counter to 0 and begins a new Request/Reply/Safe communication round. We remark that node v may end such a round *without necessarily* incrementing its counter on line 23. Thus, it may perform $y \geq x$ rounds before being able to increment its counter up to x. Thus, let us denote $t'_j(v)$ the time when node v executed the instruction on line 23 for the j^{th} time on its way for incrementing its counter from 0 up to x with $j \in \{0, 1, 2, \cdots, y-1, y\}$. Hence, we have $t_x(v) = t'_y(v)$, $t_0(v) = t'_0(v)$ and $\forall i \in [1, x-1], \exists j \in [1, y-1] | t_i(v) = t'_j(v)$. Figure 4.2 represents a simple example illustrating the so-defined time sequences with $x = 4, y = 9$.

Note that both defined time sequences $t_i(v)$ and $t'_j(v)$ depend on the considered time point t. For simplicity and clarity, we have omitted to mention this fact in our notations.

Lemma 2. *Consider a node v and an integer $x \geq 1$. Suppose that there exist two time points $t'_j(v) < t'_{j+1}(v), j \in \mathbb{N}$ such that node v sets his counter successively to*

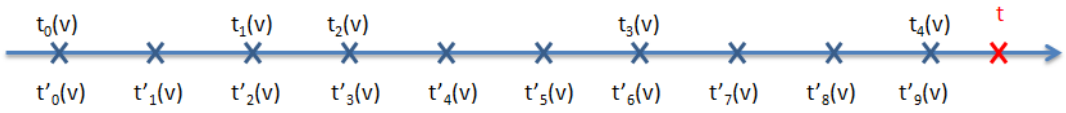

Figure 4.2: Example of definition of the time sequences used in lemmas' proofs.

x at time $t'_j(v)$ then to $\alpha_x \in \{x, x+1\}$ at time $t'_{j+1}(v)$. Then, for every neighbor $u \in N(v)$, there exist time points $\overrightarrow{s}_u$ and $\overleftarrow{s}_v$ verifying:

- $t'_j(v) < \overrightarrow{s}_u < \overleftarrow{s}_v < t'_{j+1}(v)$.
- *At time $\overrightarrow{s}_u$, node u sends a message $\mathsf{Safe}(y)$ to v with $y \geq 0$.*
- *At time $\overleftarrow{s}_v$, node v receives and processes u's $\mathsf{Safe}(y)$ message.*

Proof. In order to be able to set its counter from x to α_x, node v must have executed a Request/Reply/Safe communication round. In particular, v waits for Safe messages from all its neighbors $w \notin Q_v$ in line 21. Consider a node $u \in N(v)$ and suppose that $u \in Q_v$. From the algorithm description, this means that u replied to v's request by a $\mathsf{Reply}(-1)$ message. Thus, v would have set its counter down to 0 in line 12. Thus, v cannot set its counter to α_x in line 23: contradiction. Therefore, $u \notin Q_v$ meaning that u must have replied with a $\mathsf{Reply}(\geq 0)$ message to v's Request message. Similarly, by line 21 of the algorithm, it is clear that v must have received message $\mathsf{Safe}(\geq 0)$ from u before setting its counter to α_x.

Now, let us consider the time t when node u receives and process v's request in line 28. We distinguish two cases:

1. Suppose that u does not delay v's Request. This means that $ready_u = true$ and $v \notin Q_u$ at time t. By the previous discussion u replies with a counter value at least 0 (otherwise v could not have set its counter to α_x). Thus at time t, node u is already executing a Request/Reply/Safe communication round. In particular, node u has already sent a Request message to v. Let t' be the time corresponding to u starting that communication round. Let t'' be the time by which u's Request arrived at node v. Note that from time t' to time t node u could not have sent a Safe message yet: Otherwise, it would have set its $ready_u$ variable to $false$ and would have delayed v's request leading to a contradiction. We have two cases:

   - Suppose that u's request is received after time $t'_j(v)$, i.e., $t'' > t'_j(v)$ (See Figure 4.3). Thus, v's Reply is sent after time $t'_j(v)$ (after v's request was sent) and thus it arrives to u after time t (recall that we assumed a FIFO message passing model). Since a node is blocked after a request waiting for the corresponding response, u would send a Safe message to v only at time $\overrightarrow{s_u}$ verifying $\overrightarrow{s_u} > t > t'_j(v)$.

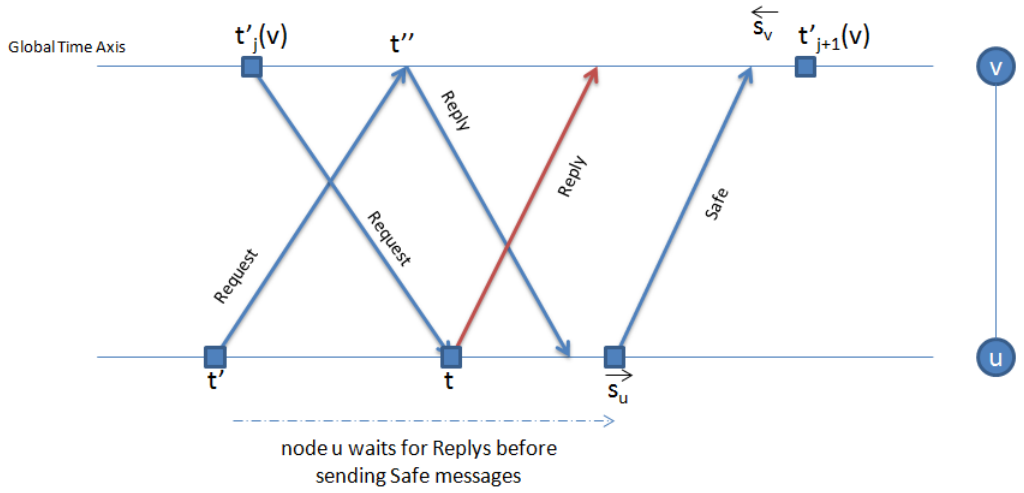

Figure 4.3: The case where u's request arrives after time $t'_j(v)$ in the proof of Lemma 2

- Suppose that u's request is received before time $t'_j(v)$, i.e., $t'' < t'_j(v)$. Assume for contradiction that $count_v(t'') = -1$, i.e., node v still holds an interval by time t''. Thus, node v would have replied with a Reply(-1) message. Thus, node u would have added node v in the queue Q_u and would have never replied to v by time t which is a contradiction (recall that by definition of t', node u is still executing a *unique* communication round from time t' and t). Suppose now that $count_v(t'') \geq 0$. This means that v has completed a Request/Reply/Safe communication round before time $t'_j(v)$. Let t_1 be the time node v begins executing this communication round. Let t_2 be the time node v sent its Safe messages (equivalently, completed its Request/Reply process). Again, we have to distinguish between two cases.
  - $t'' > t_2$ (See Figure 4.4). In this case, u's Request message is delayed by v. Hence, until time t, node u does not get any Reply message from v (i.e., node u is blocked waiting for v's Reply). Thus, node u can send a Safe message to v only by time $\overrightarrow{s_u} > t > t'_j(v)$.
  - $t_1 < t'' < t_2$. Here we distinguish two cases again. First, suppose that v replied immediately to u (i.e., $u \notin Q_v$ at time t''). Thus, v must have received a Safe message from u to complete its communication round. Thus, u must have sent a Safe message to v in time period $[t', t]$: contradiction. Second, suppose that v does not reply to u's request and delay its reply (i.e., $u \in Q_v$ at time t''). Thus, node u will be blocked waiting for v response at least until time t and thus it can send a Safe message to v only by time $\overrightarrow{s_u} > t > t'_j(v)$.

2. Now, we consider the case where u delays v's Request when executing the

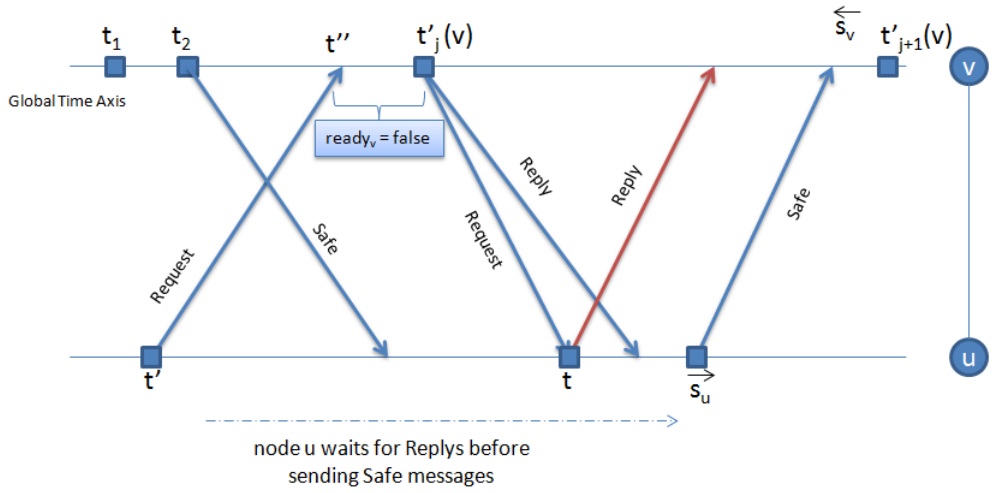

Figure 4.4: The case where u's request arrives after time t_2 in the proof of Lemma 2

receive trigger of line 28 at time t. Denote by $t' > t$ the time u sends its Reply message (See Figure 4.5). From the algorithm description, v's request can be delayed only if at time t, either $v \in Q_u$ or $ready_u = false$ (i.e., u is waiting for Safe messages). In the two cases, u's Reply(≥ 0) message is sent just after time $t_1 > t$ corresponding to u beginning a new Request/Reply/Safe communication round. Thus, the Request message sent by u at time t_1 arrives at v after time $t > t'_j(v)$. At that time v is still blocked waiting for u's Reply message and thus v can only respond with a Reply($x \geq 0$) message. Hence, u does send a Safe(≥ 0) message to v after time $t > t'_j(v)$.

To summarize, we have that if node u replied to v's Request by a Reply(≥ 0) message then it does send a Safe(≥ 0) message to v after receiving that request.

Now, suppose we have the following hypothesis:

(**H**) : At each time node v starts a new Request/Reply/Safe communication round, there are no Safe messages pending/delayed at v, i.e., whenever a Safe message is sent to a neighbor v, that neighbor actually processes that message before its next communication/exploration round.

Assuming (**H**), it becomes clear from the previous discussion and the algorithm description that before being able to set its counter from x to $\alpha_x \in \{x, x+1\}$, node v must wait for the Safe($y \geq 0$) message sent by u at time $\overrightarrow{s_u}$ defined implicitly from the previous discussion. Thus, assuming (**H**) we have that node v cannot set its counter to α before receiving and processing u's Safe(y) message at some time $\overleftarrow{s_v}$ verifying $t'_j(v) < \overrightarrow{s_u} < \overleftarrow{s_v} < t'_{j+1}(v)$.

Proving that hypothesis (**H**) is verified for every node is done by induction. In the following, we just give a sketch of how the proof goes. In fact, we can view the execution of our distributed algorithm at a given node v as a sequence of time periods

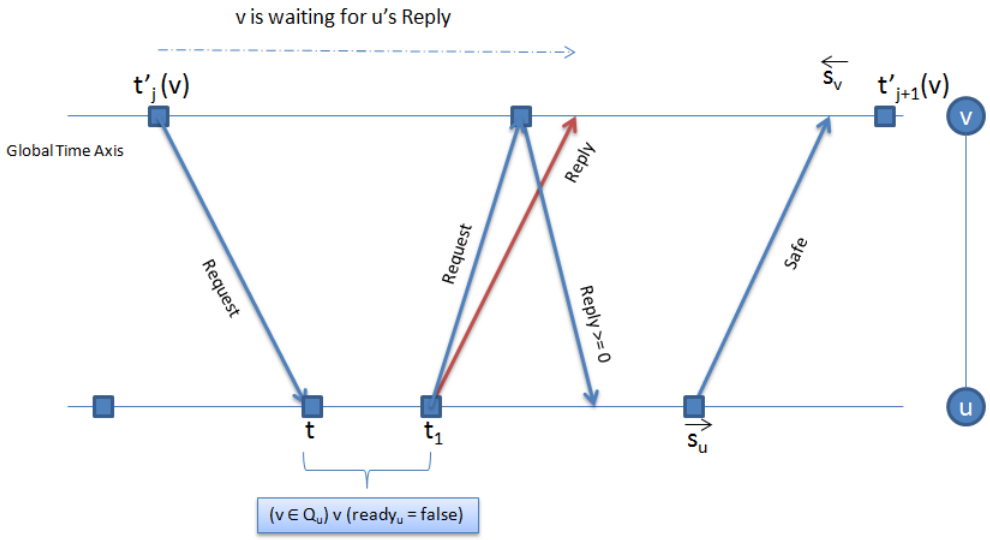

Figure 4.5: The case where node u delays v's Request message in the proof of Lemma 2

T'_i where T'_i is either a Request/Reply/Safe communication round or a time period of work interval exploration (corresponding to node v exploring a B&B interval in line 6). At time 0, the hypothesis (**H**) is verified. Thus, if at the very first sequence T'_0, node v is initially exploring an interval, the hypothesis stays true all along T'_0 since from the algorithm description no neighbor u of v can send a Safe message to v. Hence, the hypothesis stays true at the second sequence T'_1. Now, suppose that the first sequence T'_0 corresponds to a Request/Reply/Safe communication round (that is node v has no interval to explore initially). Consider a neighbor u of and suppose that u holds *no* work interval initially. Then, neither node v nor node u can switch to a new sequence T'_1 before receiving a Safe message. That message is then processed before switching to a new sequence T'_1. Suppose now that initially node u holds a work interval. If v's request arrives at u while u still holds a piece of interval to explore then no Safe message is sent by to v by u. Otherwise, if u has no interval to explore at the time it receives v's request, then by the previous discussion, it is also clear that a Safe message is sent by u to v and processed by v before switching to a new sequence T'_1. By the same arguments, and by induction on the sequences of communication/exploration periods T'_i, it is not difficult to see that hypothesis (**H**) is always verified throughout the algorithm execution. This concludes our proof. □

Lemma 3. *For every integer $x \geq 2$, at any time $t \geq 0$, and for every node $v \in V$ such that $count_v(t) = x$, it holds that for every neighbor $u \in N(v)$, there exists a sequence of $x-1$ time points $S_x^{v,u} = \{t'_1(u), t'_2(u), \cdots, t'_{x-1}(u)\}$ verifying:*

- $\forall i \in \{1, \ldots, x-1\}$, $t'_{i-1}(v) < t'_i(u) < t'_{i+1}(v)$ *and* $t'_{i-1}(u) < t'_i(u)$

- $\forall i \in \{2, \ldots, x-1\}$, $\forall t' \in [t'_{i-1}(u), t'_i(u)[$, $count_u(t') \geq i-1$
- $count_u(t'_{x-1}(u)) \geq x-1$

Proof. In the following, we give a constructive and inductive proof that defines precisely the sequence of time points t'_i throughout the algorithm execution. We recall that for v to set its counter from $y \geq 0$ to $\alpha_y \in \{y, y+1\}$ during time period $[t'_j(v), t'_{j+1}(v)[$ for $j \in \{0, \ldots, x-1\}$, node v must (at least) do the following actions : send a Request message to u then receives a Reply message from u, and send a Safe message[1] to u and symmetrically receive a Safe message from u.

From the algorithm description, it is clear that node v must have performed at least x successive rounds to set its counter up to x. Thus, if we name $\{t'_0(v), \ldots, t'_{x-1}(v)\}$ the last x dates when v started a Request/Reply/Safe communication round and $t'_x(v)$ the time when it ended the last round and set its counter to x, it is clear that $\forall i \in [0, x-1], \forall t' \in [t'_i(v), t'_{i+1}(v)[, count_v(t') \geq i$.

From the algorithm description and by lemma 2, it holds that for v to perform the rounds taking place during $[t'_{x-2}(v), t'_{x-1}(v)[$ and $[t'_{x-1}(v), t'_x(v)[$. Let $\overrightarrow{s}_{u,x-2}$ (resp. $\overrightarrow{s}_{u,x-1}$) be the time by which u sends its safe message to v. From the algorithm description, node u must have executed a round of Request/Reply with neighbors in time period $[\overrightarrow{s}_{u,x-2}, \overrightarrow{s}_{u,x-1}[$. In fact, after time $\overrightarrow{s}_{u,x-2}$, node u makes the following actions: (1) it waits for Safe messages, (2) it sends Request messages again to neighbors, (3) it waits for their Reply, and, only after that, (4) it can send a new Safe message at time $\overrightarrow{s}_{u,x-1}$.

Let q_{x-2} (resp. q_{x-1}) be the time by which u receives the Request sent by v in round $[t'_{x-2}(v), t'_{x-1}(v)[$ (resp. $[t'_{x-1}(v), t'_x(v)[$). Let p_{x-2} and p_{x-1} be the times that node u replies respectively to those requests. Since v sends his request in time period $[t'_j(v), t'_{j+1}(v)[$ where $j \in \{x-2, x-1\}$, it is clear that $t'_j(v) < q_j \leq p_j < t'_{j+1}(v)$. By Lemma 2 and the proof therein, we also have that $p_j < \overrightarrow{s}_{u,j} < t'_{j+1}(v)$. Note also that node u cannot wait indefinitely for Safe messages from neighbors[2] by time $\overrightarrow{s}_{u,x-1}$.

Since v sets its counter to $\alpha_{x-1} \in \{x-1, x\}$ then to x, the Safe(y) message sent by u at time $\overrightarrow{s}_{u,x-2}$ (resp. $\overrightarrow{s}_{u,x-1}$) must verify $y \geq x-2$ (resp. $y \geq x-1$). Thus, $count_u(\overrightarrow{s}_{u,x-2}) \geq x-2$ and $count_v(\overrightarrow{s}_{u,x-1}) \geq x-1$.

Let $y = count_u(\overrightarrow{s}_{u,x-2}) \geq x-2$. Before time $\overrightarrow{s}_{u,x-2}$ the counter of u was also equal to y and u has executed a Request/Reply communication round. We denote by r_{x-2} the time corresponding to u beginning its requests. It is clear (Lemma 2) that $r_{x-2} < \overrightarrow{s}_{u,x-2}$ and from time r_{x-2} to time $\overrightarrow{s}_{u,x-2}$ the counter of u is at least y. Suppose that after time $\overrightarrow{s}_{u,x-2}$ node u does not set its counter to $\alpha_{x-1} \geq x-1$, i.e.,either it sets its counter to $x-2$ or it resets its counter. In the first case, during round $[t'_{x-1}(v), t'_x(v)[$, node v would have received a Reply$(x-2)$ and then, would have set its $increment_v$ variable to $false$. Node v could not increment its counter

[1] Otherwise, if v does not send a Safe message to u, then it resets its counter to 0 and does not increment it.

[2] Otherwise, it cannot send new Safe messages and v would never have set its counter to x

from $x-1$ to x should the case rise, which contradicts our definition of date $t'_x(v)$. In the second case, the next time u sends a $\mathsf{Safe}(y')$ message to neighbors, this message must verify $y' \leq -1$. However, by Lemma 2 and the discussion therein, this Safe message is sent at time $\overrightarrow{s}_{u,x-1}$ and thus $y' \leq -1 \geq 0$, contradiction. Therefore, node u does set its counter to α_{x-1} at some time, say r_{x-1}, verifying $r_{x-1} > \overrightarrow{s}_{u,x-2}$. Since after $\overrightarrow{s}_{u,x-2}$, node u stops replying to new requests until receiving all Safe messages from neighbors not in Q_u, it is clear that $r_{x-1} < p_{x-1}$. After time r_{x-1}, node u executes a new Request/Reply communication round. Thus, it cannot receive a $\mathsf{Reply}(-1)$ message from any neighbor since otherwise it would not have sent a $\mathsf{Safe}(\geq 0)$ message to v by time $\overrightarrow{s}_{u,x-1}$. Thus, until $\overrightarrow{s}_{u,x-1}$, the counter of u is still equal to α.

Thus, to sum up, for $j = x-1$ we have defined two time points r_{x-2} and r_{x-1} such that: $r_{x-2} < r_{x-1}$, $t'_{x-2}(v) < r_{x-1} < t'_x(v)$, $\forall t' \in [r_{x-2}, r_{x-1}[$, $count_u(t') \geq x-2$ and $\forall t' \in [r_{x-1}, \overrightarrow{s}_{u,x-1}]$, $count_u(t') \geq x-1$. In other words, r_{x-2} and r_{x-1} stands respectively for time points $t'_{x-2}(u)$ and $t'_{x-1}(u)$ claimed in the lemma for rank $j = x-1$.

Now, suppose that we have defined similarly the dates $r_{j-1} = t'_{j-1}(u)$ and $r_j = t'_j(u)$ for a rank $j \in \{2, \ldots, x-1\}$. Recall that this assumption implies the definition of dates $q_j, p_j, q_{j-1}, p_{j-1}$ as explained earlier. Now consider the round performed by v during time period $[t'_{j-2}(v), t'_{j-1}(v)[$. Let $\overrightarrow{s}_{u,j-2}$ be the time by which u sends its Safe message to v. Still from the algorithm description, node u must have executed a round of Request/Reply with neighbors in time period $[\overrightarrow{s}_{u,j-2}, \overrightarrow{s}_{u,j-1}[$. Again, after time $\overrightarrow{s}_{u,j-2}$, node u makes the following actions : (1) it waits for Safe messages, (2) it sends Request messages again to neighbors, (3) it waits for their Reply messages and, finally, (4) it can send a new Safe message at time $\overrightarrow{s}_{u,j-1}$.

Let q_{j-2} be the time by which u receives the Request sent by v in round $[t'_{j-2}(v), t'_{j-1}(v)[$. Let p_{j-2} be the time by which u replies to this request. Since v sets its counter from $\alpha \geq j-2$ to $\alpha' \geq j-1$, the $\mathsf{Safe}(y)$ message sent by u at time $\overrightarrow{s}_{u,j-2}$ must verify $y \geq j-2$. Thus, $count_u \overrightarrow{s}_{u,j-2} \geq j-2$.

Let $y = count_u(\overrightarrow{s}_{u,j-2}) \geq j-2$. Before time $\overrightarrow{s}_{u,j-2}$, the counter of u was also equal to y and u has executed a Request/Reply communication round. We denote by r_{j-2} the time corresponding to u starting its requests. It is clear (Lemma 2) that $r_{j-2} < \overrightarrow{s}_{u,j-2}$ and from time r_{j-2} to time $\overrightarrow{s}_{u,j-2}$, the counter of u is at least y. Suppose that after time $\overrightarrow{s}_{u,j-2}$, node u does not set its counter to $\alpha' \geq j-1$ i.e. either it sets its counter to $j-2$ or it resets its counter. In the first case, during round $[t'_{j-1}(v), t'_j(v)[$, node v would have received a $\mathsf{Reply}(j-2)$ and then, would have set its $increment_v$ variable to $false$. Node v could not increment its counter from $j-1$ to j should the case rise (i.e. if $count_v(t'_{j-1}) = j-1$ and $count_v(t'_j) = j$). This contradicts our inductive assumption. In the second case, the next time u sends a $\mathsf{Safe}(y')$ message to neighbors, this message must verify $y' \leq -1$. However, by Lemma 2 and the discussion therein, this Safe message is sent at time $\overrightarrow{s}_{u,j-2}$ and thus, $y' \leq -1 \geq 0$, contradiction. Therefore, node u does set its counter to $\alpha' \geq j-1$ at time r_{j-1} , with $r_{j-1} > \overrightarrow{s}_{u,j-2}$. Since after $\overrightarrow{s}_{u,j-2}$, node u stops replying to new requests until receiving all Safe messages from neighbors

not in Q_u, it is clear that $r_{j-1} < p_{j-1}$. After time r_{j-1}, node u executes a new Request/Reply communication round. Thus, it cannot receive a Reply(-1) message from any neighbor since otherwise it would not have sent a Safe(≥ 0) message to v by time $\overrightarrow{s}_{u,j-1}$. Thus, until $\overrightarrow{s}_{u,j-1}$, the counter of u is still equal to α.

Thus, to sum up, for a rank $j-1$, the definition of points r_{j-1} and r_j permits to define a new time point r_{j-2} such that : $r_{j-2} < r_{j-1}$, $t'_{j-2}(v) < r_{j-1} < t'_j(v)$, $\forall t' \in [r_{j-2}, r_{j-1}[, count_u(t') \geq j-2$. In other words, r_{j-2} stands for the time point $t'_{j-2}(u)$ claimed in the lemma for rank $j-1$. Thus, by assuming the property proven for a rank j, we proved that property for rank $j-1$ and thus for all ranks from $x-1$ down to 2.

□

Lemma 4. *Any Request/Reply/Safe communication round of our algorithm is deadlock free.*

Proof. From the algorithm description, if a node v gets blocked waiting for a message from a neighbor u, then this is because either:

- node u does not respond to v's Request with a Reply message, or
- node u does not respond to v's Safe message with a Safe message.

Let us first prove that whenever a node v sends a Safe message to a neighbor u during a Request/Reply/Safe communication round, it does not get blocked waiting for a Safe message from u. From the algorithm description, whenever node v sends a Safe message to neighbor u, this means that $u \in Q_v$, i.e., u replied with Reply(≥ 0) to v's Request. Consider the t by which node u received v's Request. Node u replied with a Reply(≥ 0) message to v. Then node v ends its round by sending a Safe message to u and waiting for a Safe message from u. From this point, two situations can occur:

- Node u begins a new round with $count_u < X$. It sends a Request message to v, receives a Reply(≥ 0) from v because v is waiting for a Safe message. Thus, we have $v \notin Q_u$, meaning that, after receiving all the replies from neighbors, node u will send a Safe message to v.
- Node u set its counter to a value $\geq X$. After receiving the Safe message from v, node u sends Terminated message to v. Node v will add node u in Q_v and then gets unblocked.

Now, consider the case where a node v gets blocked waiting for a Reply message, say from a neighbor u. First remark that as long as node v is waiting for u's Reply, we have that $ready_v = true$. In addition, we remark that a node requesting for an interval is necessarily executing the while loop of algorithm 1, i.e., it has not yet send its Terminated message. Thus, none of its neighbors has terminated the algorithm. Thus, these neighbors do process v's Request in a finite time, i.e., the neighbors do execute the receive triggers of line 28. Consider the time t node v begins sending

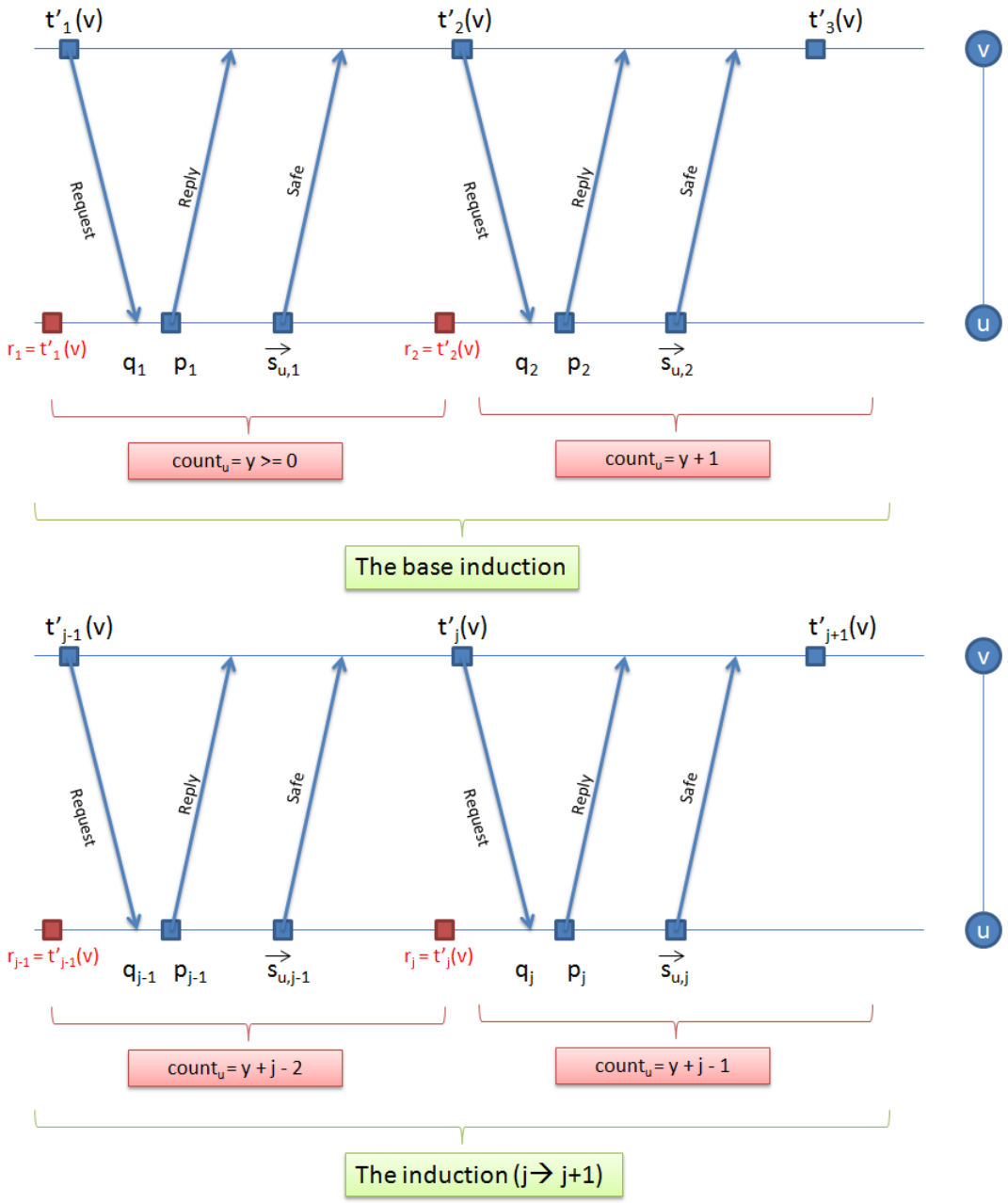

Figure 4.6: The inductive proof in Lemma 3

its requests to neighbors. Let t' be the time by which node u receives v's request. When receiving v's request, node u does not respond (i.e., it delays its response at line 32) in only two cases: either $ready_u = false$ or $v \in Q_u$ at time t'. We thus have two cases:

- If $ready_u = false$, then from the algorithm description this means that node

u is waiting for Safe messages at line 18 from neighbors in Q_u. Symmetrically u must have sent Safe message to those neighbors. Thus, from the previous discussion u does not get blocked waiting for Safe messages from those neighbors. Just after receiving those Safe messages, node u sets its $ready_u$ variable to $true$, thus it can answer v's request.

- If $ready_u = true$, then necessarily $v \in Q_u$. This means that v replied to u's request with a Reply(-1) message, i.e., before beginning its requests at time t, node v had a work interval but had entirely explored it. Thus, node u does not sent a Safe message to v and does not get blocked for neither a Reply nor a Safe message from v. Thus, the only way that node u does not respond to v's request is that node u gets blocked waiting for a Reply message from a neighbor u_1. Let us denote by t_1 the time node u sends its Request to node u_1. It is clear that that $u_1 \neq v$ since $v \in Q_u$ (v has already replied). We thus can repeat the same reasoning for node u waiting now for a Reply from neighbor u_1. We then get two cases: either $ready_{u_1} = false$ or $u \in Q_{u_1}$. If $ready_{u_1} = false$, then by the same reasoning u_1 cannot stay blocked indefinitely for a Safe message and thus ends by replying to u's request within a finite time. Hence v ends by receiving a Reply from u. If now $ready_{u_1} = true$ and $u \in Q_{u_1}$, we can apply the same reasoning and find a node $u_2 \neq u_1$ such that: u_1 is blocked for a Reply message from u_2. Similarly it is easy to see that $u_2 \neq u$. Assume that $u_2 = v$ meaning that u_1 is blocked waiting for a Reply message from v. Denote by t_2 the time node u_1 sends its Request to node $u_2 = v$. Consider the time t'_2 by which that message arrives at u_2. If $t'_2 > t$ we are done since $ready_v = true$ after time t meaning that u_1 ends by receiving a reply from $u_2 = v$ and also node u. Suppose now that $t'_2 < t$. Recall that just before time t, node $u_2 = v$ was exploring an interval. Denote by t'' the time node v begins exploring that interval. It is not difficult to see that from time t'' to time t, we have that $count_v = -1$. Suppose that $t'_2 \in [t'', t]$, then node $u_2 = v$ immediately respond to u_1 request and we are done. Suppose that $t'_2 < t''$. Consider the time t''' corresponding to the Request/Reply/Safe communication round (If no such a time point exists, then this means that $t'' = 0$ and we are done since necessarily $t'_2 \in [0, t]$). Since v terminated that round, it has necessarily responded to u_1's request with a Reply message at most by time t''. Thus u_1 ends receiving a Reply message from $u_2 = v$ within a finite time and we are done. Now if $u_2 \neq v$, then we can repeat the same arguments for node u_2 verifying $u_2 \notin \{u_1, u, v\}$. It is not difficult to see that we can repeat our reasoning by induction at most $\ell \leq n$ times and find a path of node $(v, u = u_0, u_1, u_2, \cdots, u_\ell)$ where for every $i < \ell$ node u_i is waiting for a Reply from node u_{i-1} and node u_ℓ does respond to node $u_{\ell-1}$. Thus, node v cannot get blocked for an infinite time.

□

4.2.3 Termination Detection

The following lemma aims at proving that whenever the counter of a peer reaches a given bound, then the peer can locally decide that no work remains in the network. Notice that the Lemma does not guarantee that the value of the counter of a node will eventually reach that bound — This is proved afterward.

Lemma 5. *At any time $t \geq 0$, and for any node $v \in V$, if $count_v(t) = 4D$ then no work interval remains anywhere in the network.*

Proof. For conveniency, let $X = 4D$. Suppose that $count_v(t) = X$. Consider any node $w \neq v$ and the shortest path P of length ℓ starting from v and leading to w. By definition, we have that $\ell \leq D$. Denote this path $P = (u_0 = v, u_1, \cdots, u_j, \cdots, u_\ell = w)$ with $j \in \{0, \ldots, \ell\}$. From Lemma 3, we get a sequence of time points $S_X^{u_0,u_1} = \{t'_{i_1}(u_1) \mid i_1 \in \{1, \ldots, X-1\}\}$ that verifies the four properties stated in Lemma 3. In particular, for $i_1 \in \{1, X-1\}$, it holds that $count_{u_1}(t'_1(u_1)) \geq 1$ and $count_{u_1}(t'_{X-1}(u_1)) \geq X-1$. By Lemma 3, we have also:

$$\begin{aligned} t'_1(u_1) &< t'_2(u_0) \\ t'_2(u_1) &< t'_3(u_0) \qquad (4.1) \\ &\cdots \\ t'_{X-4}(u_0) &< t'_{X-3}(u_1) \qquad (4.2) \\ t'_{X-3}(u_0) &< t'_{X-2}(u_1) \\ t'_{X-2}(u_0) &< t'_{X-1}(u_1) \end{aligned}$$

We can again apply Lemma 3 for node u_2 with respect to time point $t'_{X-1}(u_1)$ corresponding to the time where neighbor u_2 sets its counter to $X-1$. Thus, we obtain a new sequence of time points $S_{X-1}^{u_1,u_2} = \{t'_{i_2}(u_2) \mid i_2 \in \{2, \ldots, X-2\}\}$ for node u_2. This sequence of time points also verifies the properties of Lemma 3. In particular, for $i_2 \in \{1, X-2\}$, $count_{u_2}(t'_2(u_2)) \geq 2$ and $count_{u_2}(t'_{X-2}(u_2)) \geq X-2$. We also have:

$$\begin{aligned} t'_2(u_2) &< t'_3(u_1) \qquad (4.3) \\ t'_3(u_2) &< t'_4(u_1) \\ &\cdots \\ t'_{X-4}(u_1) &< t'_{X-3}(u_2) \\ t'_{X-3}(u_1) &< t'_{X-2}(u_2) \qquad (4.4) \end{aligned}$$

By combining Eq. 4.3 with Eq. 4.1, and Eq. 4.4 with Eq. 4.2, we get that node u_2 (at distance 2 from $u_0 = v$) verifies:

$$\begin{aligned} t'_2(u_2) &< t'_4(u_0 = v) \\ t'_{X-4}(u_0 = v) &< t'_{X-2}(u_2) \end{aligned}$$

Because the counter of a node $u \in \{v, u_1, u_2\}$ is at least 0 at time $t'_2(u)$ and can not decrease (by Lemma 3), the counters of nodes v, u_1 and u_2 are at least 0 in time interval[3] $[t'_4(v), t'_{X-4}(v)]$, i.e., none of these nodes holds a B&B work interval.

By induction, it is not difficult to see that, for every $j \in \{1, \ldots, \ell\}$, there exist a sequence of time points $S_{X-j+1}^{u_{j-1},u_j} = \left\{ t'_{i_j}(u_j) \mid i_j \in \{j, \ldots, X-j\} \right\}$ for node u_j with respect to time point $t'_{X-j+1}(u_{j-1})$ that verifies the properties of Lemma 3. In particular, by a routine induction on j, the following holds for well chosen values of $i_j \in \{j, \ldots, X-j\}$:

$$t'_{i_j}(u_j) < t'_{i_j+j}(v) \tag{4.5}$$

$$t'_{i_j-j}(v) < t'_{i_j}(u_j) \tag{4.6}$$

Let us take any $j \in \{1, \ldots, \ell\}$. From Eq. 4.5 and for $i_j = 2D - j \geq 1$, we get: $t'_{2D-j}(u_j) < t'_{2D}(v)$. Similarly, for $i_j = 2D + j \leq X - j$ in Eq. 4.6, we get: $t'_{2D}(v) < t'_{2D+j}(u_j)$. Notice that by Lemma 3, the counter of u_j in time period $[t'_{2D-j}(u_j), t'_{2D+j}(u_j)]$ is at least 0, i.e., node u_j holds no B&B work interval. Thus, for every $j \in \{1, \ldots, \ell\}$, neither v nor u_j has a work interval at time $t'_{2D}(v)$.

Finally, since any node in the overlay graph is at distance at most D from node v, no node in the network holds a B&B work interval to explore at time $t'_{2D}(v)$. Since work intervals cannot appear spontaneously or be created by nodes after time $t'_{2D}(v)$, the lemma is proved. □

From the algorithm description, whenever a node u receives a work request (i.e., Get_Work message) from a neighbor v, it answers with a part of the interval it holds if any (no redundancy). After requesting a sub-interval from a neighbor u, a node v waits until receiving a sub-interval from u, i.e., it does not ask other neighbors for a sub-interval unless[4] it receives an empty one from u. Then, after finishing a Request/Reply/Safe communication round, node v runs the B&B search process on the obtained sub-interval. Thus, by Lemma 4, whenever a node asks for an interval from a neighbor it either ends exploring the obtained sub-interval or it starts a new Request/Reply/Safe communication round. Since the B&B process eventually ends exploring the whole interval, it is easy to see that there exists a time point throughout the algorithm execution such that no interval remains to explore anywhere in the network. In the following, we denote by $t^\star$ the first time where there remain no intervals to explore in the network.

Let us consider any node v at time $t^\star$. It is clear that either v has already started a Request/Reply/Safe round at some time $t_v^0 < t^\star$ and it has not finished it yet, or node v is starting a new Request/Reply/Safe communication round at time $t_v^0 = t^\star$. Similarly, we define t_v^i (where $i \geq 1$ is an integer) the time node v started its i^{th} Request/Reply/Safe communication round after time $t^\star$. Besides, let us define the following : $\forall k \in \mathbb{N}, \forall v \in V, N_k(v)$ is the set of nodes located at a distance equal to k from the node v.

[3] Of course, and to make the proof more comprehensive, we implicitly assume that $X - 4 > 4$.

[4] This may happen if node u finishes exploring its interval before receiving v's Get_Work message.

In the following, we argue that after time $t^\star$, the nodes cannot reset their counters infinitely many, and thus the algorithm terminates by Lemma 5.

Lemma 6. $\forall u \in V, \exists n \in \mathbb{N} | \forall i \geq n, count_u(t_u^i) \geq 4D$

Proof. Let us prove this lemma by absurd. Suppose that we have : $\exists u \in V | \forall i \in \mathbb{N}, count_u(t_u^i) < 4D$

Thus, this property is true for $i \geq 5D$. So, the statement becomes $count_u(t_u^{5D}) < 4D$. This means that if node u had set its counter to $4D-1$ upon finishing its round ending at time t_u^{5D-1}, it has not incremented its counter by the end of the following round ending at t_u^{5D}. Considering Algorithm 1, the condition of line 23 is not satisfied. So, node u has received whether a $\mathsf{Safe}(< 4D-1)$ message from a neighbor not in Q_u on line 20 or a $\mathsf{Reply}(-1)$ from a neighbor in $N(u)$. The second case obviously leads to a contradiction with the termination assumption. Let us consider the first case.

Let us define $\forall i \in \mathbb{N}, \forall u \in V, B_i(u) = \bigcup_{k=0}^{\min(i,D)} N_k(u)$. The second case depicted hereabove implies that one of u's neighbors had its counter set at a value strictly lower than $4D-1$ at the end of the previous round. More formally, $\exists v_1 \in B_1(u) | count_{v_1}(t_{v_1}^{5D-1}) < 4D-1$. Using our initial assumption, one can state : $\exists v_1 \in B_1(u) | \forall k \geq 5D-1, count_{v_1}(t_{v_1}^k) < 4D-1$. (See Figure 4.7)

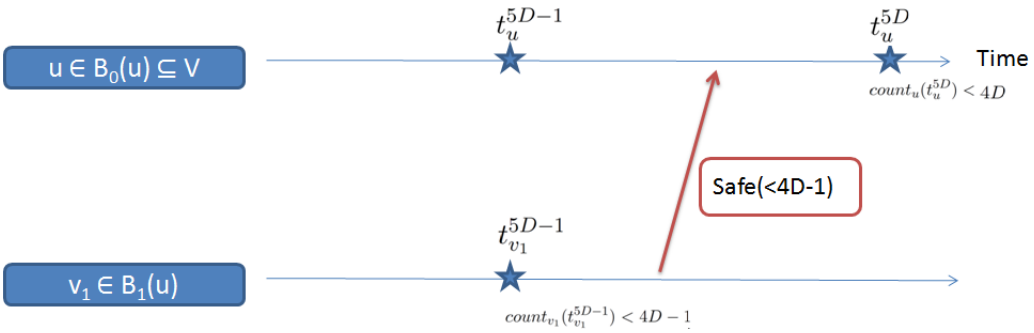

Figure 4.7: Initial step of the proof of Lemma 6

Let us prove the induction for an integer $p \in [1, 4D]$. Suppose that we have : $\exists v_p \in B_p(u) | \forall k \geq 5D-p, count_{v_p}(t_{v_p}^k) < 4D-p$

If this node v_p has set its counter to $4D-(p+1)$ upon finishing its round ending at time $t_{v_p}^{5D-(p+1)}$, then it does not increment its counter up to $4D-p$ by the end of the following round ending at $t_{v_p}^{5D-p}$. Thus, similarly, the condition of line 23 has not been satisfied which implies two cases. First,the condition $increment_{v_p} = false$ can be due to a neighbor in $v_{p+1} \in N(v_p) \subseteq B_{p+1}(u)$ which sent a $\mathsf{Safe}(< 4D-(p+1))$ message during its round ending at time $t_{v_{p+1}}^{5D-(p+1)}$ or due to a neighbor $v_{p+1} \in N(v_p)$ which sent a $\mathsf{Reply}(-1)$ message to v_p. The second case leads to a contradiction with our initial assumption.

Thus, in the first case, one of v_p's neighbors, referred to as v_{p+1}, has set its value to a value strictly lower than $4D-(p+1)$ by time $t_{v_{p+1}}^{5D-(p+1)}$. More formally, $\exists v_{p+1} \in N(v_p) \subseteq B_{p+1}(u) | count_{v_{p+1}}(t_{v_{p+1}}^{5D-(p+1)}) < 4D-(p+1)$. Using the induction's assumption, one can state $\exists v_{p+1} \in N(v_p) \subseteq B_{p+1}(u) | \forall k \geq 5D-(p+1), count_{v_{p+1}}(t_{v_{p+1}}^{5D-(p+1)}) < 4D-(p+1)$. The property remains valid for rank $p+1$. (See Figure 4.8)

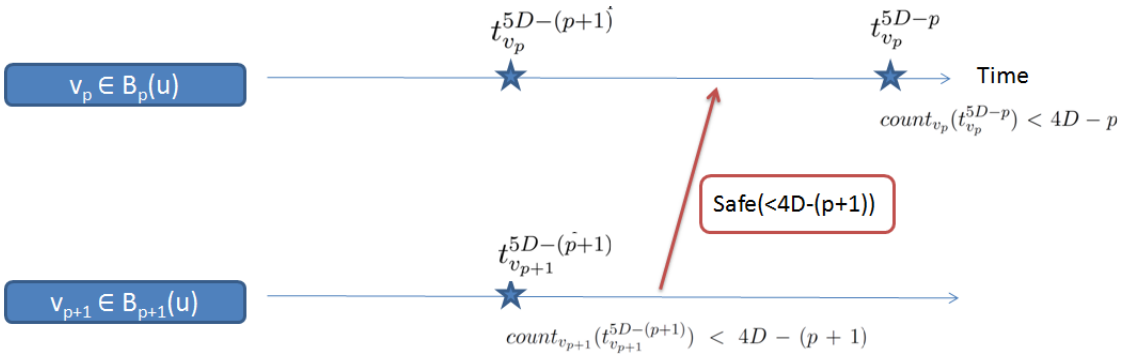

Figure 4.8: Inductive step of the proof of Lemma 6

Consequently, if we consider the case $p = 4D$, the property becomes $\exists v_p \in B_p(u) | \forall k \geq 5D-p, count_{v_p}(t_{v_p}^{5D-p}) < 4D-p$ which means equivalently: $\exists v_{4D} \in B_{4D}(u) = V | \forall k \geq D, count_{v_{4D}}(t_{v_{4D}}^{D}) < 0$.

This last property indicates that one peer in the network, arbitrarily named v_{4D}, had set its counter to a strictly negative value when it finished it round on date $t_{v_{4D}}^{D} > t^\star$ i.e. it sent whether a Reply(-1) message or a Safe(-1) message to its neighbors, indicating that it was exploring a work unit by time $t_{v_{4D}}^{D}$ (See Figure 4.9). This contradicts the definition of time $t^\star$ after which no work units remain in the Peer-to-Peer network and the lemma holds.

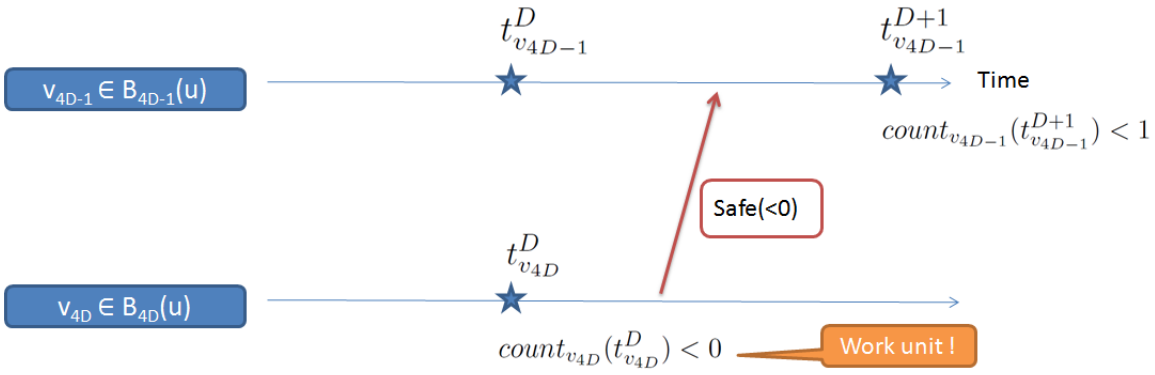

Figure 4.9: Final step in the proof of Lemma 6

□

4.3 Conclusion

In this chapter, we provided a formal proof of the correctness of our approach. We proceeded into five main steps successively proving that: (i) All the solutions will be explored within a finite amount of time, i.e. no solution can be explored twice. (ii) A peer synchronizes its rounds of requests with its neighbors, to avoid continuously asking for the same piece of information repeatedly and uselessly and to make sure that the information being communicated remains identical and coherent during the communication process. (iii) A peer cannot detect termination while another peer is still exploring solutions. (iv) Any round of requests is achieved within a finite amount of time. (v) Every peer eventually detects termination within a finite amount of time.

Chapter 5

Experimental Study of B&B-P2P

Contents

5.1 Introduction

In this chapter, we report the experimental results we obtained for the performance evaluation of our approach using the Grid'5000 experimental grid [Grid'5000]. Our experiments have two main objectives: (i) proving experimentally the correctness of our approach by correctly solving to optimality some Combinatorial Optimization Problems, and (ii) assessing the performance and the scalability of our approach in a large scale distributed setting. In particular, we show that the communication load is efficiently distributed among all the peers in the network so that it allows to harness huge amounts of resources. To achieve this goal, we first evaluate the performances of our approach on a set of instances of the Flow-Shop Scheduling Problem (FSPs) in a medium scale network. Then, we shift towards large-scale environments and compare our results to a Master-Slave based approach to analyze its scalability. Finally, we study the impact of the network overlay topology on performance through studying various topologies taken from the literature.

The remainder of this chapter is organized as follows. Section 5.2 provides a description of our experimental methodology. Section 5.3 consists in experiments conducted on medium-scale networks involving up to 1000 physical cores to solve some combinatorial optimization instances. Section 5.4 provides a study of the performances of our approach at large scales involving up to 150.000 logical peers and compared to a Master-Slave approach taken from [Mezmaz 2007b]. Section 5.5 studies the impact of the network overlay on the communications induced by the Peer-to-Peer design of our approach.

5.2 Experimental Setting

5.2.1 Experimental Testbed

Our experiments have been conducted on the French nation-wide Grid'5000 Experimental Grid [Grid'5000]. The Grid5̀000 project had been launched in 2003 thanks to the French Research and Higher Education Ministry, in the context of the GRID project "Programme national Grid'5000". It was also supported by other institutions such as INRIA, CNRS, various universities, RENATER and some local and regional administrations throughout France. The interconnection network is a Virtual Private Network (VPN) built on top of the RENATER network, composed of 10 Gbps network links using optical fibers. At the time of writing, in 2012, the grid comprises about 8.150 computational cores which are geographically

dispatched on 10 different sites in France: Lille, Reims, Orsay[1], Nancy, Rennes, Bordeaux, Toulouse, Grenoble, Lyon, Sophia-Antipolis (See Figure 5.1). Indeed, this grid has an important scale in terms of computational resources and network interconnection. All the physical machines used features multi-core processors (varying from 2 to 24 depending on the machine). These resources are managed by different institutions. Each site of the grid is supervised by a dedicated engineer. However, our experimental testbed is a dynamic environment. Indeed, resources in Grid'5000 have to be reserved before being used. A reservation (also called *a job*) can be made under three modes: normal, deploy (where you can deploy your own operating system on the nodes) and best-effort (where some of the reserved resources can be reassigned to another job during the reservation). In our experiments, where a *static* environment is required (to avoid the volatility of the resources), we made our reservations using the normal mode. This allows us to fulfill the requirements that computational nodes are supposed to be reliable and no failure of any type can occur. No entities can join or leave the network at any time.

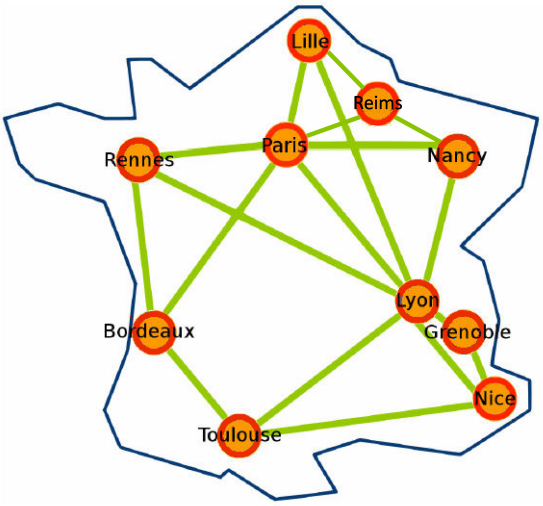

Figure 5.1: Grid'5000 geographical Sites.

Having this in mind, our goal is to study the behavior of our approach at various scales and for different overlay topologies. The target scales range from 1000 peers up to 150.000 peers. For that purpose, a fixed set of machines is used on Grid'5000, namely 569 physical nodes (consisting in 2046 computational cores) distributed on 8 geographical sites (all sites except Nancy and Reims). All the obtained results are an average of three separate runs. Depending on the size of the overlay graph we want to experiment, each physical machine can hold a well-defined number of processes, each process modeling a node in the overlay. Each process then runs our distributed algorithms as described in previous sections. In particular, exploration of intervals is done using the classical sequential B&B algorithm and communication

[1]This site is used in our experiment but has been officially removed from the grid by early 2012.

between processes is implemented using C++ sockets. The way these processes are dispatched is described in Subsection 5.4.2 hereafter.

5.2.2 Experimental protocols

Since we target large scale P2P network with the number of peers being possibly much larger then the actual number of computing cores available from the Grid'5000 platform, we will consider the following two protocols when experimenting our approach.

5.2.2.1 Real-Case deployment scenario

In this scenario, we deploy one peer per physical core. More precisely, let us consider a number N of peers. To study the performances of a network composed of N peers cooperating with each other, we deploy this network on exactly N of physical computational cores (See Figure 5.2). One peer will be running "full-time" on a single core and the N peers can run distributively in parallel using the N cores. One has to notice that depending on how the logical overlay is constructed, two peers on two different cores belonging to the same machine may or may not be neighbors in the overlay.

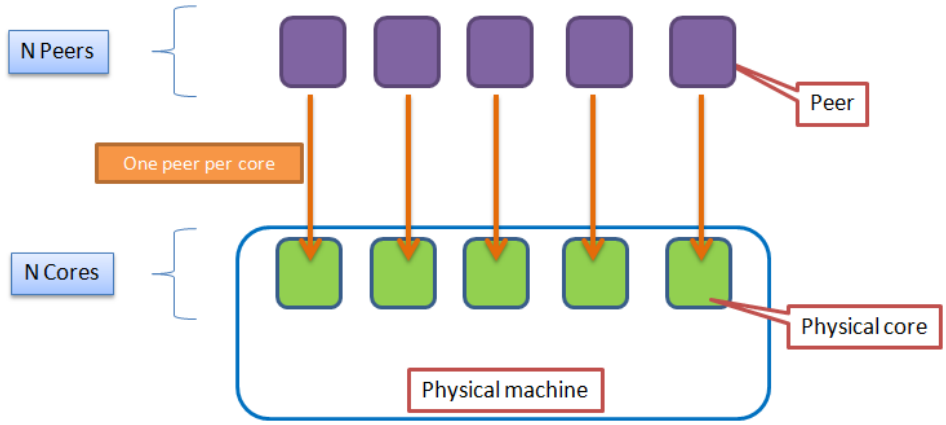

Figure 5.2: Real-case deployment scheme

5.2.2.2 Simulated P2P Environment

In this scenario, we experiment our approach for large-scale P2P networks while using a reduced amount of physical computing resources (see Figure 5.3). One of the main issues which have to be dealt with when deploying large-scale P2P networks over a reduced number of physical machines is the scheduling the processes. In our experiments, we use approximatively 2000 physical cores, on which we deploy up to 150.000 peers. This represents a ratio up to 75 peers per computational core. One may think that, on each machine, the operating system will share equally the

CPU time among all the peers. However, it is not the case, practically. On a given machine, some of the peers may receive a work unit prior to the others. Thus, they will start exploring solutions and then, require more computational resources than the "idle" peers. The operating system will then allocate more CPU time to the "resource consuming" peers. This situation can end up with some peers that have been running for several hours whereas some other peers only for some seconds.

In our experiments, we manage explicitly the scheduling of the processes on the CPU using a simple technical procedure. More precisely, let us consider a machine whose CPU has a number n of physical cores and running a number X of peers. Then, we allow a number $\alpha = \beta \times n \leq X$ of peers to access to the CPU, i.e. a number β of peers per core and to pause the execution of the other peers. In the case where none of the α peers are processing work units, it allows them to perform some "low resource consuming" operations like sending messages. To make sure that all the peers have access to the computational resources, these peers must be periodically part of the α running peers. More precisely, consider that all the peers running on the machine can be identified by a number comprised between 0 and $X - 1$. Initially, peers ranging from 0 to $\alpha - 1$ are allowed to run. Then, after an arbitrary time period T, peers ranging from 1 to α will be allowed to run, then from 2 to $\alpha + 1$ and so on. This procedure is depicted in more details in Figure 5.4 and Algorithm 2.

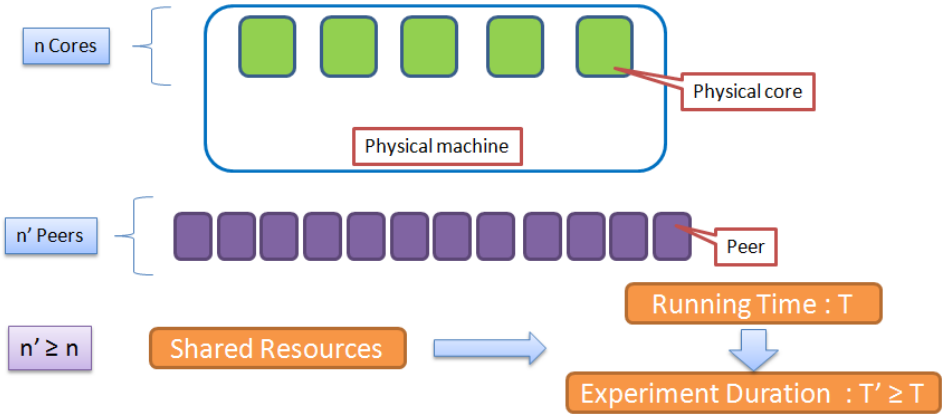

Figure 5.3: Simulated environment deployment scheme.

5.2.3 Implementation and deployment of the overlay topology in Grid'5000

Our P2P approach operates in static environments, where resources are stable and non volatile i.e. no peer can dynamically join or leave the network during the execution of the approach. Therefore, the overlay topology is given as an input to our application, as a file named *"topology.txt"*. We remind that a description of the topology is given in Section 3.5. Let N be the number of peers in the network. The deployment procedure of the toric hypercube overlay on Grid'5000 is performed in

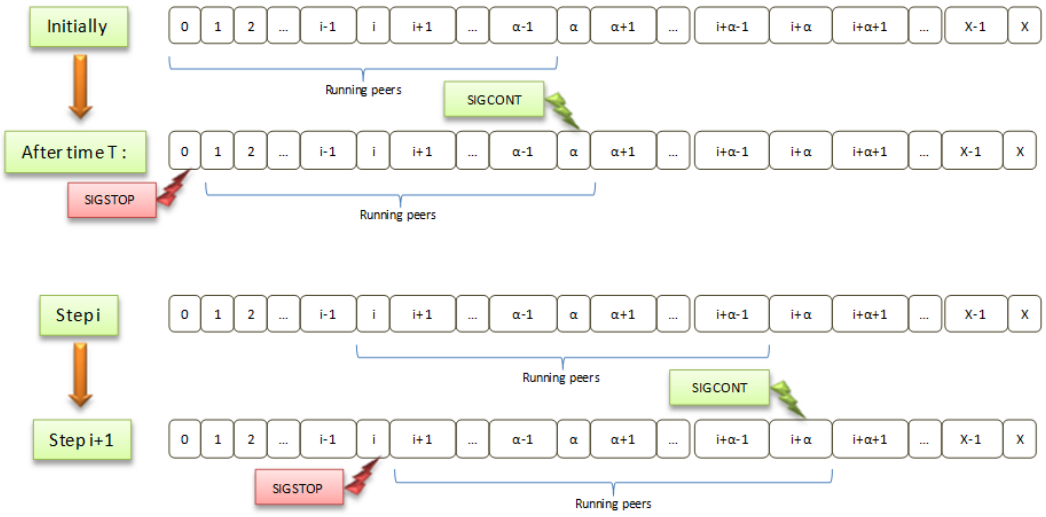

Figure 5.4: Scheduling processes on a single machine.

Algorithm 2: Scheduling procedure

```
   /* Variables                                                    */
1  T : Arbitrary period of time;
2  X : Number of peers running on the local machine. α : Number of peers
     allowed to access computational resources. {0, ..., X − 1} : Set of integers
     used to identify all the peers running on the local machine. i, j ∈ N : Local
     variable.
   /* Scheduling algorithm                                         */
3  i ← 0;
4  j ← 0;
5  while true do
6      Wait for a period of time T;
7      j ← (i + α  mod X);
8      Send SIGSTOP signal to peer i;
9      Send SIGCONT signal to peer j;
10     i ← (i + 1  mod X) ;
```

several steps.

At the first step, upon reserving a set of machines on Grid'5000, the *"oargrid"* tool (more precisely, the *"oargridstat"* command) is used to get the list containing the hostnames of the reserved nodes, stored in "reserved_machines.txt".

The second step consists in creating the list of peers that will be deployed on the machines. Technically speaking, for each machine <host> contained in *"reserved.txt"*, we define $k_{<host>}$ as the number of cores available on the machine. Then, we create k couples "<host> <port>+m" where m ranges between 0 and $k_{<host>} - 1$. The variable <port> is an arbitrary value (it is set to 6000 in our

experiments), and each couple "<host> <port>+m" indicates that a peer can be joined on machine <host> on port <port>+m. These informations are appended to a file named *"peers.txt"*. This file constitutes a kind of global index for the P2P network.

At the third step, the files *"topology.txt"*, *"peers.txt"*, and *"reserved_ nodes.txt"* are copied on the sites containing the reserved machines as well as the source files needed to run the application. Then, the deployment script orders each machine <host$_i$> in *"reserved_ nodes.txt"* to run a script named *"machine_ deploy.sh"* to continue the deployment procedure in parallel. This script takes two integers *begin* and *end* as parameters. It first compiles locally the source files of the application and runs all the peers whose identifiers are comprised in the interval $[begin, end]$. These two integers are determined by the order of <host$_i$> in the list contained in *"reserved_ nodes.txt"*. The mapping of the toric hypercube overlay on the Grid'5000 physical network uses a cluster-based proximity of peers provided by the OAR reservation tool. Such mapping is similar to the IP-based proximity metric used in [Nguyen 2012]. For instance, if <host$_i$> is the first machine listed in the file, then the script launch the peers whose identifiers are in $[0, k_{<host_1>} - 1]$, that will communicate respectively on ports $[< port >, < port > +k_{<host_1>} - 1]$. For the second machine, it will launch the peers whose identifiers are in $[k_{<host_1>}, k_{<host_1>} + k_{<host_2>} - 1]$ that will communicate respectively on ports $[< port >, < port > +k_{<host_2>} - 1$ and so on. This process stops when the number of deployed peers reaches N.

So far, to illustrate this mechanism, consider an example of reservation on Grid'5000. Suppose that some computing nodes are reserved as depicted in Figure 5.5. The OAR server returns a list of nodes similar to that in Figure 5.6 which is saved in *reserved_ nodes.txt*. Using this list, the list of all peers deployed over the machines is created and saved in *peers.txt*, as shown in Figure 5.8. Then, each machine is ordered to deploy peers corresponding to a range $[begin, end]$. Example of these two values are given in Figure 5.7

Site	Cluster	Quantity	Total of cores
Sophia	azur	2	4
	suno	2	16
Grenoble	adonis	1	8
Lille	chicon	2	8
	chirloute	2	16
3 Sites	Total	9	52

Figure 5.5: Example of Clusters that can be reserved.

At the fourth step, every peer in the network is assigned an identifier in $[0, N-1]$. When a peer is launched, it takes its identifier as a parameter. The peer parses the $(i+1)^{th}$ line of the file *"topology.txt"* to retrieve the list of identifiers corresponding to its neighbors. For each identifier j contained in this list, the peer parses the $(j+1)^{th}$ line of the file *"peers.txt"* to retrieve the couple "<host> <port>" corresponding to

Content of *reserved_ nodes.txt*
azur-8.sophia.grid5000.fr
azur-9.sophia.grid5000.fr
suno-16.sophia.grid5000.fr
suno-24.sophia.grid5000.fr
adonis-7.grenoble.grid5000.fr
chicon-13.lille.grid5000.fr
chicon-24.lille.grid5000.fr
chirloute-3.lille.grid5000.fr
chirloute-4.lille.grid5000.fr

Figure 5.6: Example of *reserved_ nodes.txt* file obtained from OAR

Content of *reserved_ nodes.txt*	*begin*	*end*
azur-8.sophia.grid5000.fr	0	1
azur-9.sophia.grid5000.fr	2	3
suno-16.sophia.grid5000.fr	4	11
suno-24.sophia.grid5000.fr	12	19
adonis-7.grenoble.grid5000.fr	20	27
chicon-13.lille.grid5000.fr	28	31
chicon-24.lille.grid5000.fr	32	35
chirloute-3.lille.grid5000.fr	36	43
chirloute-4.lille.grid5000.fr	44	51

Figure 5.7: Example of the corresponding values of *begin* and *end* given as parameters to *machine_ deploy.sh*

the peer identified by j and stores it into a list in its memory. Thus, the peer can now contact its neighbors whenever needed during the remainder of its execution.

At the fifth step, before executing the P2P B&B algorithm, each peer must ensure that its neighbors have been effectively deployed. To do so, every peer continuously tries to send "ready" messages to its neighbors until it succeeds and receives in return a "ready" message from each neighbor. Any other kind of received message shall be delayed after the procedure is completed. This procedure is similar to the tool Taktuk [Claudel 2009] available on Grid'5000.

Finally, as one peer must be initially assigned the entire interval $[0, n![$ of solutions to explore (with n being the number of jobs to be scheduled in the FSP instance), if the identifier of the peer is 0 then it starts immediately exploring the interval $[0, n![$, else it begins performing rounds of requests with its neighbors.

Note that in our Simulated P2P environment protocol, the number of peers deployed on each machine is greater than the number of cores. Therefore, our deployment script takes a variable *peers_per_core* as a parameter. The main difference with the described deployment procedure is that, for a machine <host>, the script *"machine_ deploy.sh"* will start a number of peers equal to $peers\_per\_core \times k_{<host>}$.

The cost of this parallel deployment in terms of time is negligible and is not taken into account in our experiments.

Content of *peers.txt*		Peer Identifier
azur-8.sophia.grid5000.fr	6000	0
azur-8.sophia.grid5000.fr	6001	1
azur-9.sophia.grid5000.fr	6000	2
azur-9.sophia.grid5000.fr	6001	3
suno-16.sophia.grid5000.fr	6000	4
suno-16.sophia.grid5000.fr	6001	5
suno-16.sophia.grid5000.fr	6002	6
suno-16.sophia.grid5000.fr	6003	7
suno-16.sophia.grid5000.fr	6004	8
suno-16.sophia.grid5000.fr	6005	9
suno-16.sophia.grid5000.fr	6006	10
suno-16.sophia.grid5000.fr	6007	11
suno-24.sophia.grid5000.fr	6000	12
suno-24.sophia.grid5000.fr	6001	13
suno-24.sophia.grid5000.fr	6002	14
suno-24.sophia.grid5000.fr	6003	15
suno-24.sophia.grid5000.fr	6004	16
suno-24.sophia.grid5000.fr	6005	17
suno-24.sophia.grid5000.fr	6006	18
suno-24.sophia.grid5000.fr	6007	19
adonis-7.grenoble.grid5000.fr	6000	20
adonis-7.grenoble.grid5000.fr	6001	21
adonis-7.grenoble.grid5000.fr	6002	22
adonis-7.grenoble.grid5000.fr	6003	23
adonis-7.grenoble.grid5000.fr	6004	24
adonis-7.grenoble.grid5000.fr	6005	25
adonis-7.grenoble.grid5000.fr	6006	26
adonis-7.grenoble.grid5000.fr	6007	27
chicon-13.lille.grid5000.fr	6000	28
chicon-13.lille.grid5000.fr	6001	29
chicon-13.lille.grid5000.fr	6002	30
chicon-13.lille.grid5000.fr	6003	31
chicon-24.lille.grid5000.fr	6000	32
chicon-24.lille.grid5000.fr	6001	33
chicon-24.lille.grid5000.fr	6002	34
chicon-24.lille.grid5000.fr	6003	35
chirloute-3.lille.grid5000.fr	6000	36
chirloute-3.lille.grid5000.fr	6001	37
chirloute-3.lille.grid5000.fr	6002	38
chirloute-3.lille.grid5000.fr	6003	39
chirloute-3.lille.grid5000.fr	6004	40
chirloute-3.lille.grid5000.fr	6005	41
chirloute-3.lille.grid5000.fr	6006	42
chirloute-3.lille.grid5000.fr	6007	43
chirloute-4.lille.grid5000.fr	6000	44
chirloute-4.lille.grid5000.fr	6001	45
chirloute-4.lille.grid5000.fr	6002	46
chirloute-4.lille.grid5000.fr	6003	47
chirloute-4.lille.grid5000.fr	6004	48
chirloute-4.lille.grid5000.fr	6005	49
chirloute-4.lille.grid5000.fr	6006	50
chirloute-4.lille.grid5000.fr	6007	51

Figure 5.8: Example of *peers.txt* file obtained from the list of reserved nodes, along with peers IDs

5.2.4 The Flow-Shop Scheduling Problem

Our Peer-to-Peer approach uses the interval based-encoding of work units proposed in [Mezmaz 2007a]. Such encoding requires to know the total number of potential solutions (the right value of the initial interval to be explored). Permutation-based problems are good candidates as they meet such requirements. Indeed, for a permutation problem of size n, the initial interval is $[0, n![$. Therefore, in our experimental study, we have considered the Flow-Shop Scheduling permutation-based Problem.

The different variants [Abadi 2000, Allahverdi 2004, Reddi 1972, Bonney 1976, KING 1980, Gangadharan 1993, Röck 1984, Hall 1996, Abadi 2000, Andrés 2005] of the Flow-Shop Scheduling Problem include two main elements: (i) a set of M machines and (ii) a set of N jobs that are to be processed on these machines. Each job must be scheduled on each machine as a whole entity, implying no job splitting. Furthermore, each machine can process only one job at a time and each job is processed only once on each machine. Operations are not preemptable and set-up times of operations are independent of the sequences and therefore can be included in the processing time. The main goal of the Flow-Shop Scheduling Problem is to specify the order and timing of the jobs on the machines, satisfying at most the objective criteria and respecting the mentioned constraints. In our experiments, we focus only on one objective function which is called *makespan*, that is the total processing time, the elapsed time between the beginning of the first job on the first machine and the ending of the last job on the last machine. In our experiments, we focus on a classical variant of the FSP, named *permutation Flow-Shop Scheduling Problem*, where the N jobs must be processed in the same order by each of the M machines [Reza Hejazi 2005],[Allahverdi 1999]. In Figure 5.9, we represent an example of a Permutation Flow-Shop Scheduling Problem, where 4 jobs have to be scheduled on 3 machines. Each job, labeled from J_1 to J_4, satisfies the above-mentioned criteria and the total makespan is the parameter which has to be minimized.

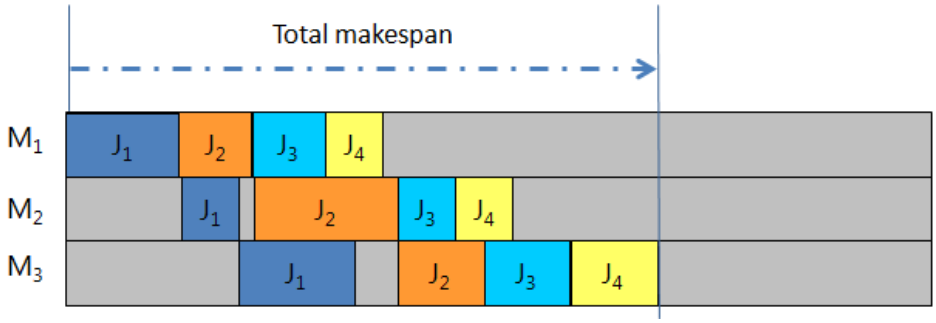

Figure 5.9: Instance of a Permutation FSP: 4 jobs on 3 machines

5.2.5 Definition of Experimental Measures

Before evaluating the performances of our approach, we give some definitions of the experimental measures we will perform as well as the data being collected during

the experiments.

- **Parallel Efficiency (PE):** When a computational entity performs calculations in a parallel environment, it performs two main tasks during its entire execution: it calculates and communicates with other entities. In the case of our P2P B&B algorithm, a peer explores solutions and communicates with other peers for load balancing purposes or for sharing the best solution found so far. Thus, to ensure a good scalability for our approach, the time spent to exploring solutions must be predominant and maximal. We define a measure named *Parallel Efficiency* (PE) which is calculated as the ratio of the time spent to exploring solutions (T_{expl}) over the total execution time (which includes the time spent to communications (T_{com})): $PE = \frac{T_{expl}}{T_{expl}+T_{com}}$. The higher this ratio is, the better our P2P approach scales up.

- **Messages Exchanged:** In order to deepen our measure of scalability, we focus on the amount of communications over the network. This measure consists in the sum of all the messages each peer has sent to its neighbors, whatever the reason is: for communicating a better solution to other peers, for requesting a new work unit or for termination detection purposes.

- **Speed-up:** This measure intends to provide a more intrinsic view of the performances of our approach. Indeed, the two first measures aforementioned provide an indication on the amount of communication of the network. Here, we determine if our approach takes efficiently advantage of the parallelism of the environment, that is, if the size of the network increases, do the raw performances of the network increase as well ?
  When solving entirely an instance, this measure is done the following way: For a given number of peers (N) and for a given problem instance, we compare the Resolution Time for the N peers (t_N) with the time taken by one peer to solve the same instance (t_1) and we calculate the ratio ($\frac{t_N}{t_1}$). As peers need to spend time to communication and termination detection purposes, one can reasonably expect that this ratio remains under N. When experiments are conducted on large instances where the Resolution Time is very important and the duration of the experiment is fixed, the measure considers the number of solutions explored. For a given number of peers (N) and during a given period of time, we compare the number of solutions explored by the N peers (s_N) with the number of solutions explored by one peer during the same period of time (s_1) and we calculate the ratio ($\frac{s_N}{s_1}$).

5.3 Small/Medium-Scale Experiments

In this section, we conduct an experimental study in order to validate our approach. It is conducted according to the *Real-case peer deployment* presented in Section 5.2.2.1. Results are obtained on instances of the Flow-Shop Scheduling Problem,

ranging from Ta021 to Ta030, aiming at scheduling 20 jobs on 20 machines. These instances allow us to study the performances of our approach along the entire resolution process. Indeed, as results will show, they can be solved entirely from scratch within less than one hour, which allows to perform a larger number of runs and deepen our study. In Figure 5.10 , we indicate the sequential resolution times for the instances Ta021 to Ta030 obtained by using a single core of an Intel Xeon E5520 processor.

Instance	Resolution Time (Seconds)
Ta021	98056
Ta022	120806
Ta023	384300
Ta024	93544
Ta025	376094
Ta026	18202
Ta027	165082
Ta028	31444
Ta029	97699
Ta030	17257

Figure 5.10: Sequential resolution times for instances Ta021-Ta030

Performances are to be analyzed using the aforementioned experimental measures. As mentioned in Section 3.5, our approach performs at best using an adequate network overlay, having small-world properties. Thus, our P2P approach is deployed using a toric hypercube topology. We remind that both the diameter of the graph and the average degree can be guaranteed to be order of $\log(n)$ with n being the number of vertices.

The characteristics of the physical machines used in our experiments are described in Figure 5.11.

Site	Cluster	Quantity	Total of cores
Bordeaux	bordeplage	48	96
	bordereau	45	180
	borderline	3	24
Grenoble	edel	38	304
Rennes	paradent	31	248
	parapide	19	152
3 Sites	Total	184	1004

Figure 5.11: Clusters used in our Small/Medium Scale Experiments

5.3.1 Experimental Measures

5.3.1.1 Parallel Efficiency

In this section, we focus on the Parallel Efficiency rate (following the same definition as previously) for our approach. In Figure 5.12, it appears that the peers in our P2P approach, spend most of their time to computing solutions. This shows that, whereas the protocol is more complex in terms of time spent to communication, the subsequent "load" seems to be efficiently distributed among all the peers. However, this gain in performance comes at a price, in terms of messages.

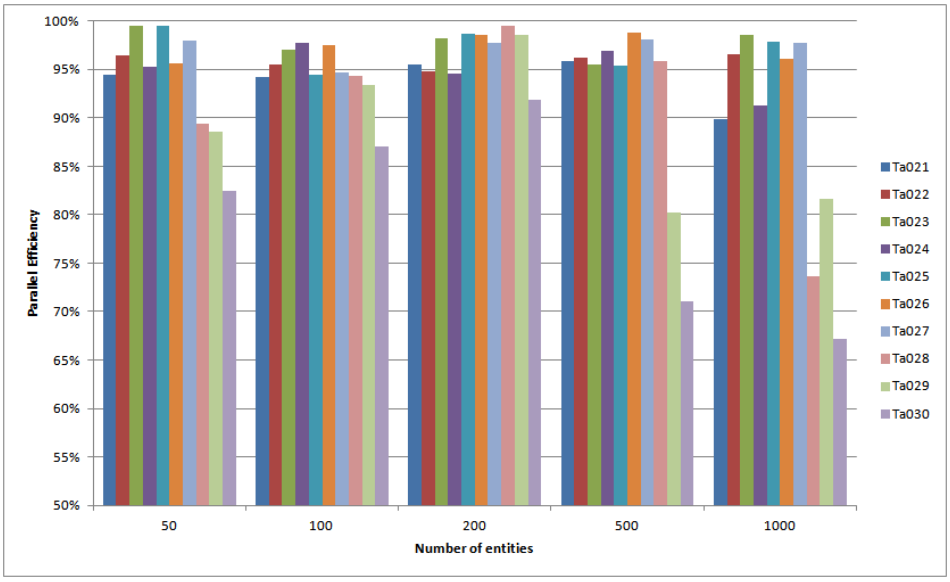

Figure 5.12: Parallel Efficiency ratio when solving small instances with the P2P approach using a toric hypercube topology.

5.3.1.2 Exchanged Messages

In Figure 5.13, we focus on the number of messages which are exchanged between the computational entities during the whole execution. The number of messages sent in the P2P approach is important. This result indicates that, during the execution of the B&B algorithm, the amount of communications generated by our approach's protocol is non-negligible. This result can be expected from the nature of the mechanisms[2] of our approach as each peer has to send waves of messages to its neighbors in order to gather a piece of information.

As the Parallel Efficiency remains high, this dos not impact the performances of our approach at these scales. However, it is important to remind that this communication overhead can become very important at larger scales.

[2]The most message-consuming one being the termination detection mechanism.

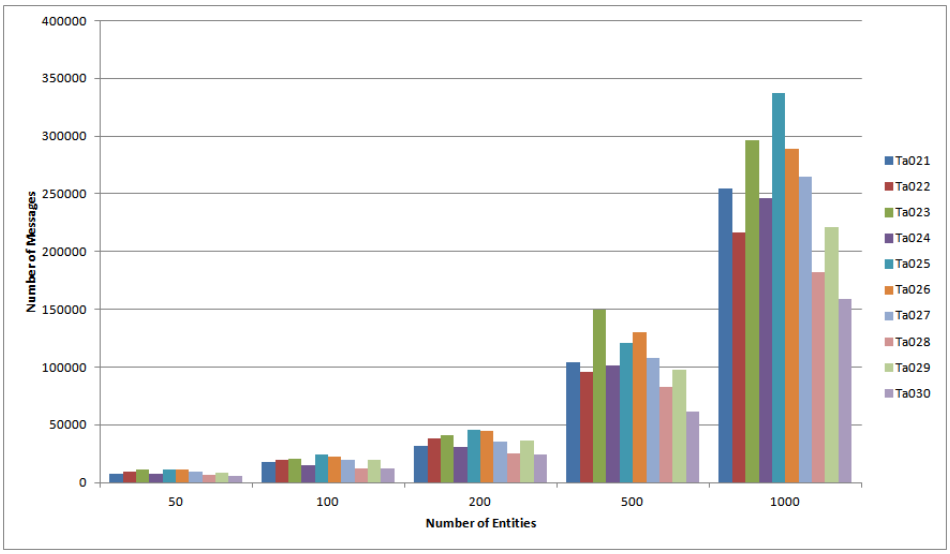

Figure 5.13: Messages exchanged between computational entities when solving small instances with the P2P approach using a hypercube topology.

5.3.1.3 Speed-Up

In Figure 5.14, we study the Speed-Up, according to the number of peers in the network for our P2P approach.

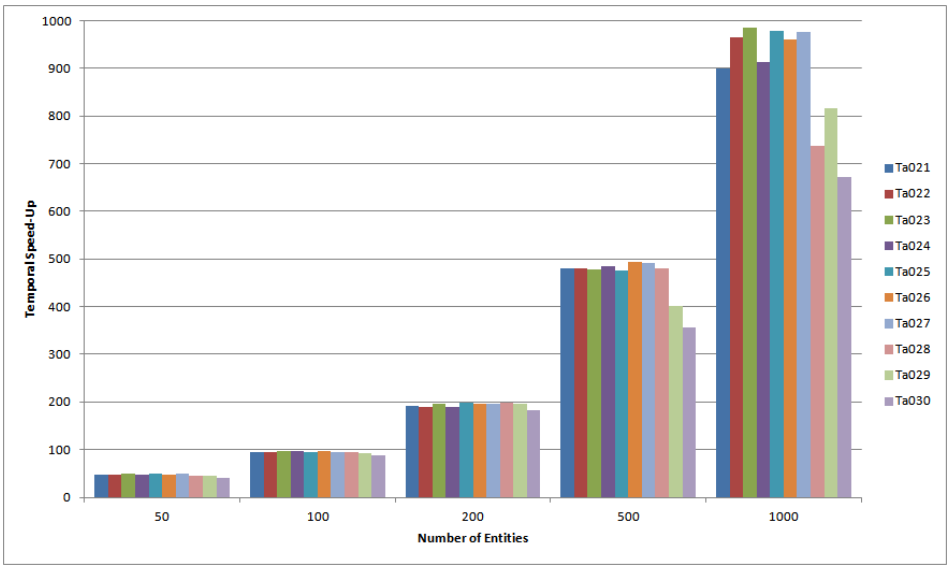

Figure 5.14: Speed-Up for small instances with the P2P approach using a toric hypercube topology.

Results confirm that the ten instances are solved successfully to optimality. One interesting point would be to determine the importance of the termination

detection phase, as the related mechanism is the most *message-consuming* in our approach. Thus, in Figure 5.15, we focus on the Speed-Up by considering the time elapsed between the first peer starting its execution and ending on the last peer starting the work request procedure for the last time i.e. the set of rounds which led to claiming termination.

Linking this result to the Parallel Efficiency in Figure 5.12, the "termination detection phase" represents, in average, about 10% of the total execution time. As it may appear very important when solving small instances, one can reasonably expect this duration to become negligible when solving large instances, that is requiring days or months of computation. In the following section, we deepen our study of the performances of our approach by focusing on the different execution phases along the entire resolution process.

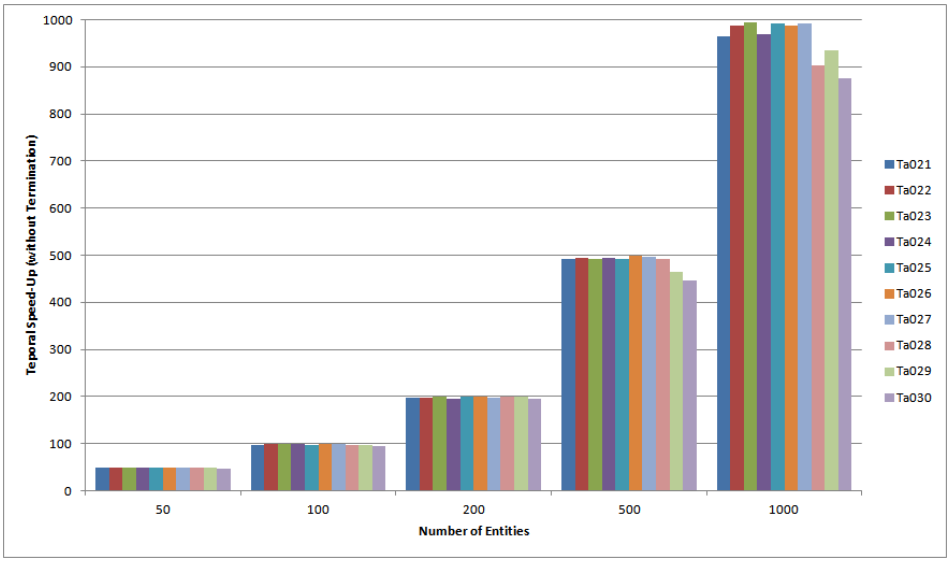

Figure 5.15: Speed-up for small instances with the P2P approach using a toric hypercube topology, without the time taken to detect termination.

5.3.2 Execution Phases

In this section, we propose a more detailed analysis of the performances of our approach by studying their evolution through time. For sake of simplicity, we focus on a single instance of the FSP: Ta025. In the following figures, we present, for 1000 peers, the temporal evolution of the Parallel Efficiency ratio the number of messages exchanged between entities. At first sight, three phases can be distinguished: an Initialization phase, a "Full-Charge" phase and a Termination phase. The following analysis is based on Figures 5.16 and 5.17.

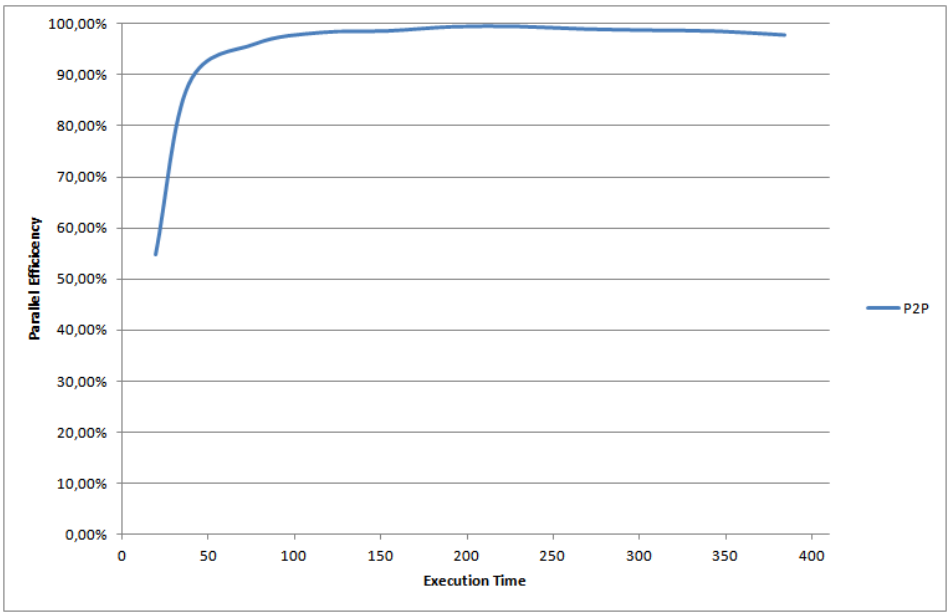

Figure 5.16: Temporal evolution of the Parallel Efficiency ratio when solving the Ta025 instance with the P2P approach involving 1000 peers, using a toric hypercube topology.

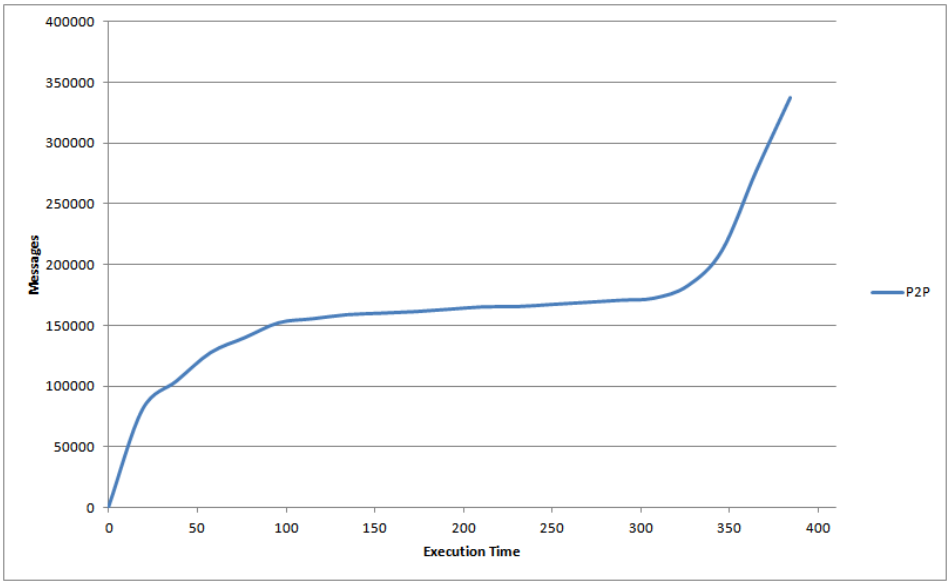

Figure 5.17: Temporal evolution of the number of exchanged messages when solving the Ta025 instance with the P2P approach involving 1000 peers, using a hypercube topology.

5.3.2.1 Initialization phase

In this phase, the computational entities perform work requests to obtain their first work unit. In our P2P approach, after deploying the peers upon the computational cores, one of the peers is designated as the "Initiator", meaning that this peer will hold the whole work unit which has to be processed. Its immediate neighbors will ask for and obtain a work unit from this peer and, then their own neighbors will do the same and so on. Work units somehow "propagate" among the peers until every peer in the network gets a work unit to explore. This process is costly in terms of communication because of the multiple requests performed by the peers and in terms of time also. Indeed, considering the graph representing the P2P network, a peer which is located ad a distance d from the "Initiator" will need to perform at least d rounds of requests to obtain a work unit. This explains why the Parallel Efficiency ratio for the P2P approach rises progressively. Similarly, it allows to understand the fact that the number of exchanged messages increases significantly in the P2P approach.

5.3.2.2 Full-Charge phase

In this second phase, most of the entities in our approach succeeded in obtaining a work unit. As most of the entities are exploring solutions of the FSP problem, the Parallel Efficiency ratio reaches its maximum level. Throughout this phase, some of the entities may finish exploring their work unit and then, ask for a new one. As in the P2P approach, an entity sends requests to multiple other entities, the number of messages exchanged between entities thus rises quickly. However, the subsequent communications are performed in a local fashion which allows the Parallel Efficiency ratio to remain high in our approach. Indeed, as the communication load is distributed among multiple peers, there is no risk of witnessing any bottleneck where an entity finds itself with too many requests to process at a time.

5.3.2.3 Termination phase

In this third phase, the entities spend their time performing work requests in order to detect the presence (or absence) of a work unit somewhere in the network. For the same reasons as for the Initialization phase, this process takes much time in our approach and thus, the Parallel Efficiency ratio decreases.

The analysis of the different execution phases depicts an important property. The P2P approach has a long initialization phase and a long termination detection phase. Thus, when solving very small instances or deploying very low-scale networks, we can clearly imagine that these phases become predominant over the full-charge phase. We can then conclude that, based on previously obtained results, our approach is better suited for extreme scale environments.

5.4 Large-Scale Experiments

In this section, we experiment our approach at larger scales ranging from 2.000 to 150.000 peers. All results are thus obtained using the "Simulation Protocol" described previously in Section 5.2.2.2. The Grid'5000 experimental-bench is composed of 2046 computational cores distributed over 8 geographical sites. The characteristics of the physical machines used in our Large-Scale Experiments are described in Figure 5.18.

Site	Cluster	Quantity	Total of cores
Lille	chinqchint	30	240
	chti	10	20
	chicon	20	40
	chuque	8	16
Bordeaux	bordereau	34	136
Grenoble	genepi	29	232
	edel	53	424
Sophia	azur	49	98
	sol	26	104
	suno	8	64
Orsay	gdx	156	312
Rennes	paravent	48	96
Lyon	capricorne	34	68
Toulouse	pastel	34	136
8 Sites	Total	569	2046

Figure 5.18: Clusters used in our Large-Scale Experiments

We also use $Ta057$ as input instance. This instance is in fact a typical instance that one would like to solve using a huge amount of computing resources. It consists in scheduling 50 jobs over 20 machines. Solving to optimality this instance is out of the scope of this study. In our experiments, we choose to set the running time of *each* entity to 10 minutes. One may think at first sight that this value may be low. However, this value is relevant as medium scale instances (e.g., 20 jobs and 20 machines) can be solved within this amount of time with a low number of cores (In our experiments in Section 5.3, we solved the instance Ta026 within 11 minutes, using 500 cores). Moreover, since we will consider P2P networks managing several thousands of entities, the overall running time from the B&B optimization point of view is still huge. Secondly, considering such a value allows to conduct a larger number of tests within a reasonable amount of time and thus to better understand the behavior of our approach.

5.4.1 Comparison between P2P (toric hypercube) and Master-Slave approaches

The following set of experiments is intended to show the gain in scalability of our fully decentralized approach compared to a Master-Slave approach. More precisely,

we report the performance of our Peer-to-Peer approach compared the Mezmaz *et al.*'s Master-Slave approach [Mezmaz 2007b].

5.4.1.1 Parallel Efficiency

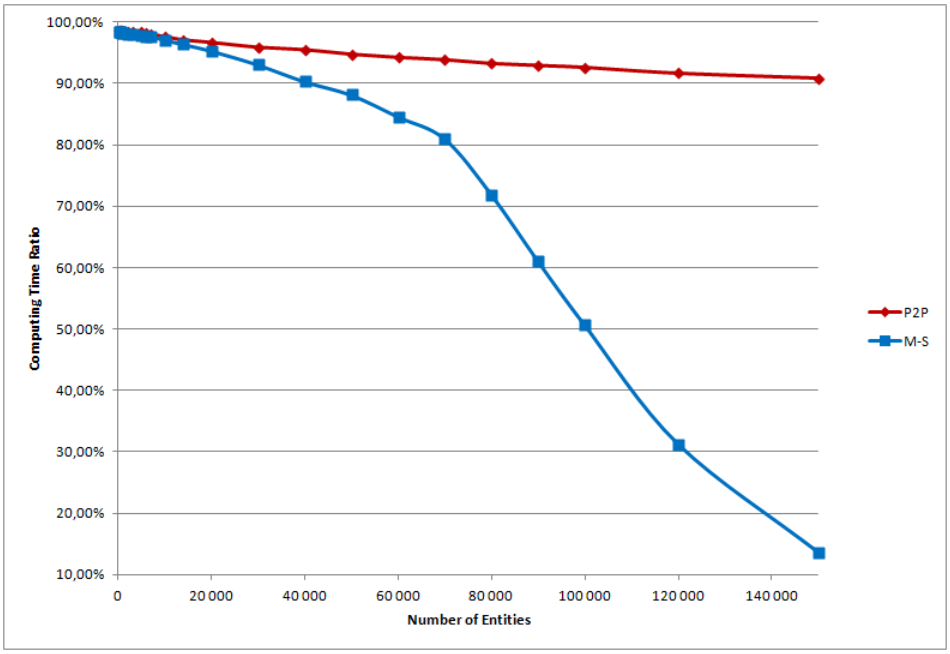

Figure 5.19: Parallel Efficiency rates with the number of entities.

As pointed earlier, the main drawback of a parallel application designed upon a Master-Slave model is the use of a central entity which coordinates all other entities in the network. In a large-scale scenario, this central entity, the Master, has to process an increasing number of requests from slave entities. Consequently, the mean delay taken to process a work request gets longer and the overall performance of the network degrades. In Figure 5.19, we depict the Parallel Efficiency (PE for short) rate along with the number of entities involved in the network. The Parallel Efficiency is obtained by comparing, for each computational entity, the amount of time spent on performing the B&B algorithm (i.e. exploring solutions in the search space) to the overall execution time. Around about 60.000 entities involved in the network, the PE of the Master-Worker approach begins to drop significantly. In addition, around about 70.000 and 80.000 entities, the Master-Slave curve does not go down as smoothly as for smaller values and breaks down quickly to reach considerably low levels when involving more than 100.000 entities (around 10%). This was expected in theory due to the increasing synchronization operations and the increasing computational load on the Master entity. On the opposite, in our Peer-to-Peer approach, work requests are processed locally since a peer only communicates with its immediate neighbors. In fact, the load is somehow "distributed" among the peers. Thus, the mean delay for processing a request is

kept low. This is why the PE rate for the P2P approach remains high, namely above 90%, even for large scale-simulations, involving more than 100.000 entities. However, one may legitimately wonder if the gain in terms of Parallel Efficiency also exists while evolving at very low scales. The graph provided in Figure5.20 confirms this idea although the numerical difference is quite low.

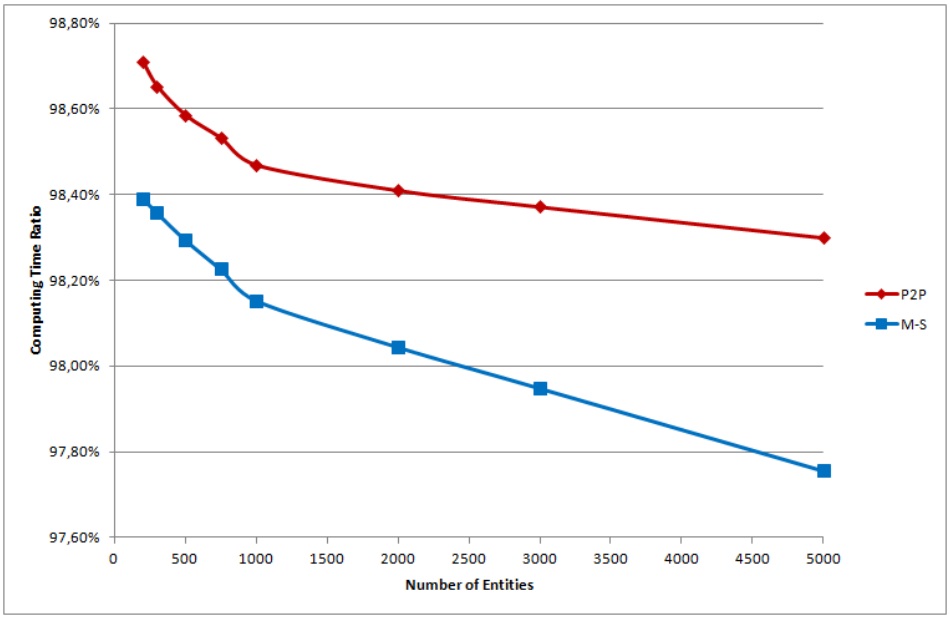

Figure 5.20: Parallel Efficiency rates for small-scale networks.

5.4.1.2 Exchanged Messages

In Figure 5.21 (red/bottom curve with the left Y axis), we depict the ratio of the number of messages sent by all processes (including local communications on the same physical machine) in our P2P approach and in the Master-Slave approach. One can see that for our P2P approach, peers send about 11 times more messages than the Master-Slave approach. In Figure 5.21 (blue/top curve with the right Y axis), we can also see that the *experimental* message overhead factor is significantly lower than the ratio that could be expected according to the complexity analysis presented earlier. Indeed, one can see that as the experimental overhead factor remains between 9 and 11, the theoretical overhead (a very worst-case scenario) expected according to the constructed logical topology, is much higher, namely starting from 20 and rising up to 75. We thus can state that the theoretical analysis of Section 3.5 is even pessimistic compared to what happens in practice. Hence, we can conclude that the message overhead cost we paid in order to design our fully distributed scheme is perfectly negligible from a Parallel Efficiency point of view since compared to a Master-Slave approach where the master becomes a communication bottleneck, communication load is evenly shared among peers.

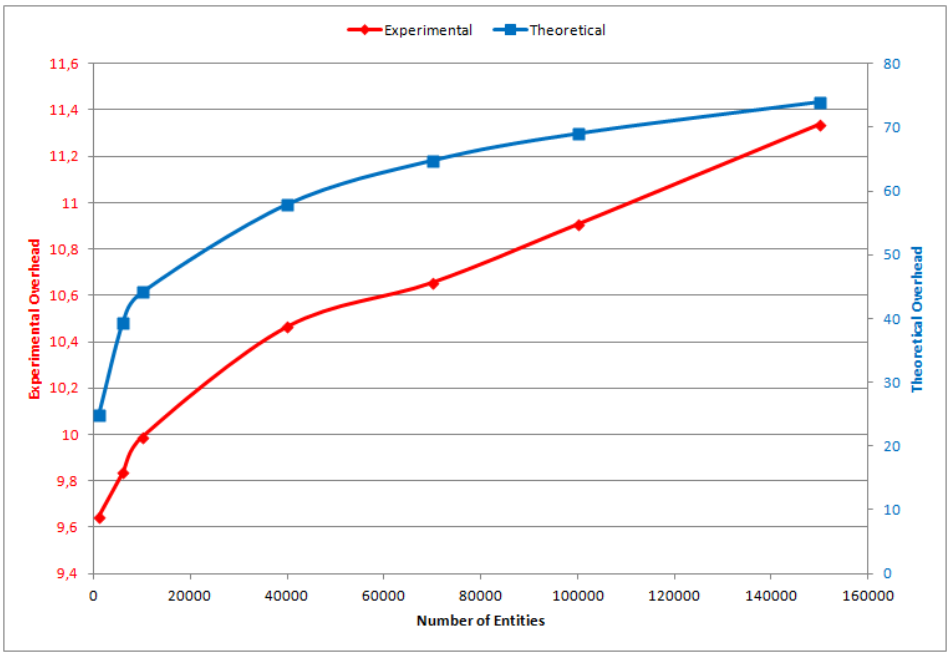

Figure 5.21: Messages Overhead

5.4.2 Network Congestion and Simulation Scenarios

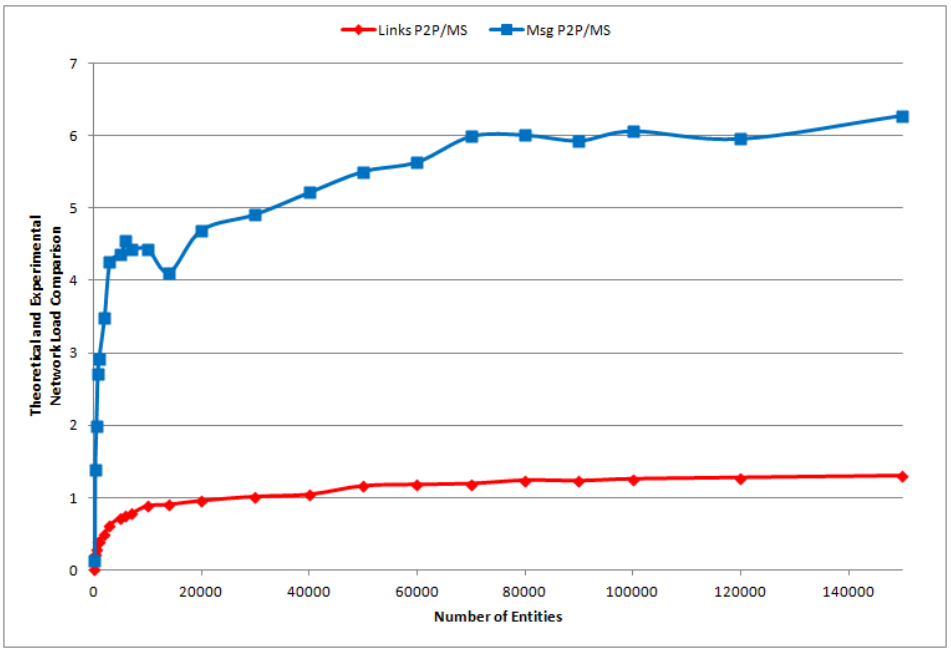

Figure 5.22: Theoretical and Experimental Network Load Ratios

While performing large-scale simulations, as the number of processes increases,

the amount of communications made through the physical network increases. Thus, for such a simulation to make sense, it is crucial to make sure that the network does not undergo congestion/overload. As this possible congestion will be due to massive network operations, we have to focus on whether or not a computational entity involved in our simulations (the process that simulates a peer) uses the physical network to communicate with another entity. Indeed, since the number of physical machines remains constant throughout our experiments, the number of processes deployed on each machine gets higher and higher. Thus, it may happen that two processes communicating with one another are located on the same physical machine. In this case, communications remain local and do not make use of the physical network. In Figure 5.22, we report two important measures:

- The ratio $r_1 = \frac{L_{P2P}}{L_{M-S}}$ where L_{P2P} (resp. L_{M-S}) is the number of logical links connecting two nodes (processes) deployed on two different physical machines in the P2P (resp. Master-Slave) approach. (See the red/bottom curve).
- The ratio $r_2 = \frac{Msg_{P2P}}{Msg_{M-S}}$ where Msg_{P2P} (resp. Msg_{M-S}) is the number of messages exchanged through logical links connecting two processes deployed on two different physical machines in the P2P (resp. Master-Slave) approach. (See the blue/top curve).

In a Master-Slave architecture, it can be easily predicted that most of processes will use the physical network as they will communicate with the same central entity (Master process). Consequently, we pay attention to deploy our P2P approach in such a way the physical network is also used in the same manner than in the Master-Slave architecture. More precisely, we have deployed the peers in a deterministic manner as described in Section 5.2.3 . After conducting the experiment, we verified that the previous property is satisfied for every simulation run. In [Nguyen 2012], this issue is handled in an explicit way. The overlay is defined after deploying the peers on the reserved machines by using a metric based on the IP addresses. Peers running on a same cluster have great chances to be neighbors in the overlay. In our experiments, this locality is obtained by deploying on the machines in the order they appear when collecting the list of reserved machines. Indeed, machines belonging to the same cluster are contiguous in the list and clusters belonging to the same site are also contiguous in the list.

In Figure 5.22, one can in fact see that the network is expected to be used by our P2P overlay topology in the same way than by the Master-Slave star overlay. In other words, this indicates that, in our simulation scenarios, none of the two approaches is penalized by network congestion issues than the other.

Furthermore, when looking at the number of messages sent over the physical network (blue curve in Figure 5.22), we see that our P2P approach costs more messages than the Master-Slave approach. Thus, we can reasonably state that the PE of the Master-Slave approach goes down because of the bottleneck created around the Master and not due to network congestion issues. In opposite, our P2P

Parallel Efficiency stays good even for large scale simulations using intensively the physical network. In fact, although the network is solicited up to 6.5 times more in the P2P approach, the Parallel Efficiency rate remains high, namely above 90%.

5.4.2.1 Search Space Exploration Speed-Up

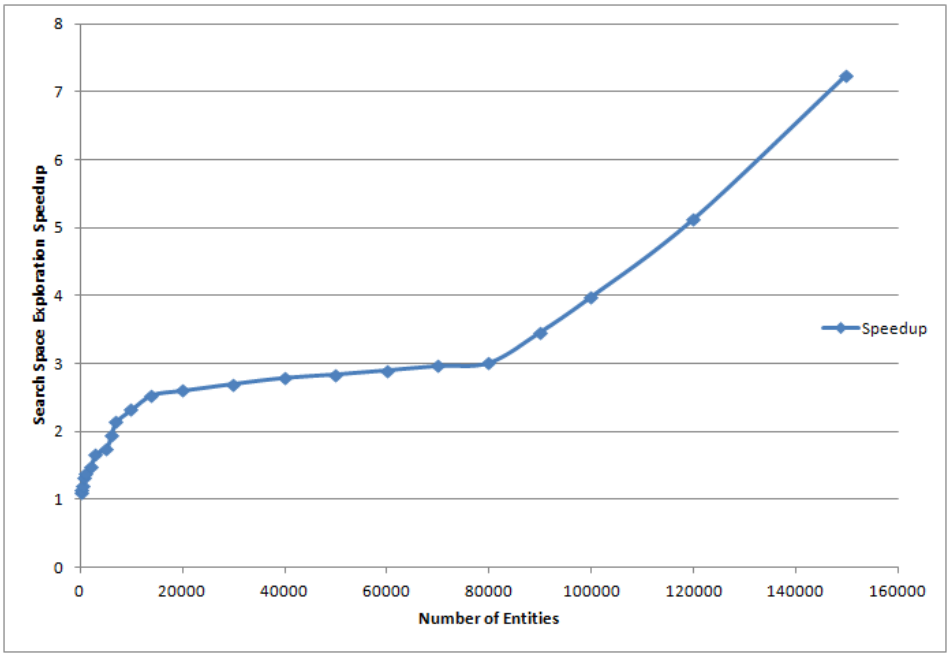

Figure 5.23: Speed-up, in terms of solutions explored.

In previous sections, we showed that the loss of performance of the Master-Slave approach is not due to a possible network congestion as our P2P approach induces a network load several times more important in terms of messages exchanged, without losing in Parallel Efficiency. However, designing an efficient and scalable parallel technique avoiding network congestion and handling more computational resources may not be sufficient to claim that, dealing with combinatorial optimization problems, it enables to explore the B&B search space efficiently. In Figure 5.23, we depict the ratio of the number of solutions of the search space explored by each computational node in our approach and in the Master-Slave approach. One can see that our fully distributed approach explores up to 7 times more solutions than the Master-Slave approach of [Mezmaz 2005]. To understand the obtained curve, we should keep in mind that the space search speed-up is guided by two factors: the degree of parallelism (PE as depicted in Figure 5.19) and the B&B pruning it-self which is theoretically unpredictable. Observed at low scales, the communication cost in both approaches is too low to have a significant impact on the global search speed up. In fact, we claim that at low scales the speed up is mostly dominated by the pruning since PE of both approaches are comparable with of course still an advantage for our P2P approach. However, when involving about 70000 computa-

tional nodes, the PE of the Master-Slave model is no longer competitive and it starts dropping significantly as described previously in Figure 5.19. Hence, we observe a breakpoint around 80000 entities in Figure 5.23. This breakpoint can be interpreted as the network size where the impact of PE on search speed-up becomes critical and the pruning cannot balance the loss in PE. We can thus state the gain in terms of Parallel Efficiency of our P2P approach has a major impact when involving large scale deployment of parallel B&B experiences on large instances.

5.4.2.2 Speed-up

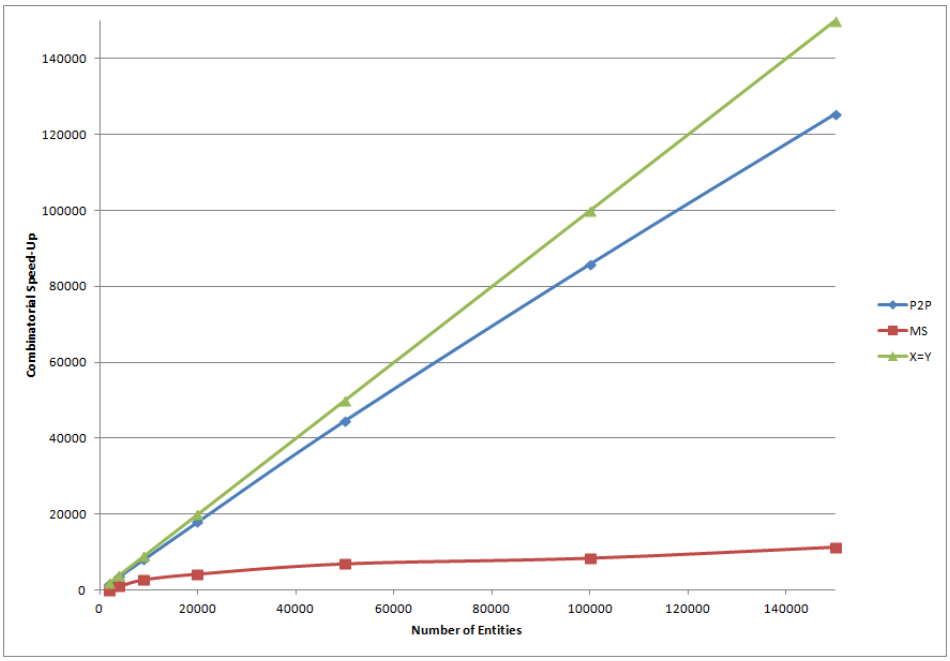

Figure 5.24: Speed-up, in terms of solutions explored.

In Figure 5.24, for a given number N of entities in the network, we compute the ratio $\frac{s_N}{s_1}$ where s_N represents the number of nodes explored in the search tree by the N entities and s_1 the same number but for only one entity. For the sake of clarity, we represented the case of a linear speed-up. We can see that our P2P approach has a good ratio, close to the linear one, whereas the Master-Slave ratio remains very low. The Master-Slave approach is not able to take advantage of an increasing amount of computational resources because of the communication bottleneck around the Master entity. At the opposite, the speedup observed for the P2P approach is up to 130.000 when involving 150.000 entities (while the Master-Slave approach reaches a ratio of 12.000).

5.5 Impact of overlay topology

Up to now, we studied our approach by focusing on a particular topology: a toric hypercube. As mentioned before, this topology was chosen because of its small-world properties: it has a low diameter, which reduces the time needed to broadcast global information (the best solution found) and a low average node degree, which reduces the number of messages required to perform communication operations. However, one can reasonably wonder how would our approach perform if the overlay network was organized in a completely different manner. In this section, we push on our analysis by studying the impact of different overlays on the performances of our approach. More precisely, in addition to the N-dimensional toric hypercube, topology described previously, we further experiment the following classical overlays: Kleinberg small world graphs [Kleinberg 2000a, Kleinberg 2000b] (regular lattice), Barabasì-Albert graphs[Barabasi 1999] (preferential attachment graph, i.e. Internet-like graph), Watts-Strogatz graphs [Watts 1998] (ring graph where each node is connected to its k closest neighbors where k is a given parameter). We also select more conventional topologies such as the 5-ary tree and random graphs (Erdos-Reyni graphs [Erdös 1959, Erdos 1960]).

5.5.1 Overlay definitions

The overlay topologies that we shall experiment in the remainder are the following (we will interchangeably use vertices to refer to peers and edges to refers to links between peers). These topologies have been selected because their properties in terms of graph diameter and node degree vary importantly from one another.

Watts-Strogatz Graphs: In [Watts 1998, Barrat 2000], the topology is built from a regular pattern with *"randomness"* introduced afterwards. More precisely, the construction process of the overlay relies on three parameters, namely: N, k and p. Initially, the vertices are organized according to a ring. Edges are added such that each of the N vertices is linked to the k closest vertices (See Fig 5.25). Then, for each edge of the graph, one of the vertices is modified with a probability p (See Fig 5.26). This topology presents a high average node degree, a low diameter du to the "random" edges and a very low standard deviation for the degree. One can expect this topology to consume a lot of messages while reducing the risk of communication bottlenecks.

K-ary trees: This well-known topology is built as following. One node is considered as the *root* of the tree and every node is given, iteratively, k *children* to which it is linked to. This process is repeated until the desired number N of nodes is reached. As this topology has a low average node degree and a low diameter, it can be expected to induce a very low communication overhead.

Kleinberg: In [Kleinberg 2000a, Kleinberg 2000b], the topology is built starting

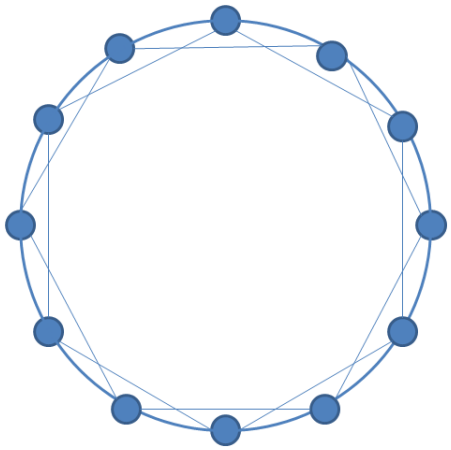

Figure 5.25: Watts-Strogatz topology's initial pattern with $k = 4$ and $N = 12$

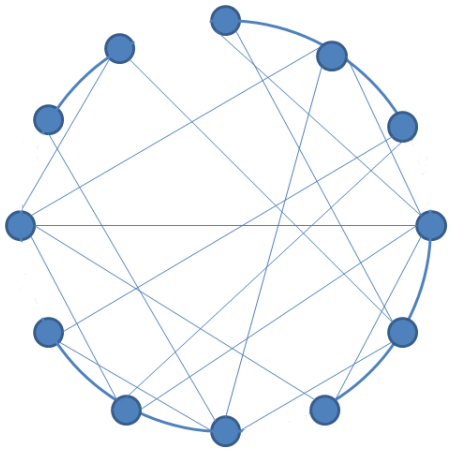

Figure 5.26: Example of final Watts-Strogatz pattern obtained with $k = 4$ and $N = 12$ and a value of $p \in]0, 1[$

from a two-dimensional grid network. The main goal is to artificially reduce the graph diameter by introducing additional edges between distant vertices. More precisely, the initial edges of the two-dimensional grid are called "*short-range*" links. Then, the *lattice distance* is defined as the number of edges comprised in the shortest path between two vertices of the graph, i.e., $d(u, v)$ for two vertices (u, v). Given a constant $r \geq 0$ and for every pair of vertices (u', v'), a new edge[3] is added with a probability equal to $\frac{[d(u',v')]^{-r}}{\sum_{v' \in V} [d(u',v')]^{-r}}$. (See Figure 5.27). This probability law is referred as the *inverse* r^{th}*- power distribution*. In terms of graph diameter and node degrees, this topology is similar to the Klienberg one, while the standard deviation of the nodes degree is higher, thus potentially increasing the

[3]The newly introduced edges are named "*long-range*" links.

communication load on some nodes.

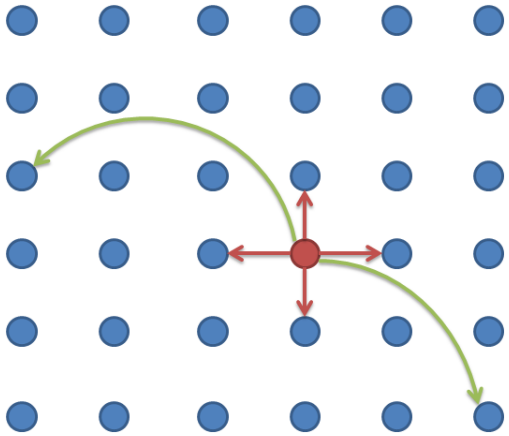

Figure 5.27: Example of a resulting Kleinberg topology with red edges as *short-range* links and green edges as *long-range* links.

Barabasi-Albert: [Barabasi 1999] This topology is inspired by the Internet network with network properties close to small-world graphs. Indeed, a reduced number of nodes hosts contents (like web pages, videos, databases...) whereas a greater number of visitors connect to these nodes in order to access these contents (See Fig 5.28). This type of graphs are called *preferential graphs*. The main idea behind designing such graphs is that, assuming the existence of a graph $G = (V, E)$, if one chooses to introduce a new vertex, it has a higher probability to be linked to a vertex with a higher degree. More analytically, if for a vertex $u \in V$, we call k_u its degree, then a new vertex $v \in V$ will be linked to u with a probability equal to $\frac{k_u}{\sum_{l \in V} k_l}$. If $\forall u \in V$, $k_u = 0$ then edges are created randomly. The main interest of this topology is that, although it has a high diameter and a low average node degree, the standard deviation for the degrees is very high, meaning that some of the nodes may centralize most of the communications in the network.

Erdos-Reyni: [Erdös 1959, Erdos 1960] This consists in a classical randomly generated graph. They generate their random graphs using two different methods. The method used here to generate such graphs consists in including each one of the $\binom{N}{2} = \frac{N \times (N-1)}{2}$ possible edges with a constant probability p, the presence or absence of any two distinct edges being independent. This topology has a low diameter but a high average node degree.

The topologies presented in this section have very different properties in terms of graph diameter, average node degree and standard deviation for the degrees. Each one of these topologies represents a particular scenario for our experimental study as it allows to influence significantly the time needed to broadcast global information,

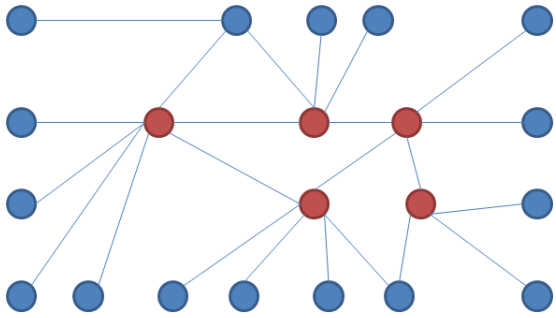

Figure 5.28: Example of a resulting Barabasi-Albert topology with red vertices as *preferential vertices* due to their higher degree.

the number of messages exchanged in the network as well as the distribution of the communication load among the peers. In the following subsection, we analyze the performances of our approach, using the same measures as before, on these topologies. We also show indicative results for the hypercube topology and a Star-like one.

5.5.2 Results

5.5.2.1 Parallel efficiency

Figure 5.29 depicts the Parallel Efficiency along with the number of entities involved in the network. Most of the topologies have a Parallel Efficiency rate that remains high when the scale of the network increases, namely a PE rate comprised between 80% and 90% when involving 150.000 computational entities in the network. The Barabasì-Albert and the Star topologies have a PE rate that decreases with the scale of the network. Around 60.000 entities in the network, the PE rates begin to drop. When using 150.000 computational entities, this rate reaches around 55% for the Barabasì-Albert topology and less than 10% for the Star topology. This was expected in theory. Indeed, the Star topology centralizes most of the communications around a single entity and similarly, the Barabasì-Albert topology centralizes the communications around several but few entities. Both topologies induce communication bottlenecks around some entities and thus, this slows down the entire network. For the other topologies, the communication load is somehow more efficiently "distributed" among the entities. Thus, the mean delay for processing a request is kept low. This is why the PE rate for the P2P approach remains high, namely above 80%, even for large-scale evaluations, involving more than 100.000 entities.

5.5.2.2 Message Overhead

This measure intends to show how the non-negligible communication overhead on the network affects the performance of the P2P approach. Thus, one has to ensure both that this overhead is kept low and that the drop of performances observed for some

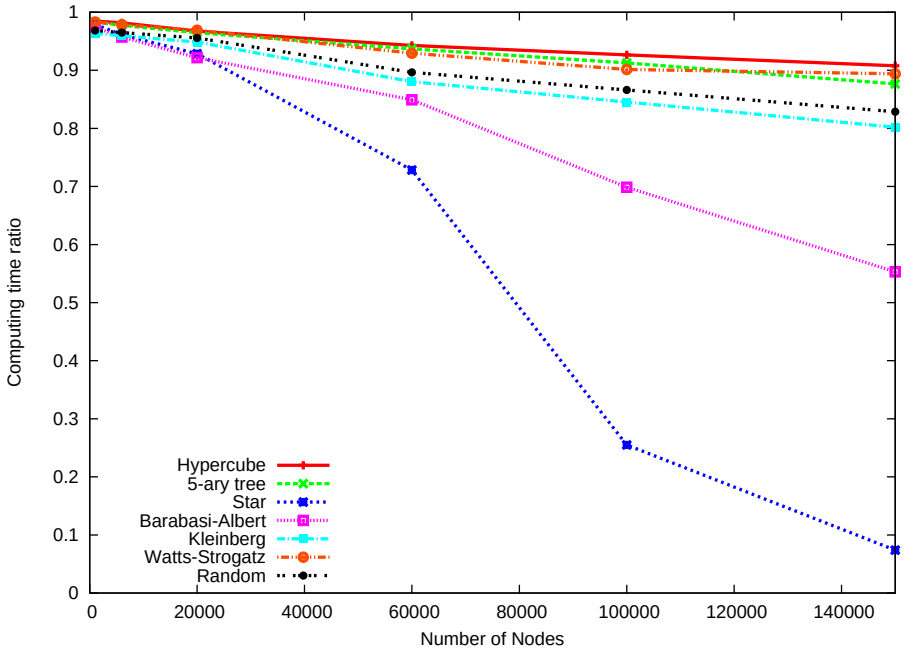

Figure 5.29: Parallel Efficiency

of the topologies is effectively due to communication bottleneck issues around some entities and not to a network congestion caused by this communication overhead. This overhead is computed, for a given topology, as the ratio[4] $\frac{m_{topo}}{m_{Star}}$ where m_{topo} (resp. m_{Star}) represents the number of messages exchanged over the network with the *topo* topology (resp. Star topology) approach during the entire experiment.

Most of the topologies have a communication overhead ratio comprised between 8 and 11. This means that the number of messages exchanged between computational entities with these topologies is between 8 and 11 times higher than with the Star topology. The 5-ary tree induces a reduced overhead, namely, a factor of 3 and the Barabasi-Albert induces a lower overhead, similar to the Star topology. This communication overhead is a phenomenon which is due to the fact that a computing entity communicates with several other entities instead of one in a Master-Slave architecture.

5.5.2.3 Topologies properties and Experimental Performances

According to the data presented in Section 5.5.1, we observe that the Barabasì-Albert and the Star topology both have a quite low diameter, a low average node degree and a high standard deviation for the nodes degree. The low-diameter property of these topologies allows to reduce the number of messages and the amount of time needed to communicate global information over the network (In our case, this global information consists in the best incumbent for the B&B algorithm.).

[4]The obtained result is still an average measure on three separate runs.

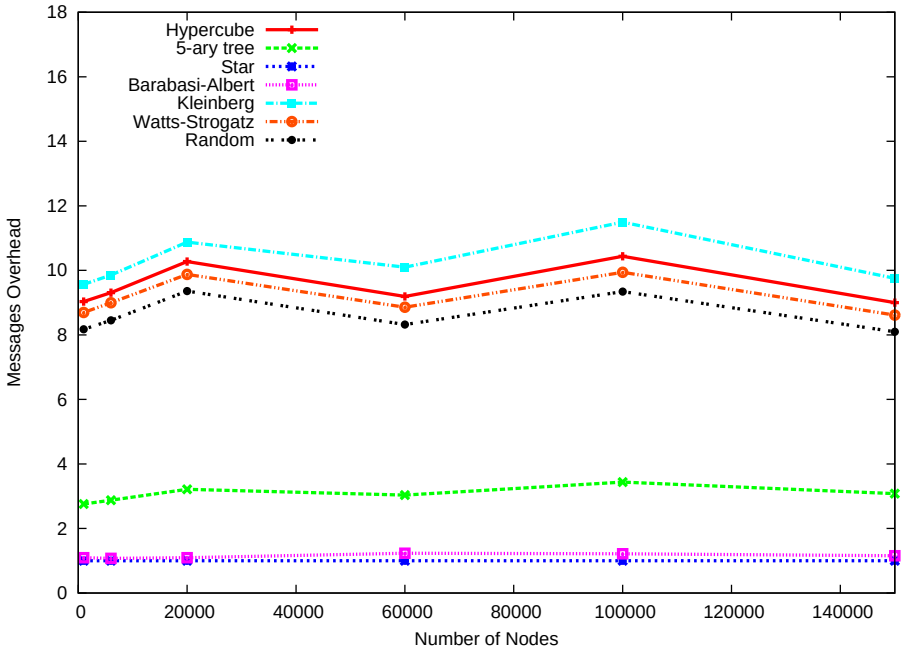

Figure 5.30: Message Overhead

Similarly, the low average node degree of these topologies reduces the number of messages needed to perform work requests over the network as well as to detect computation's termination. This explains why both topologies show a low communication overhead factor (See Figure 5.30). However, these topologies have a distribution of nodes' degree which presents high standard deviation (higher for the Star topology). This implies that many nodes have a low number of neighbors and conversely. Thus, network communications are aggregated around a small group of "high-degree" nodes (several nodes for the Barabasí-Albert topology and one for the Star topology), creating bottlenecks around these nodes. The amount of time spent to communication rises which explains the drop of performances in terms of Parallel Efficiency (See Figure 5.29). What we can conclude at this point is that choosing a small-world graph for the P2P network is not sufficient in order to obtain good performances. Indeed, this reduces the amount of communications over the network but we have to make sure that no bottlecks appear around some nodes. This is done by ensuring that the distribution of nodes' degree has a low standard deviation.

5.5.2.4 Speedup

For a given scale and a given topology, we compute the ratio $\frac{s_{topo}}{s_{Star}}$ where s_{topo} (resp. s_{Star}) represents the total number of nodes in the search tree[5] which have been explored by the entities with the *topo* (resp. Star) topology over the entire running time. As expected, we can see that the topologies allowing a high Parallel Efficiency

[5]Recall that we use the tree-based exploration process proposed by [Mezmaz 2007b].

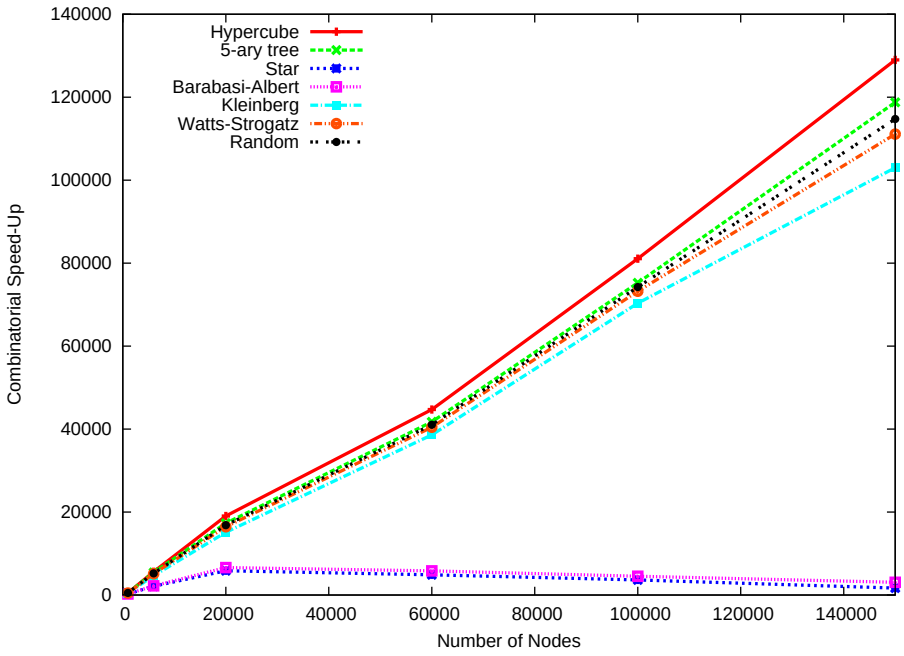

Figure 5.31: Combinatorial Speedup

ratio (See Figure 5.29) provide also a high speed-up ratio. The poor results obtained for the Star topology and the Barabasí-Albert reflect the loss of performance due to the communication bottleneck we described earlier through both experimental results (See the Parallel Efficiency measure) and topology analysis (See Section 5.5.2.3).

In addition, we can observe that the 5-ary tree and the *random* (Erdos-Reyni) topology are among the topologies on which our approach performs at best. These topologies have a simple structure and, thus, can be easily maintained in a *dynamic* environment, where computational nodes can join or leave the network at any time.

5.6 Conclusion

In this chapter, we conducted extensive experiments to provide a detailed and comprehensive study of the performances of our approach. The obtained results show that our approach can efficiently solve difficult Combinatorial Optimization Problems for a range of instances taken from Flow-Shop Scheduling Problem. To study the scalability of our approach, experiments have been conducted at large scales and the results show that the load of communications induced by the use of a fully Peer-to-Peer topology is efficiently distributed among the computational entities even when considering large scales networks, namely up to 150.000 peers. Based on our experiments, a limitation that should be mentioned concerns the cost of the termination detection. The reported results show that it is in average, about 10% for pools of processing cores up to 1000. However, as for very large pools (up to

150.000 cores), the experimented problem instances are not solved to optimality, it is difficult to predict the behavior of such cost at larger scales, even if the involved communications are distributed. On the other hand, in our experiments, we have considered the cluster-based proximity of peers provided by the OAR reservation tool of Grid'5000 to deal with the mapping. Such mapping is quite similar to the IP-based proximity metric used in [Nguyen 2012]. Other distance metrics [Huffaker 2002] can be investigated to improve the efficiency of the mapping.

Chapter 6

Fault-Tolerant B&B-P2P for dynamic environments

Contents

6.1 Introduction

The approach described in previous chapters was designed to work in "*static*" environments where computational resources are assumed to be reliable over time. Dealing with large scale distributed systems, it is however mandatory to also address the fault tolerance issue and to allow the application to recover properly. In particular, for parallel B&B, and to guarantee the *optimality* of the computed solution, it is crucial to ensure that all the search space has been explored without omitting any part of it. The goal of this chapter is precisely to extend our B&B approach so that it can be executed in a dynamic context where peers are subject to failures.

Generally speaking, faults can be of different types. The most common definition of a fault for a system is a malfunction or an unexpected deviation from normal behavior. Faults can thus vary a lot in terms of nature and, for a parallel application, they can be classified into four general classes [Bakken 2002, Tanenbaum 2002] : *Crash*, *Time*, *Omission* and *Byzantine*. *Crash* faults occur when a process completely ceases operating, its execution is permanently stopped. *Time* faults deal with situations where the duration of a given operation becomes longer than usual or expected. Example of these faults is temporary network failures for distributed applications. *Omission* faults designate the case where a process is supposed to perform a task upon a specific event but remains idle. One example could be a network process ignoring an incoming request from a remote process. The last type of faults is *Byzantine*: the output or the result of an operation is altered or corrupted by external causes. In this work, we address *Crash* and *Time* faults.

When extending our fully distributed approach to a dynamic environment where peers can fail, the critical issue is to guarantee that B&B work units are not lost after some nodes have crashed. Tackling this issue in a pure distributed way is a challenging task. Besides the algorithmic difficulty of setting up distributed protocols that can be proved to be correct, one has to keep in mind the possible performance degradation implied by such a design. In fact, our distributed approach was proved to perform good when the overlay topology verifies specific properties in terms of peer degree and network diameter. In addition, it is not clear how termination detection can be handled when faults can imply disconnected overlay components.

In order to extend our approach in a dynamic environment, we can make on two major observations. Firstly, sharing work in a distributed cooperative manner was proved to be extremely efficient in a static context. Thus, one should try to ensure this property for those peers that are not failed. Secondly, managing faults in a fully decentralized way may introduce difficult issues which can be avoided if a central server can be made available to handle them. The idea developed in this chapter is to design a hybrid approach where work units can be shared among peers in a fully distributed way as before, whereas faults are managed by a centralized server supervising the computations and helping peers to recover. As a consequence, we shall be able to design a simple but effective fault-tolerant approach, where work sharing and fault-tolerance are handled tightly but in a separable way. In the rest

of this chapter, we describe this approach in more details. More precisely, Section 6.2.1 provides a detailed description of the structure of our FT approach as well as the mechanisms involved in the Branch-and-Bound process. Section 6.2.2 describes the main steps of the algorithm's execution from a more functional point of view, through the use of state automata.

Extensive experiments are conducted to evaluate the robustness of our approach. These experiments must emphasize one particular point: our approach must be proved to be robust independently from the type or the frequency of failures. To achieve this goal, we evaluate the performances of our approach as well as the impact of failures on it by defining various failure scenarios which can reflect some real-world situations, on an instance of the Flow-Shop Scheduling Problem.

This chapter is organized as follows: Subsection 6.3.1 provides a description of our experimental protocol and methodology as well as the different scenarios. Subsection 6.3.2 consists in experiments conducted on a small-scale network to prove the robustness of our approach by solving completely an instance from scratch, using the various failure scenarios. Subsection 6.3.3 extends the experiments to include a failure model based on peers' lifetime distribution.

6.2 The Proposed Approach

6.2.1 Overview of the Fault-Tolerant Architecture

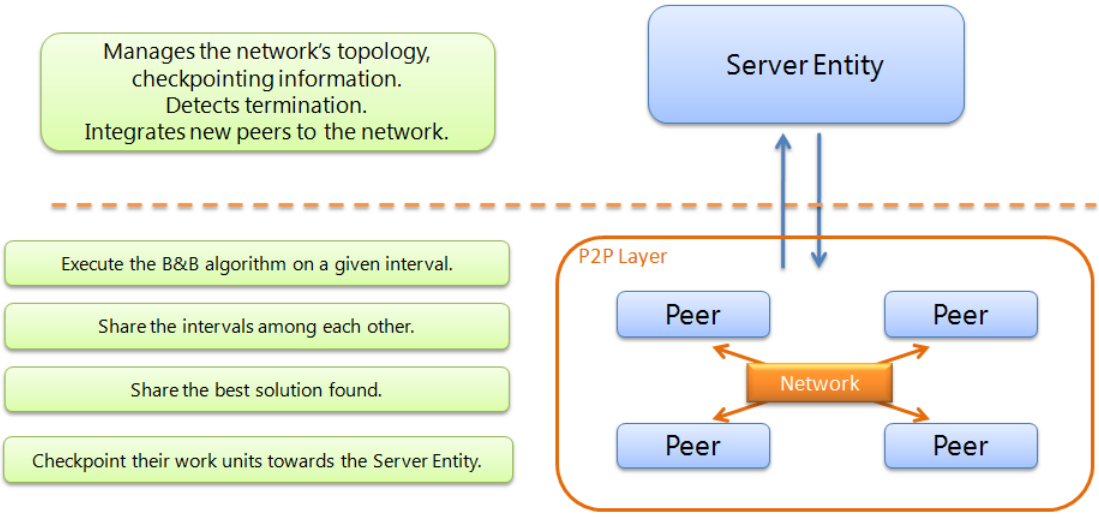

Figure 6.1: The two-layer structure of the Fault-Tolerant Approach.

Our new fault tolerant-approach is build on top of two layers as summarized in Figure 6.1). On the bottom layer, computing nodes are organized in a peer-to-

peer manner. Peers are in charge of exploring the set of feasible solutions for the problem being solved. As discussed in previous chapters, every peer explores its own local pool of B&B intervals and shares it with its local neighbors in the P2P overlay. Whenever a peer discovers a new solution, it forwards this information to its neighbors. On the second upper layer, we introduce a server entity which is dedicated to fault handling. This entity is in charge of recovering local pools of peers in case of failures, maintaining the overlay and detecting termination.

When a peer starts its execution, it first needs to *join* the network. For that purpose, it sends its communication information (its network address and the port it is listening on) to the Server entity. Upon receiving this information, the Server entity becomes aware of the newcomer's existence. From this point, one may notice similarities with the procedure used to initialize the Peer-to-Peer network in a static context. Two cases are distinguished at the Server entity level:

- If the newcomer is the first peer to join the network, the Server entity sends it the whole interval to explore so that the peer may share it with future neighbors.
- In other cases, the newcomer is given a set of peers as neighbors so that the peer may contact them to get a new work unit.

In both situations, the newcomer is informed about the initial/current value of the best upper bound found so far for the problem being solved.

Once the newcomer has joined the network, it starts asking its neighbors for a work unit. Unlike in the static approach in which a peer asks all its neighbors following a one-to-all communication procedure, this is now done in an iterative manner. The main reason for that is to be able to manage neighbors fault properly. More precisely, a peer sends a request iteratively to each one of its neighbors and then waits for a reply until it receives a work unit or until it has processed all its neighbors.

If a work interval is received, the peer starts exploring the newly received interval and, upon receiving requests, shares it with its neighbors. On a regular basis during the processing of this work unit, the peer will perform checkpointing operations towards the Server entity. More precisely, it saves information about the remaining part of the work unit and communicates it to the Server entity. Similarly, when sharing an interval with another peer, the current peer will save information about the work unit it holds and the one it sends to the other peer. However, if the work request procedure fails, the peer sends a request to the Server entity to obtain a new neighborhood, say N neighbors. Upon receiving such a request, the Server entity selects the N peers that performed their checkpointing operations the most recently and sends back their identifiers. Indeed, one can reasonably assume that peers that saved their work unit the most recently have a higher probability of being still alive. Since peers are supposed to perform checkpointing operations every given period of time, if a peer performed its last checkpointing operations more than one period of time ago, the Server entity can presumably assume that this peer is facing

an anomaly, and thus can suspect that the peer has crashed. As a consequence, the server is allowed to reassign the interval of the *suspected failed* peer to another one. In particular, in the case of a peer asking for a new neighborhood, the Server entity is allowed in addition to provide the set of new neighbors, to reassign such an interval (if it exists). To summarize, peers in the network first explore the intervals they acquire from their neighbors in previous rounds, but they can also recover intervals that were possibly held by failed peers and reassigned by the Server entity. As for termination, it becomes easy for the server to detect it since all the peers are checkpointing during the exploration process. When the termination occurs, the peers are informed about it by the Server entity: when a peer fails in obtaining a new work unit and asks a new neighborhood, the Server entity simply sends a special flag. By receiving this message, the peer can determine that no more intervals are explored by other peers and we have the guarantee that no intervals remain from previously failed peers.

6.2.2 Detailed description of the approach

6.2.2.1 Functional description of a Peer and the Server

6.2.2.1.1 Operation of a Peer

In this section, we describe the behavior of a peer along its entire execution for the fault tolerant approach, through the use of a state automata (See Figure 6.2).

Initially, a Peer **initializes** itself, that is it performs any operations needed to be ready to run (like allocating resources or initializing network interface). Once it is ready, it contacts the Server Entity to **register** itself, i.e. to make the Server Entity be aware of its existence. When it receives a new identifier, it becomes **idle**. From this point three situations can occur. If the peer does not have originally any neighbor, it asks the Server entity for a **new neighborhood**, i.e. a new set of peers that it will communicate with. Second, when a peer has neighbors but does not have any interval to explore, it will start **requesting** a work unit from its neighbors. If this request is unsuccessful, it will renew its **neighborhood**, otherwise it starts **exploring** the newly received interval. During the exploration process, a peer may save its state towards the Server entity for fault tolerance purposes, through a **checkpointing** mechanism. Similarly, if it discovers a **new improving solution**, it informs the Server about it and then resumes its exploration. Still during this exploration process, the peer may be contacted by another peer willing to obtain a new work unit to process. Then, the current peer pauses its exploration, **shares** its interval with the requesting peer, informs the Server entity about the exchange through the **checkpointing** mechanism and resumes exploring its interval. Finally, when the peer contacts the Server entity for any of the tasks here-mentioned, it may be informed about the termination of the calculation, i.e. no interval remains in the network. Then, it goes in a **terminating** state and **ends** its execution.

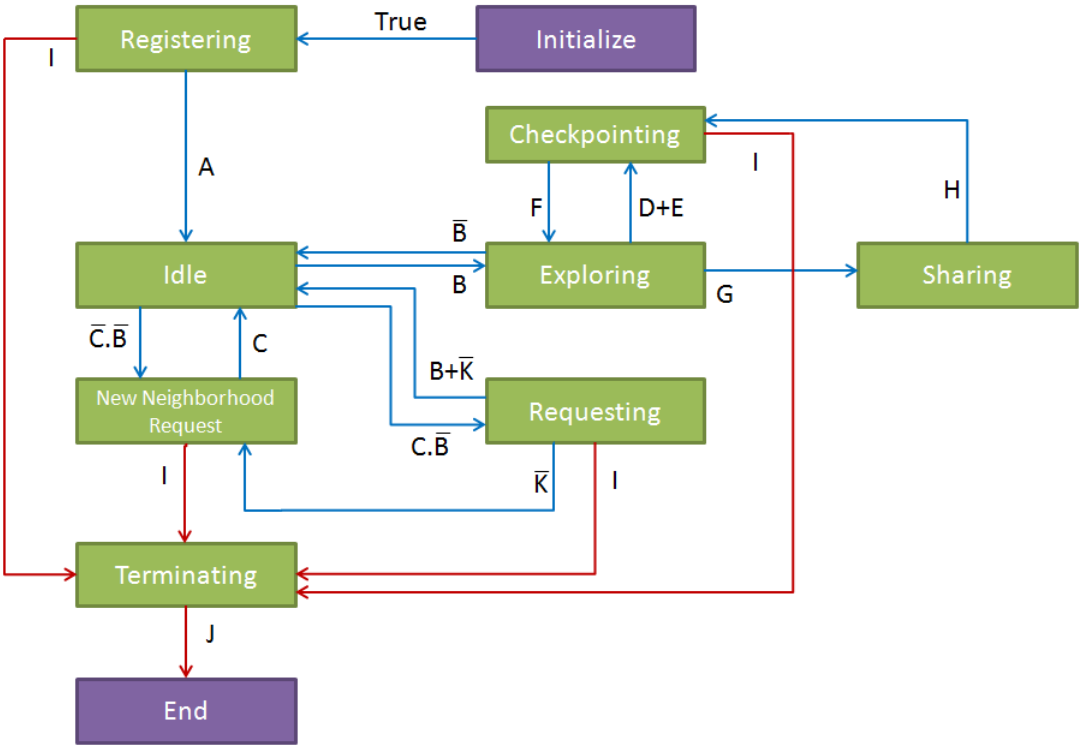

Figure 6.2: State automata representing the behavior of a Peer in our fault-tolerant approach.

Here are the logical propositions related to the state automata in Figure 6.2.

- A = New Identifier Received.
- $B = [x, y] \neq \varnothing$.
- $C = N(v) \neq \varnothing$.
- D = A new improving solution has been discovered.
- E = Checkpointing time period has passed.
- F = Checkpointing operation accomplished.
- G = Received a work request from a neighbor.
- H = Interval shared with a neighbor.
- I = After updating $[x, y]$, termination has been claimed by the Server Entity.
- J = All peers in the network are in *Terminating* state.
- K = No interval is received from neighbors.

6.2.2.1.2 Operation of the Server entity

In this section, we describe the behavior of the Server entity along its entire execution for the fault tolerant approach, through the use of a state automata (See Figure 6.3).

Initially, the Server entity **initializes** itself, that is performs any operations needed to be ready to run (like allocating resources or initializing network interface). Once it is finished, it becomes **idle** and is ready to process incoming requests. During its execution, the Server entity may face three types of requests. First, it can receive a message from a peer requesting a **new neighborhood**. The Server entity then sends a new set of neighbors (the related mechanism being described more in details later) to the requesting peer and resumes processing incoming requests. Second, it may be contacted by a new peer willing to **register** itself. Then, the Server entity will attribute a new identifier to the newcomer and resume its previous task. Third, the Server can process a **checkpointing** request from a peer. It saves the state of the Peer and resumes it previous task. Nevertheless, if a peer updates its interval to the Server, it is possible that no interval remain in the network from the Server's point of view. Then, the Server goes into a **termination** state, meaning that upon any request received from a peer, the Server will inform that peer that calculations are terminated. When all the peers are aware of the termination of the calculations, the Server can **end** its execution.

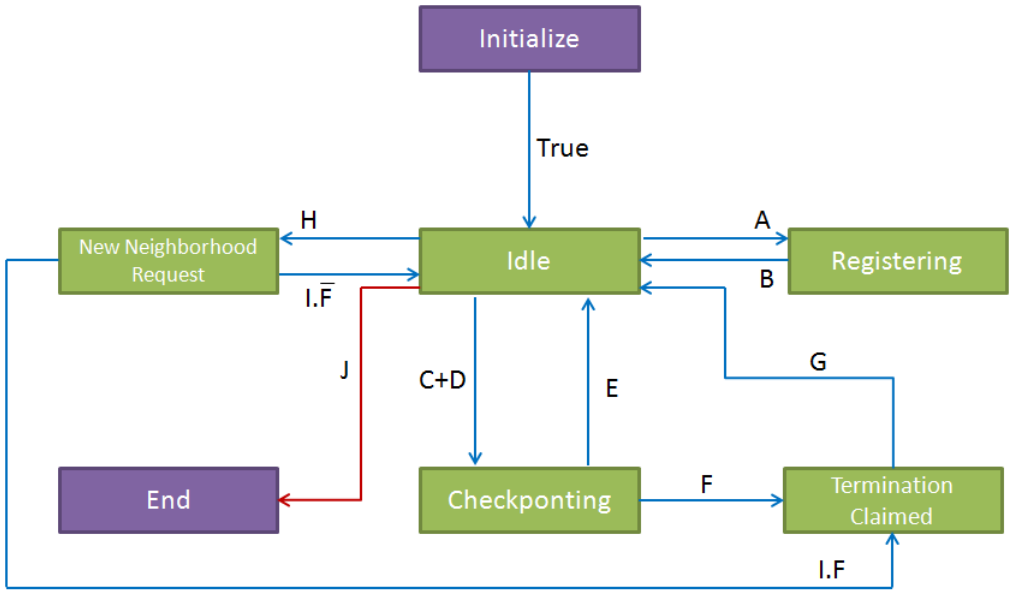

Figure 6.3: State automata representing the behavior of the Server Entity in our fault-tolerant approach.

Here are the logical propositions related to the state automata in Figure 6.3.

- A = Received Registration request from a new peer.
- B = New peer successfully registered.
- C = Checkpointing request received.
- D = A new improving solution has been discovered by a peer.
- E = Checkpointing operation accomplished.
- F = No interval remain in the network.
- G = Termination flag set.
- H = Received a request for a new neighborhood from a peer.
- I = New neighborhood sent to the peer.
- J = All peers in the network are in *Terminating* state.

6.2.2.2 Registration mechanism

When a new peer joins the network, it sends a request to the Server entity for registering itself. Basically, it informs this entity with its communication information: the machine where it is being run and the network information for communicating with it. When the Server entity receives this request, it determines an identifier for the new peer. This identifier will enable the Server entity to perform more easily the operations described hereafter. Similarly to the approach we described in a static environment, the Peer-to-Peer network, in order to begin the exploration of the search tree, must be provided with the initial value for the upper bound and the entire set of work to process. In the static approach, one peer is given initially the whole task to perform and share with the rest of the network and all the peers are informed about the upper bound initial value.

Now, in this *dynamic* approach, peers join the network at any time of the execution and do so in a given order. Thus, along with the identifier, the Server informs the newcomer about the best upper bound discovered so far by the network, for the problem being solved. If the newcomer is the first peer to join the network, then the Server entity will communicate an initial value which has been defined by the user. Proceeding this way allows one to whether begin the execution of the Branch-and-Bound Algorithm from scratch, by setting this initial value to infinity or from the best solution known so far for the problem being solved, for benchmarking purposes or solving a *"never-solved"* instance of the problem. In addition to this, the first peer to join the network is given the whole interval to explore. For the others, a list of neighbors containing information to communicate with them is provided. The mechanism defining the neighborhood is described in the following subsection.

To illustrate this mechanism, we give a practical example on Figure 6.4. In this case, we consider P and the Server entity S. Upon joining the network, P sends to the S a Registration request by providing its communication information. More technically, it is its network address and the port on which P is listening for incoming messages. When P receives this request, its sends back a message containing a new Identifier , the best solution found so far by the network, and, should the case arise, an interval to explore and a list of neighbors to connect to for possible work requests.

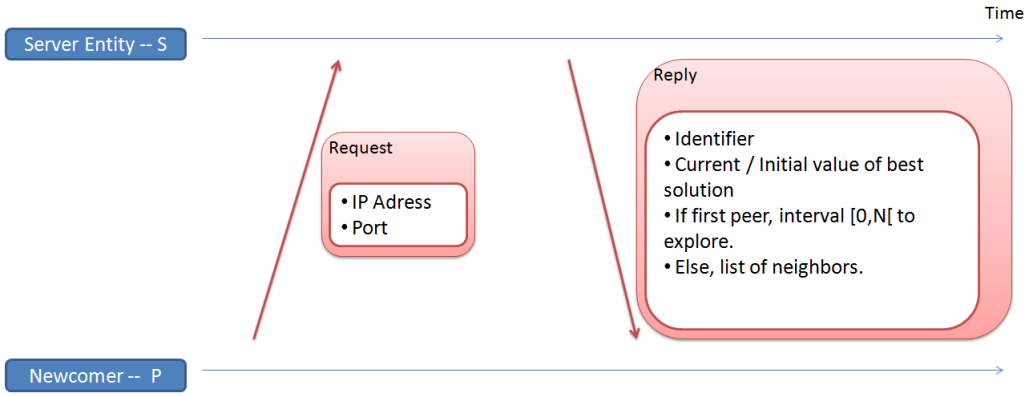

Figure 6.4: Sequence diagram representing the Registration Procedure.

6.2.2.3 Neighborhood mechanism

As described earlier, the maintenance of the topology is now dealt with by the Server entity. Reminding that a peer is granted neighbors upon joining the network, this mechanism is involved to provide a new set of neighbors to a peer in case all the existing ones fail to share an interval upon work requests. The number of neighbors a peer has is arbitrarily chosen by the user. One may wonder which criteria are chosen for selecting neighbors. In our approach, if a peer requests a number N of neighbors, the Server entity selects the N peers which have performed checkpointing operations the most recently. The idea is that if those N peers have performed checkpointing operations lately, then the probability of this information being obsolete is lower and thus the probability that these N peers still having work to share is higher. Practically, this mechanism has a relatively simple operation as, when a peer P asks the Server entity S for a new neighborhood, S sends back a set of peers information. (See Figure 6.5)

In addition to this, this process allows the Server entity to reassign some work units from failed peers. Indeed, the checkpointing mechanism makes the peers save their work unit towards the master on a periodical basis. Based on this information, the Server entity can easily determine which peers are facing faults. The work units saved by *crashed* peers have an age greater than the checkpointing period duration. In a more formal way, if t_{save} is the time when the current peer performed its last

checkpointing operation, t_{now} the current time and T_{check} the checkpointing period duration then the Server entity can determine whether a peer has *crashed* if it fulfills the following condition: $t_{save} < t_{now} - T_{check}$.

Given this mechanism, when a peer contacts the Server entity for a new neighborhood, the latter can deduce that the peer failed to obtain a work unit from its neighborhood. Thus, along with a new set of neighbors, the Server entity sends, if it exists, a work unit which satisfies the above condition. This allows the entire network to recover work units from failed peers.

This mechanism raises concerns about another aspect of the P2P network: the topology of the overlay. Indeed, when peers perform checkpointing operations and requests for a new set of neighbors, the network's topology can be dramatically over time. In our "*static*" approach, the topology was predefined. It was chosen according to specific characteristics about the graph diameter and the average degree of the nodes. The graph was connected and non-oriented, and all the B&B mechanisms were based on this property. Here, the resulting graph becomes oriented (if a peer P_1 is a neighbor of a peer P_2, the converse is not necessarily true) and non-connected.

Nevertheless, as peers can be all exploring an interval, each peer can be virtually connected to any peer in the network, thus eliminating concerns about having peers isolated from the network and remaining indefinitely idle. The second issue deals with global information. As the graph is no longer connected, one can no longer guarantee that if a peer discovers an improving solution and broadcasts it to its neighbors, all the peers eventually become aware of this solution. Thus, in addition to this P2P broadcast procedure, the new solution is communicated to the Server Entity through the checkpointing mechanism, as every peer is supposed to contact this entity at least on a regular basis. Similarly, the detection of the termination will have to be handled by the Server Entity.

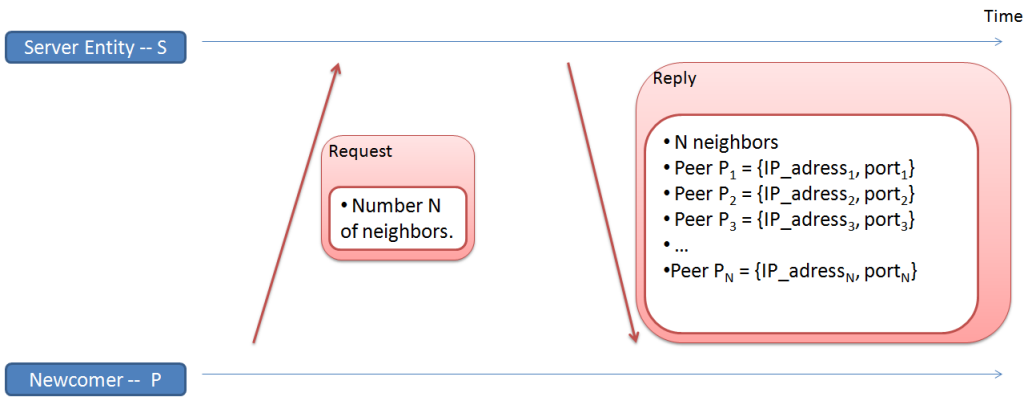

Figure 6.5: Sequence diagram representing the Neighborhood Request Procedure.

6.2.2.4 Load Balancing Mechanism

Load balancing consists in sharing the tasks to perform among as many peers as possible to take advantage of the involved computational resources. Similarly to the approach described in a static context, this process is handled in a fully distributed way. When a peer is assigned a set of peers as neighbors and whenever it finishes the exploration of an interval, it sends requests to its neighbors in an iterative way. If a neighbor holds an interval, it splits it into two equal parts and sends the second half to the requesting peer. Otherwise, the requesting peer proceeds to the next neighbor until all neighbors have been probed. (See Figure 6.6). Figure 3.3 recalls the work sharing mechanism in a static context.

One can notice that this mechanism is strongly dependent from the neighborhood granted by the Neighborhood Mechanism described hereabove. Indeed, the neighborhood mechanism selects the peers that have saved their interval the most recently. From a "load balancing" point of view, this strategy is based on the idea that those peers have the highest probability of holding an interval when the current peer begins its round of requests.

Of course, this strategy is advantageous when failures occur with a high probability. One can imagine other strategies for selecting neighbors. For instance, one could select peers that have the biggest intervals, assuming that bigger intervals have more chances to require more computation time.

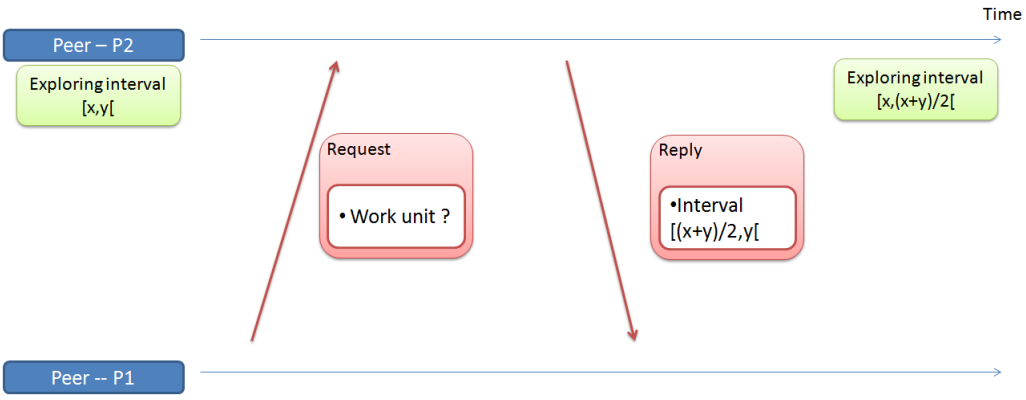

Figure 6.6: Sequence diagram representing the Load Balancing mechanism between two peers.

6.2.2.5 Checkpointing mechanism

To enable the network to recover from a fault of one of the peers, those latters have to save periodically their state. Here, it consists in the tasks/sub-tasks they still have to process. More precisely, they send a message containing the remaining interval to explore towards the Server entity. The Server entity memorizes this data and,

using similar information from other peers, it can update this interval by checking if it has been explored whether partially or totally by another peer. Indeed, such a situation can occur when the current peer undergoes a fault so that the Server entity reassigns its interval to another peer. (See Figure 6.7).

One of the main concerns dealing with fault tolerance is the work sharing procedure. Indeed, one must ensure that the newly assigned interval is not lost and that, in case of failure from the requesting peer, this interval can be reassigned later. More technically, suppose that two peers P_1 and P_2 are neighbors and P_2 asks P_1 for a work unit. Peer P_1 shall proceed into three steps. First, it splits its interval into two equal parts. Second, it informs the Server entity about the two newly created intervals. Third, peer P_1 sends the second half of the original interval to peer P_2. During this procedure, peers P_1 and P_2 may fail at any time. Three situations have to be considered:

1. *The failure occurs before step* 1 *or after step* 3*:* If P_1 fails, then its interval will be reassigned to another peer. If P_2 fails before step 1, nothing happens. If it fails after step 3, its interval will be reassigned as well.

2. *The failure occurs between steps* 1 *and* 2*:* If P_1 fails, the Server entity still thinks it holds its previous interval. Thus, it will be reassigned to another peer later, and peer P_2 will ask another peer for an interval. If P_2 fails, then peer P_1 will resume exploring the original interval.

3. *The failure occurs between steps* 2 *and* 3*:* If P_1 fails, the Server entity now thinks that the second half of its interval has been effectively sent to P_2. Thus, if P_2 succeeds in obtaining an interval from another peer, the Server entity will think that P_2 is exploring two intervals. Actually, P_2 is exploring only the last one. As the former one will not be updated by P_2, it will be reassigned to another peer as well as the interval P_1 was supposed to explore after splitting the original interval. If peer P_2 fails, then peer P_1 will explore its own interval and the interval assigned to P_2 will be reassigned to another peer.

6.2.2.6 Best Solution Sharing

In opposition to the static environment approach, sharing the best solution can not be handled exclusively in a fully decentralized fashion. Indeed, for a given peer, the neighborhood mechanism defines a set of neighbors unilaterally: if a peer P has a set of neighbors $\{N_1 \dots N_k\}$, the respective sets of neighbors of these peers might not necessarily include P. More mathematically, the graph representing the network topology becomes oriented. Thus, making use of the Server entity is the only way to ensure that all the peers in the network get informed about a new upper bound whenever it is discovered. In comparison with the approach in a static environment, the mechanism differs slightly. Instead of broadcasting the new upper bound to

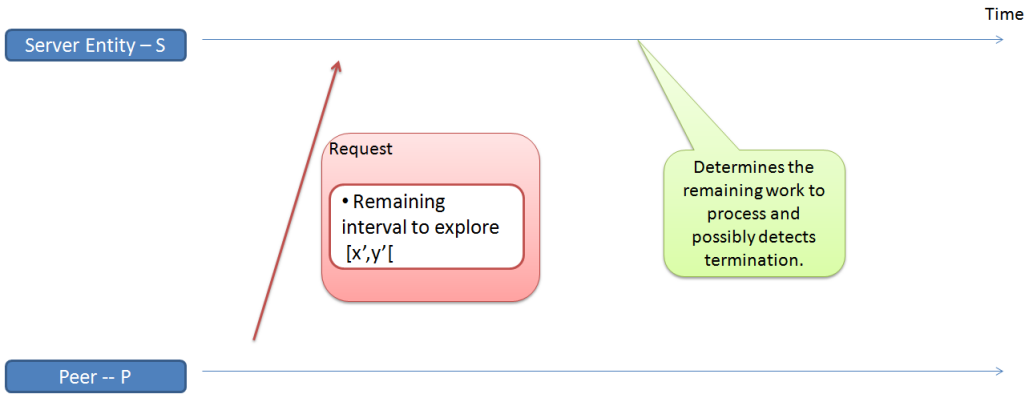

Figure 6.7: Sequence diagram representing the checkpointing mechanism.

neighbors upon discovering it or receiving it from a neighbor, a peer performs a checkpointing operation by sending the new upper bound along with the interval to explore to the Server entity. Other peers get informed about this new upper bound upon performing their next checkpointing operation.

From a practical point of view, we take advantage of the P2P mechanisms layer by sending the new upper bound along with work requests messages. The idea here is to allow a peer to be informed about a new upper bound faster if it performs a work request before checkpointing towards the Server entity and receive the new upper bound from it. (See Figure 6.8).

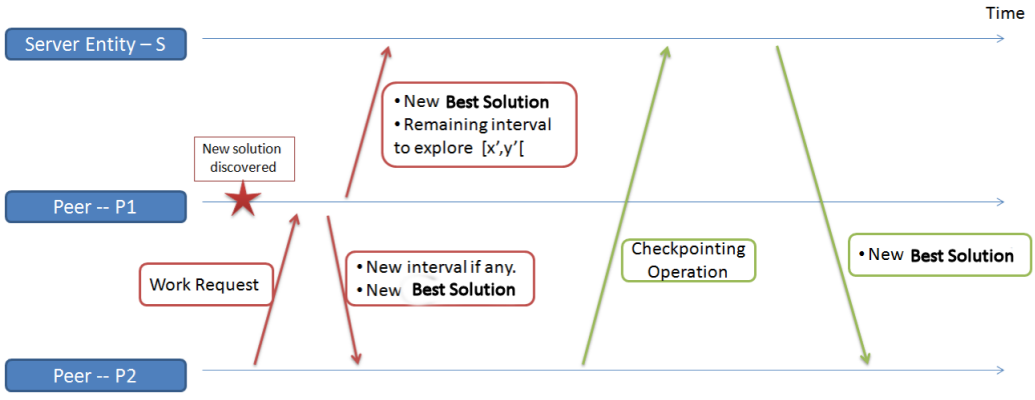

Figure 6.8: Sequence diagram representing the Best Solution Sharing mechanism.

6.2.2.7 Fault Tolerance

In this approach, the Fault Tolerance feature is handled in an implicit way. Indeed, the fault of a peer is not detected in an active way through a dedicated mechanism. The assumption is made that every peer is supposed to periodically save its data

towards the Server entity. When this Server entity finds out that some data has not been updated for a *greater-than-period* amount of time, it can deduce that the related peer must be undergoing a fault (more generally, an anomaly). Thus, its work unit can possibly be reassigned to another peer later, thus ensuring the exploration of all the intervals in the network.

In the following section, we present a thorough experimental study of the performance of our approach.

6.3 Experimental Evaluation

6.3.1 Experimental Setting

6.3.1.1 Protocol

The experiments have been conducted on the Grid'5000 Experimental Grid. As the behavior of the Branch-and-Bound algorithm is, by nature, unpredictable, the obtained results are an average of four separate runs. For each execution, the computational resources that are involved are located on the same geographical site.

The goal is to study the impact of pre-determined failure scenarios on the performance of our approach. The most accessible way to evaluate it is through gathering information about the Branch-and-Bound algorithm's execution. Indeed, for example, if some peers in the network fail then the time needed to solve a problem may increase and this could be seen by studying the evolution of the number of remaining solutions to be explored through time.

In order to evaluate efficiently the robustness of our approach, this latter is run using various models for simulating faults, those models being described hereafter. The distribution of faults through time can be used as a parameter to simulate real-case scenarios. Along with this parameter, we also vary the size of the network, the failure rate (i.e. the proportion of peers that can fail) and the number of neighbors for each peer. Figure 6.9 indicates the different settings used in our experiments. Note that the failure models are numbered from 0 to 4: these models are described later.

The problem being solved is the Flow-Shop Scheduling Problem. The instance chosen is Ta050 (permutations of 50 jobs on 10 machines). The initial value of the upper bound for the B&B algorithm has been set to the optimal solution's score : 3065. The instance is run until it is completely solved. Using these parameters allow to reduce the duration of the runs to dozens of minutes. In situations where the failure rate is high, the performance of the approach can be significantly degraded and the resolution time can become much higher. Thus, the duration of the runs has been limited to 30 minutes.

6.3.1.2 Fault Injection

Some works propose different protocols to inject faults into the network. ORCHESTRA [Dawson 1996] allows the user to act on the communication layer by manipulating messages. These messages can be delayed, reordered, lost, duplicated, altered and new messages can be spontaneously introduced in the system. NFTAPE [Stott 2000] allows to simulate high workloads and trigger failures. This platform allows the user to design custom injection mechanisms. LOKI [Chandra 2000] is a fault-injection system whose policy is based on a partial view of the whole network and faults are injected *a posteriori* (after analyzing global properties of the network at a given time).

In our experiments, faults are injected into the network as follows: every 30 seconds, a peer determines if it can fail according to the failure rate parameter. If so, a peer simulates a failure by remaining idle for a predetermined duration, depending on the failure model. It remains active to reply to potential work request that can be received from other peers. Once the duration has passed, the peer restarts its execution from scratch, as if it were a new peer joining the network.

Model	Network Size	Failure Rate	Number of Neighbors
0	100	25%	20% of Network Size
1	200	50%	40% of Network Size
2	300	75%	60% of Network Size
3		100%	80% of Network Size
4			

Figure 6.9: Range of the parameters for failure simulation scenarios.

6.3.1.3 Failure scenarios

For our experiments, we studied four different scenarios for generating and injecting faults in the network.

1. **Failure-free model:** This model is used as a reference to analyze the experimental results. None of the peers fail during the execution of the approach.

2. **Periodical renewal model:** In this model, a fixed amount of peers (corresponding to the failure rate parameter) leave the network and an identical number of peers join the network. This simulates an environment where peers regularly join/leave the network as, for instance, in P2P file-sharing systems where users connect to the network, exchange data and leave the network.

3. **Periodical renewal with fixed delay:** This model is identical to the previous one, except that, after leaving the network, peers wait for a fixed amount of time (here , 15 seconds) before joining back the network. This simulates

situations where a whole set of peers fail simultaneously. Typically, it can correspond to the crash of a machine or the network infrastructure.

4. **Periodical renewal with random delay:** Same model as the previous one, except that the delay for a peer before rejoining the network is set to a random value comprised between 0 and 30 seconds. This model simulates faults which affect only individual peers. One can imagine situations where a peer running on a given machine fails because of a software error or because of a temporary network overload which suddenly increases network latency, for instance.

5. **Periodical renewal with delay based on a normal distribution law:** This model is a combination of models 2 and 3. The durations of failure are determined according to a normal law whose parameters are given by an average value of 15 seconds and a standard deviation of 5 seconds.

Figure 6.10 gives an overview of the parameters used to simulate peers leaving or joining the P2P network.

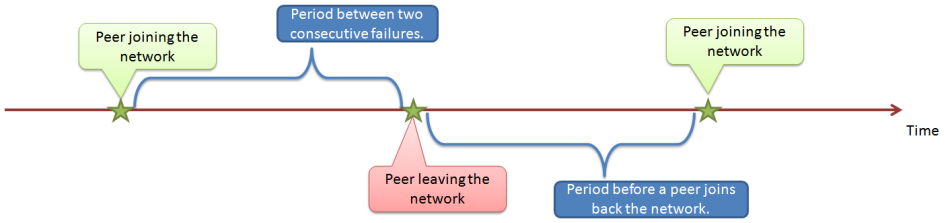

Figure 6.10: Parameters used to simulate peers leaving or joining the P2P network.

6.3.2 Experimental results

In this section, we intend to analyze the impact of each parameter upon the performances and more precisely, the robustness of our approach. Note that, given the parameters' set described in Figure 6.9, the number of runs reaches 204. For the sake of clarity, we perform our analysis by only selecting runs which allow one to see the influence of each parameter as clearly as possible.

6.3.2.1 Reference executions

This experiment consists in running our approach without simulating any failure in the network. This will allow to have a clearer view of the impact of failures on the performances of our approach. The size of the network ranges from 100 to 300 peers. Figure 6.11 represents the evolution of the number of remaining solutions to explore over time. The number of neighbors for a peer has very little impact so only results for a neighborhood size equal to 20% of the network's size are represented. The $Ta050$ instance is solved within between 125 and 225 seconds.

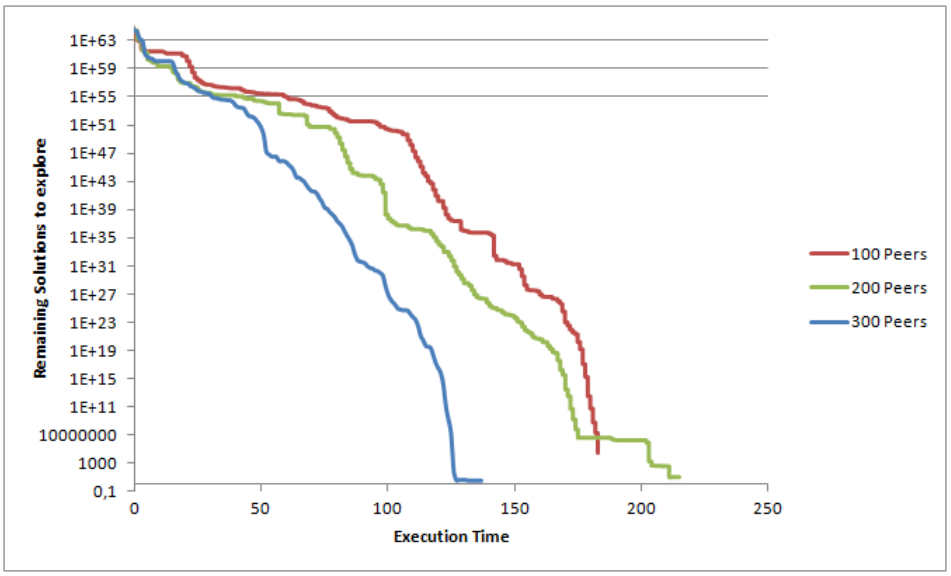

Figure 6.11: Evolution of the number of remaining solutions to explore without any fault.

6.3.2.2 Failure rate

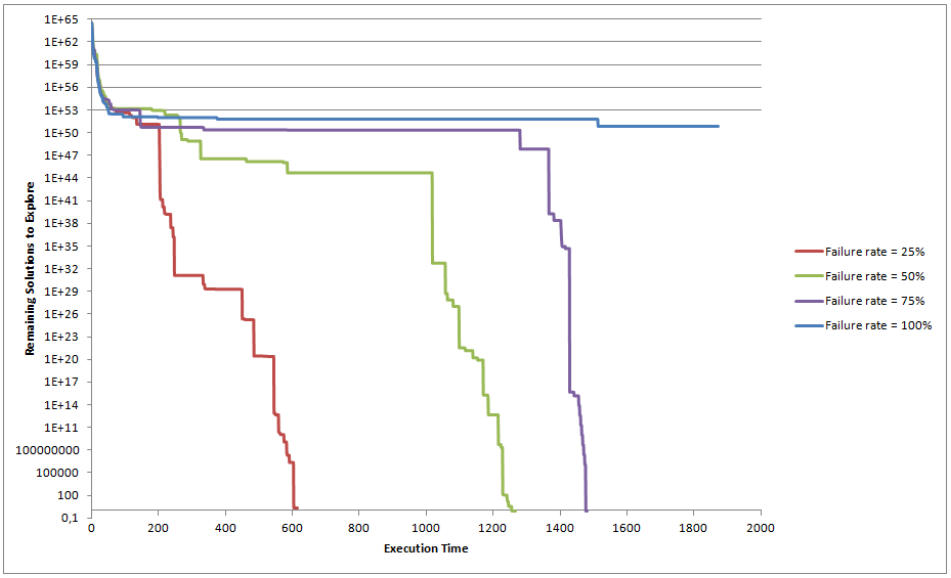

Figure 6.12: Evolution of the number of remaining solutions to explore with multiple failure rates.

Figure 6.12 represents the results obtained by selecting model 1, setting the size of the network to 300 and the percentage of neighbors to 20%.

One can see that increasing the number of failures degrades significantly the performance of the approach. Indeed, chances that a peer can obtain an interval from a neighbor upon request diminish as the latter may face a fault. Results indicate that for failure rates higher than 75%, the resolution process is almost stalled.

6.3.2.3 Number of Neighbors

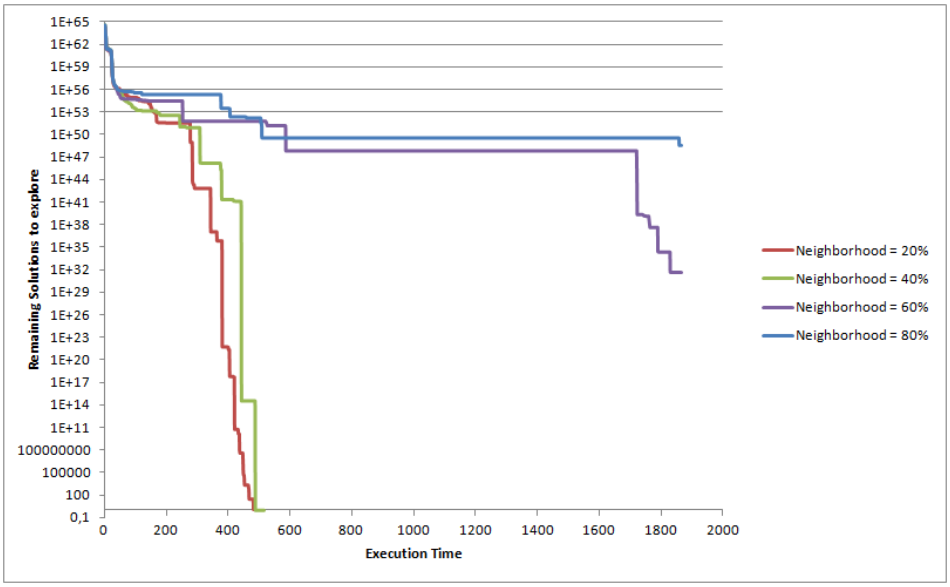

Figure 6.13: Evolution of the number of remaining solutions to explore with multiple neighborhood sizes.

Figure 6.13 represents the results obtained by selecting model 2, setting the failure rate to 25% and the size of the network to 100 peers.

This result can be somehow surprising at first sight. Indeed, one could have expected that when the number of neighbors increases, the resolution process performs faster as the load is balanced more quickly. However, results seem to show the contrary. This emphasizes the following phenomenon: if a peer has a high number of neighbors, then the work request procedure takes more time to complete as neighbors are interrogated one at a time. Thus, if the number of failed peers is high, it takes much more time in average for a given peer to obtain an interval whether from another peer or from the Server entity, which increases the resolution time in the experiment.

One intuitive idea comes to mind when analyzing the impact of the neighborhood size on the performances of the P2P network. One may wonder if a deployment based on a fully Master-Slave architecture, which can be achieved by

setting the number of neighbors to zero, performs better.

Figure 6.14 compares, with the same parameters, the results obtained with a number of neighbors equal to 20% of the size of the network and those with no neighbors at all. One can see that the "Master-Slave" approach performs worse. The main reason of this phenomenon is that all the load balancing operations are now handled by the server entity, thus increasing significantly the load of communications upon it. When the peers perform checkpointing operations as well as work requests, additional delays are introduced because the central entity requires extra time to process all the incoming requests, thus degrading the performances of the network.

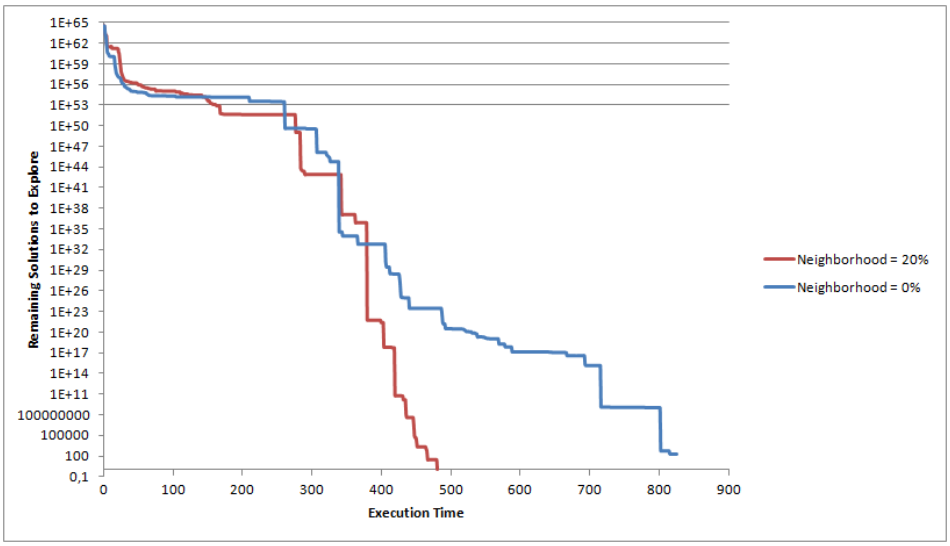

Figure 6.14: Evolution of the number of remaining solutions: Comparison with a "Master-Slave" approach, with peers having no neighbors.

These results have been obtained by using intensive failure scenarios, where failures occur at a very high frequency. Our approach has been pushed to its limits and its robustness has been confirmed experimentally. However, the studied scenarios have one limitation: they distribute the failure only among peers and uniformly through time. In real environments, systems can fail with a probability which generally increases with time: the longer a system has been active, the higher chances are for it to fail.

In the following section, we study another model for which the main parameter is the lifetime of the peers.

6.3.3 Lifetime-based failure scenario

6.3.3.1 Failure model based on the Weibull exponential distribution law.

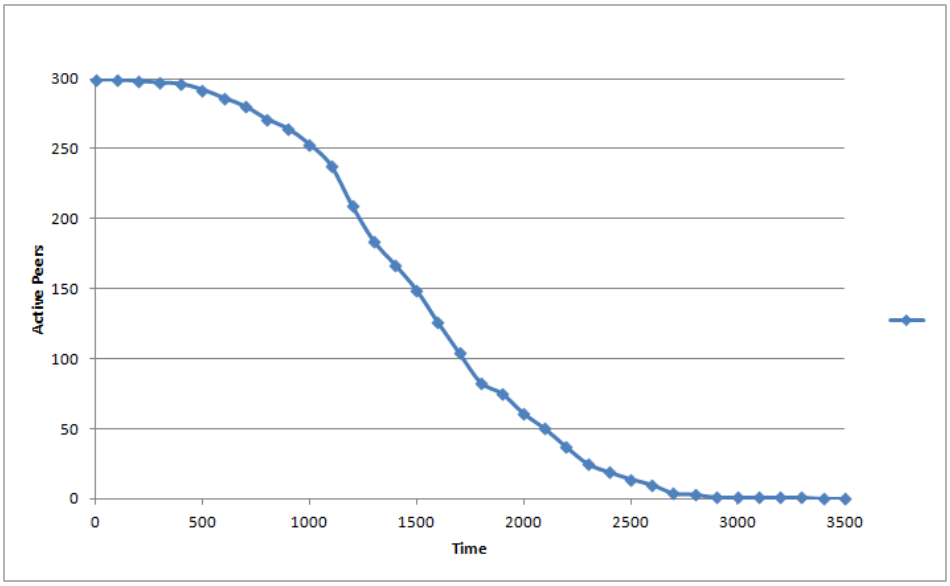

Figure 6.15: Evolution of the number of active peers through time.

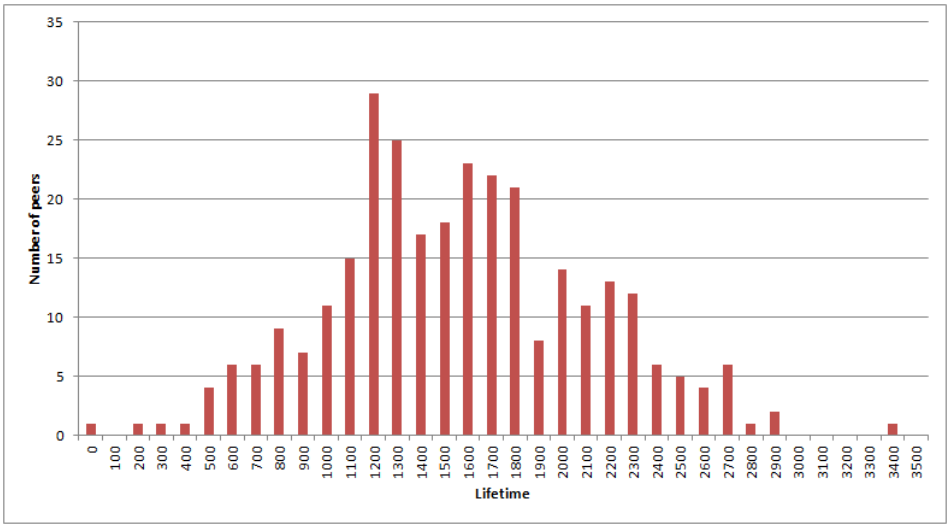

Figure 6.16: Distribution of the peers' lifetimes.

The failure models we used in the previous experiments as well as the most commonly use fault injectors described earlier do not take into account the lifetime of the computational entities. To give more realism to the experiments in a volatile

environment, we consider a model in which the lifetime distribution of the processes follows an exponential law.

In the literature, most of the lifetime-based failure models use such distributions to describe the lifetime of computing entities. In [Stutzbach 2006], authors analyze some properties of the most commonly used P2P applications, mainly dedicated to data sharing like Bittorrent [Bittorrent 2005], Gnutella [Milojicic 2008], Kademilia [Maymounkov 2002] and others. Most of the work focuses on churn (the dynamics of peer participation in a P2P network) and authors study, from data gathered from actual users, the rate of peer departure or arrival by using distribution laws to describe it. Those laws include the normal distribution, the exponential distribution, the Weibull distribution and other heavy-tailed distributions. In [Bendjoudi 2012], Bendjoudi *et al.* base their fault injection mechanism on an exponential law for the lifetime of the peers.

More generally, the reliability of a process i is considered as the probability that it operates 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 during the time interval $[0, t]$, $R_i(t) = P(T > t)$, where T is a random variable representing the failure time. Therefore, the failure probability is given by: $F(t) = 1 - R(t) = P(T = t)$.

The last model we make use in our experiments determines the lifetime of the peers according to the Weibull exponential distribution law. In our experiment, we choose to set the number of peers to 300 and, to generate the corresponding list of lifetimes (in seconds), we used the R statistical tool. The parameters used for generating the distribution were 3 for the shape and 1 for the scale and the values have been multiplied by a factor of 1800 afterwards. The probability density function for the Weibull distribution (with two parameters) is: $f(x) = \frac{k}{\lambda}(\frac{x}{\lambda})^{(k-1)} \cdot \mathrm{e}^{-(\frac{x}{\lambda})^k}$, where $k > 0$ is the *shape* parameter and λ the *scale* parameter. With parameters set to $k = 3$ and $\lambda = 1$, we have $f(x) = 3x^2 \cdot \mathrm{e}^{-x^3}$. The cumulative distribution function becomes $F(x; k, \lambda) = 1 - \mathrm{e}^{-\left(\frac{x}{\lambda}\right)^k} = 1 - \mathrm{e}^{-x^3}$.

Figure 6.16 represents the distribution of lifetime values for the peers which are represented by intervals. For instance, the column labeled "1100" represents the number of peers whose lifetime is comprised between 1100 and 1200 seconds (here, 15). Figure 6.15 extrapolates this information by indicating the number of active peers through time.

6.3.3.2 Protocol

The experimental protocol designed for our previous experiments was designed to study the impact of failures on the robustness of our approach during the resolution of a small instance. An instance was chosen as well as an initial value so that the resolution could be achieved in a matter of minutes. This allows to perform a great

number of runs within a short amount of time. Here, we changed these parameters so the execution time without failures would be approximately 20 minutes or 1200 seconds. Thus, the size of the network is set to 300 peers, the instance being solved is $Ta050$ and the initial value for the upper bound is set to 3165, the optimal solution for this instance having a score equal to 3065.

6.3.3.3 Result

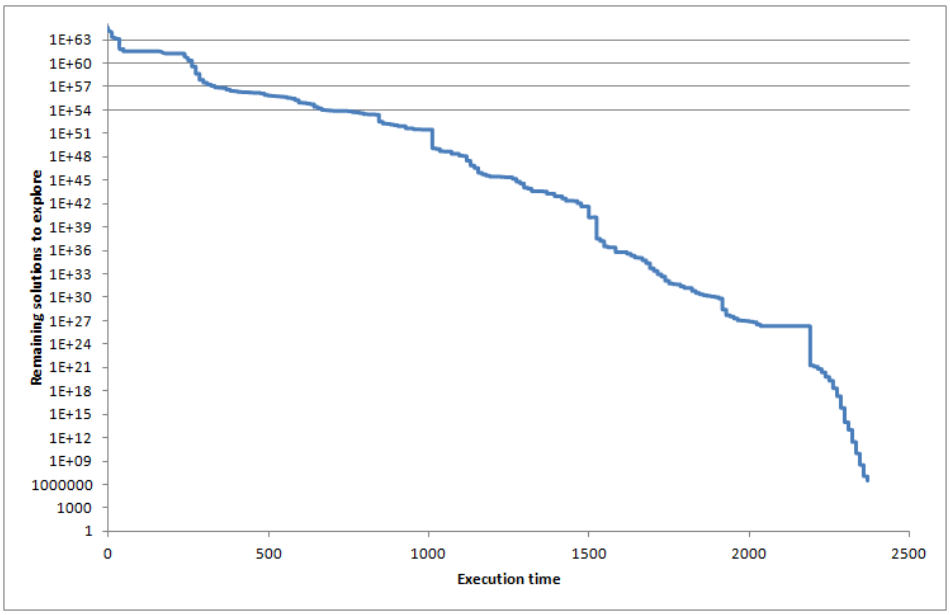

Figure 6.17: Evolution of the number of remaining solutions to explore through time.

Figure 6.17 represents the evolution of the number of remaining solutions to explore along with execution time. The 300 peers of the P2P network take approximately 2400 seconds to solve completely the $Ta050$ instance with the given parameters. This shows that, in real-case scenarios, our approach can face failures and recover from it in an efficient way. Note that the values for the remaining solutions are shown using a logarithmic scale. Thus, the last value being 0 (when the instance is solved) is not displayed, thereby creating a "gap".

6.4 Conclusion

In this chapter, we described a Fault-Tolerant approach based on the one designed for static environments. This approach is based on the use of a central entity that manages checkpointing information, network topology and termination detection. The approach can handle the departure or the joining of a peer through a Registration mechanism which keeps trace of the existing peers in the network. The

work sharing mechanism remains similar to the static approach. The best solution sharing mechanism takes advantage of the central entity to broadcast more efficiently the best solution found and the network's connectivity is ensured through a neighborhood mechanism which can redefine the neighbors of a peer in case of node failure.

In addition, we conducted experiments to study and prove the robustness of our Fault-Tolerant within a dynamic environment. Our approach has been tested with two types of failure scenarios. We have chosen some failure-intensive situations where failures occur periodically at a high pace and represent various situations that can be encountered in the real world. Second, we have chosen a model based on the lifetime of the peers, which is based on an exponential-like distribution law. The obtained results show that our approach can efficiently recover from faults in most of the situations and when the rate of failures is very high, our approach runs much more slowly.

Chapter 7

Conclusion

In this thesis, we presented our contributions to grid-based and Peer-to-Peer Branch-and-Bound algorithms for the exact resolution of Combinatorial Optimization Problems. We particularly addressed some challenging issues related to scalability, fault tolerance, resources heterogeneity and load balancing, and the design and implementation of grid-aware applications.

We first began by defining some key concepts related to Combinatorial Optimization (different optimization methods, exploration strategies, ...), Parallel Programming (what elements can be parallelized and how to synchronize multiple processes), Grid Computing (existing parallelizations of the Branch-and-Bound algorithm, issues faced when evolving at large scales, ...) and Peer-to-Peer Computing. This study emphasized the importance of the degree of parallelism used for a parallel algorithm, the main issues to be dealt with when evolving in grid-like environments. It also gave an overview of the existing works, mostly based on Master-Slave architectures and the main limitation they face when dealing with scalability.

First, we designed a fully Peer-to-Peer approach to overcome the limitation of scalability caused by network communications. The architecture of our application is based on the concept of a Peer. Each peer executes the Branch-and-Bound on a subset of solutions and computes a partial solution. To do so, several mechanisms were designed to perform the main Branch-and-Bound operations. Through the Work Sharing mechanism, a peer manages its local work pool and shares its work units with its neighbors. This process is done so it guarantees no redundancy during the exploration process. Through the Best Solution Sharing mechanism, a peer can compute the best solution for its work pool and broadcast it over the network through its neighbors which allows those latter to possibly branch their exploration trees and speed up the exploration process. Through the Termination Detection mechanism, a peer can determine the presence or absence of a work unit somewhere in the network through iterative work requests sent to its neighbors. This approach is implemented without using any third-party library or platform which makes it deployable on a high variety of systems. To the best of our knowledge, no parallel B&B algorithm was designed under the fully Peer-to-Peer paradigm at the application level before. Along with this contribution, we provided a formal proof

of the correctness of our approach, that is ensuring that the algorithm terminates correctly, the exploration process is performed within a finite amount of time and that when termination is detected, all peers are aware of the optimal solution for the problem being solved.

Second, we make an extensive experimental study of our approach. Its correctness has been proven experimentally by solving to optimality multiple instances of the Flow-Shop Scheduling Problem from scratch. We then studied its performances at large scales, by deploying our algorithm on networks of up to 150.000 peers. By comparing it to a traditional Master-Slave based approach, results showed an improved scalability, that is, with 150.000 peers deployed, those peers still spend more than 90% of their time to exploring solutions whereas this ratio drops to less than 10% in the Master-Slave approach. Through studying the amount of communications over the network in both approaches, we showed that a drop of scalability is not due to network congestion. In a second step, we analyzed the impact of the network topology on the performances of our approach by testing it with some of the most commonly used topologies in the literature. Results indicate that our P2P approach can operate independently from the network topology and that it operates at best with topologies having small-world properties.

Third, we extended our approach to adapt it to dynamic environments. We introduced a central entity which is in charge of gathering checkpointing information from peers and handle the joining and departure of the peers through a Registration system. This central entity can reassign neighbors to the peers in case of failure and thus can easily detect termination. To evaluate the robustness of our approach, we chose multiple failure scenarios. Some are *intensive*, that is they simulate repetitive failures occurring at a high rate. Results show that with low rates of failures, our approach can successfully solve instances, but under high rates of failure (namely, more than 75% of the peers failing), the exploration process is heavily impacted. When the number of neighbors decreases, the load balancing mechanisms performs faster. However, results also show that our Fault-Tolerant approach still performs better than a fully Master-Slave approach as it again distributes efficiently the communication load among the peers. Finally, we experimented our Fault-Tolerant approach with a lifetime-based scenario, where the lifetime of the peers follow an exponential-like law. Results showed that under real-case scenarios, our approach remains robust as it successfully solves an instance of the Flow-Shop Scheduling Problem.

In the future, we plan to investigate the following research tracks to extend our work. The first issue we plan to address is the mapping of the toric hypercube overlay on the Grid'5000 physical network. In our experiments, we have considered the cluster-based proximity of peers provided by the OAR reservation tool of

Grid'5000 to deal with the mapping. Such mapping is quite similar to the IP-based proximity metric used in [Nguyen 2012]. In the future, other distance metrics [Huffaker 2002] will be investigated to improve the efficiency of the mapping.

Second, fault tolerance is critical in order to solve large scale difficult instances. Two main issues left open in our fault-tolerant approach will be addressed in the Ph.D. thesis of Trong Tuan Vu: (i) using robust overlays while distributing work among peers, and (ii) using different checkpointing strategies where different distributed servers are in charge of collaboratively and distributively dealing with churn and peer faults (e.g., [Nguyen 2012, Bendjoudi 2013]). We expect that designing advanced robust overlays, e.g., by getting inspired from existing data-oriented p2p frameworks, will allow us to increase the logical connectivity among peers and thus to decrease the cost and the need to maintain the overlay by using any centralized entities. We also expect that distributed checkpointing strategies shall allow us to better balance the cost of checkpoint operations and thus to improve performance. With respect to the Grid'5000 computational grid, this would allow us to deploy a parallel B&B fault-tolerant framework under the best-effort mode, in which one has no strong guarantee about the availability of reserved resources. Such a framework would allow us to tackle large scale unsolved instances and to effectively solve them, as was done in [Mezmaz 2007a] for one of the 50×20 highly time-intensive problem. instances.

Third, for validation we have considered the Flow-Shop single permutation-based problem for which very short information (intervals, best solution, ...) is communicated. Other problems requiring larger communication messages should be investigated. For instance, for the 3D Quadratic Assignment Problem (Q3AP) studied in [Mehdi 2011], more data (flow and distance matrices) are communicated. The challenge will be to revisit the proposed P2P approach to deal with work sharing involving larger-sized messages. In addition, other problems than permutation-based ones should be investigated.

Fourth, according to many HPC experts (IDC - Analyze the Future, Nov. 2011), heterogeneous (GPU-based) computing is the new paradigm for the Exascale Era. The reason is, given that exascale computing is announced for 2018-2020, achieving reasonable exascale performance in such compressed time frame presents an array of daunting challenges that cannot be met only through evolutionary explorations from existing technologies and approaches. With the arrival of GPU resources in clusters and computational grids, our objective is to study the design of large scale heterogeneous peer-to-peer GPU-based Branch and Bound based on the conjunction of GPU [Lalami 2012], multicore and distributed computing. The objective is to combine the P2P approach proposed in this work with the GPU-based B&B approach proposed in the Ph.D thesis of Imen Chakroun [Melab 2012].

Appendix A

Peer-to-Peer Protocols

Contents

This section deals with the main constraints related to the Peer-to-Peer technology, which can have a non-negiligible impact on the deployment and the performances of Peer-to-Peer applications and systems. In [Milojicic 2008], authors provide a detailed analysis of the main characteristics of any Peer-to-Peer networks.

A.1 Main characteristics of a P2P network.

A.1.1 Decentralization

Peer-to-Peer models requestion the concept of storing and processing data using centralized servers and accessing this data through "*Request-Reply*"-based protocols. In traditional *Client-Server* models, data is stored on a few main servers and is communicated, through network infrastructures, to client machines acting as user

interfaces. Such centralized systems are best-fit for particular applications or tasks. For instance, security-based accesses can be better handled using such systems. However, this kind of architecture (Client-Server) can degrade performances because of communication bottlenecks or non-optimal resource usage. Although hardware performances and costs have improved, centralizing the sources of information can remain expensive in terms of configuration and maintenance. Human intervention maigh still be necessary to guarantee the coherence and non-obsoleteness of the content.

One of the key ideas behind decentralization is the influence that every user of the P2P network can have on data and resources. In a completely distributed system, every peer contributes equally to the network. Thus, implementing P2P can be very difficult to implement, technically speaking as there is no central entity having a global view of the network as well as the information stored in it. This is why many P2P systems are based on hybrid approaches like Napster [?], where a centralized resource directory is used and peers directly exchange data.

In fully decentralized systems, like Freenet [Clarke 2001] and Gnutella [Milojicic 2008], joining the network can be challenging. In Gnutella, for instance, new peers must be aware of a given number of pre-existing peers belonging to the network. They can also use a list of IP addresses of existing peers. The newcomer joins the network by connecting to at least one of them. From that point, it can proceed to discover other peers and gather their network information.

From a historical point of view, the first great project of Distributed Computing was the *Great Internet Mersenne Prime Search* (GIMPS, www.mersenne.org), whose objective was to harness computational resources distributed over the Internet network in order to calculate Mersenne's prime numbers. The GIMPS project initially used mails for communication purposes as this protocol is intrinsically decentralized. When a central entity assigns work units to unavailable computing resources, mail servers put corresponding mails into a waiting queue and send it again as soon as the resources becomes available again.

Then, another Distributed Computing project appeared (*"distributed.net"*), which was dedicated to decryption/decyphering. Initially, only one central server was used, which caused issues related to availability (Sometimes, for multiple weeks in-a-row). Therefore, a two-level proxy-system was used, on which a version of the server entity could be deployed and act as a proxy. Practically, task scheduling and assignment was easy as work coding was very simple : each work unit was referred to by a key and a set of work units could be represented by an interval of two integers.

P2P systems can be classified into two categories depending of their level of autonomy : "Pure P2P" and "Hybrid P2P". Of course, the classification can be made more precisely as shown in Figure A.1. This autonomy can have a direct impact of the self-organization ability and the scalability of a system, as the most decentralized systems are loosely linked to the network's or machines' infrastructure.

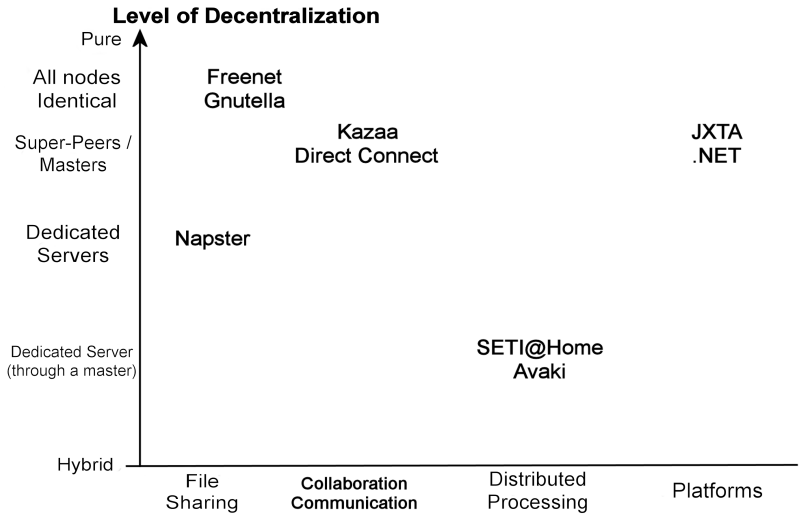

Figure A.1: Degrees of decentralization into various P2P networks.

A.1.2 Scalability

One immediate advantage of decentralization is scalability. It can be limited by various parameters like the number of "*centralized*" operations (i.e. synchronization or coordination operations) which have to be performed, the application's intrinsic parallelism as well as the programming model used to design the calculation process.

Napster faced the scalability issue by making peers download music files directly from the peers having these files. Thus, Napster gathered more than six million users at its highest. On the opposite side, SETI@Home [2001] [?] focuses on a unique task that is inherently parallel. The objective is to harness computational resources over Internet to analyze data collected from telescopes in the hope of discovering extraterrestrial life forms. This platform accounts for almost 3.5 million users. Other systems like Avaki try to face the issue of scalability through a distributed object programming model.

Scalability also depends on the ratio between communications and computation between P2P nodes. Some computing-intensive application like decrypting or finding prime numbers, spend most of the time to computation, thus having a ratio close to zero. This makes these systems very extensible. In other situations, as in SETI@HOME, a communication bottleneck appears around the central entity during intensive data exchanges. For instance, one megabyte of data requires 10 hours of computation to be processed, necessitating a network bandwidth equal to 1 MB multiplied by the number of computational entities involved in its processing. There exists other applications, like, graphical rendering, that require thousands of machines to create an animation within hours instead of a year. However, this application induces a lot of data transfers, which makes it much less extensible.

A.1.3 Ad-hoc property

First data-sharing P2P systems like Gnutella [2001] et Freenet [Clarke 2001] are ad-hoc by definition.

A peer *blindly* submits requests to other peers which will in turn search for the requested resource. This can make the search time become unbounded. In addition, the lookup process can fail even if the resource exists, thus making the system be non-deterministic.

More recent systems like CAN, Chord ([Stoica 2001],[Castro 2002]), Oceanstore [Kubiatowicz 2000] and PAST [Druschel 2001], impose a rigorous matching between the resource's identifier and the node providing that resource. Thus, a resource can always be retrieved as long as the hosting machines can be reached. Nodes, in those systems, become a network layer. Each node contains information about a small number of other nodes of the system. This limits the number of states to update if the list of resources changes through time and improves extensibility/scalability. The logical topology of this new network layer guarantees some properties about the cost of request processing. Oceanstore had been designed to extend to billions of users, millions of servers and more than 10^{14} files [Kubiatowicz 2000].

A.1.4 Anonymity

One of the usage made of the Peer-to-Peer technology is to allow individuals to use communication systems, without any sort of legal (or other type) constraints, on one hand, and to guarantee that no censorship of digital content is possible. The designers of Free Haven [Dingledine 2001] identified the following elements related to anonymity :

- Author : The author or the creator of a document can not be identified.
- Publisher : The individual that published the document on the system can not be identified.
- Reader : People that read or use content can not be identified.
- Server : Servers hosting a document can not be identified through this document.
- Document : Servers do not keep any knowledge about the content they host.
- Request : A server can not say which document is used to satisfy somebody's request.

In addition to these points , there exist three different types of anonymity between each communicating peer :

- Sender anonymity, masking sender's identity
- Receiver anonymity, masking receiver's identity

- Mutual anonymity, where both sender and receiver are hidden to each other as well as other peers[Pfitzmann 1987].

It is also crucial to determine the degree of anonymity that can be reached by a given method. Reiter and Rubin [1998] propose an overview of the different degrees of anonymity that ranges among "*absolute intimacy*", "*possible exposure*" and "*definitely exposed*" Absolute intimacy means that , even if an attacker can determine that a message has been sent, the message's sender can not be determine with absolute probability.

There exist six methods among the most commonly used, adapted to different degrees of anonymity under various constraints.

A.1.4.1 Multicasting

. Multicasting (or broadcasting) can be used to ensure receiver's anonymity [Pfitzmann and Waidner 1987].

A multicast group is created for users willing to remain anonymous. A user wanting to get a document registers to the corresponding group. The user inside the group which holds the document, sends it to the whole group. The receiver's identity is then effectively hidden for the other peers and sender's anonymity is guaranteed. This technique takes advantage from the underlying network that has the multicast feature, like Ethernet or Token Ring.

A.1.4.2 Mask sender's identity

In non-connected protocols like UDP, sender's anonymity can be guaranteed by spoofing the sender's IP adress. However, this requires to change protocol. Besides, it can not always be achieved as most ISPs keep trace of network packets originating from invalid IP adresses.

A.1.4.3 Spoof identity.

Instead of changing the sender's identity, anonymity can be achieved by modifying the identity of an interlocutor. For instance, in Freenet [Clarke 2001], a peer sending data to another peer can claim itself as the data's owner. Thus, an attacker can not be sure that the sender of a data is the actual sender.

A.1.4.4 Stealth paths.

Instead of communicating directly, two entities can do so through intermediate nodes. Most of existing techniques only ensure sender's anonymity. A peer willing to hide its identity elaborates a stealth path towards its interlocutor. One can cite Mix , Onion [Syverson 1997], Anonymizing Proxy [Gabber 1999], Crowds [Reiter 1998] and Herdes [Shields 2000]. Stealth paths can use persistent connections or buffered connections. By varying the path's length and by changing the path with a variable frequency, multiple degrees of anonymity can be achieved.

A.1.4.5 Untraceable aliases.

LPWA [Gabber 1999] is a proxy server that generates untraceable aliases for client machines. The client can create an account and be identified through it, thus masking its true identity from the server. This kind of method ensure sender's anonymity and make use of *reliable* proxies. The degree of anonymity is almost absolute.

A.1.4.6 Unwanted hosting.

Another interesting approach is to provide anonymity by hosting non-voluntarily a document on a node, using for instance hash functions. As the hosting is non-determinist, host can not be held responsible for hosting a document.

Now, we sum up the different forms of anonymity used into the most popular P2P networks as well as the techniques employed.

Gnutella [2001] and Freenet [Clarke 2001] provide anonymity through the way peers exchange documents. In Gnutella, a request is broadcast again and again until it reaches a peer hosting the requested document. In Freenet, a request is sent to peers that have the highest probability of hosting the requested document. The reply is sent along the same path.

APFS [Scarlata 2001] deal with mutual anonymity by considering that reliable centralization is impossible. Peers must inform a coordinator that they can act as directories. Both the sender and the receiver of a message must build up stealth paths.

Free Haven [Dingledine 2001] and Publius [Waldman 2000] are designed to guarantee protection against censorship. Document's anonymity is reinforced by splitting it afterwards and saved on multiple servers. Thus, a given server never hosts all the necessary data for external attackers. Mutual anonymity between the publisher and the reader is ensured through stealth paths. Both platforms build these paths by sending many anonymous emails. Publius could be improved by integrating reader's anonymity. Anonymous emails could be used also for publishing.

PAST [Rowstron 2001b], CAN [Ratnasamy 2001] and Chord [Stoica 2001] constitute a class of P2P systems based on a reliable infrastructure. One common characteristic is that object hosting is non-deterministic and a node can not be held responsible for hosting that object. The included routing mechanisms can be easily used to build stealth paths and ensure mutual anonymity.

A.1.5 Auto-organisation

In cybernetics, self-organization is defined as "a process where a system's organization increases spontaneously, that is it increases without being controlled by its environment or any external system". [Heylighen 1997].

In P2P systems, self-organization is necessary for extensibility (scalability), fault tolerance, resources volatility and infrastructure cost. The scale of P2P systems can not be predicted according to the number of systems involved, the number of users or the workload. Predicting one of these parameters can be very challenging without centralizing partially the system. Scalability intrinsically induces a higher probability of failure, which requires self-maintenance abilities from the system.

A similar reasoning can be applied to resources volatility. It can be challenging for a system configuration to remain as it is for a long period of time. Adaptation is necessary to face the departure or the joining of peers into the Peer-to-Peer network. As it would be to maintain a dedicated equipment and/or hire staff to manage such a dynamic environment, the issue is dealt with directly by the peers.

Some systems and products deal with self-organization. In Oceanstore [Bindel 2002, Kubiatowicz 2000], self-organization is applied to data localization and routing infrastructure [Kubiatowicz 2000],[Rhea 2001], [Zhao 2001]. Because of the sporadic availability of peers as well as variations of network latency and bandwidth, the infrastructure continuously adapts its routing and localization mechanisms.

In Pastry, [Rowstron 2001a], self-organization is managed through protocols dedicated to a node's departure or joining and based on a Fault-Tolerant network layer. Clients' requests are routed within $\lceil \log_{16} N \rceil$ hops.

FastTrack assigns faster searches and transfers to distributed self-organized networks. In these networks, the most powerful machines automatically become Superpeers and act as content directories, under criteria related to processing capacity and network communication (for instance, latency time and bandwidth).

SearchLing uses self-organization to adapt the network according the nature of search requests, allowing to reduce network traffic and the number of unsatisfied requests.

A.1.6 Ad-hoc connectivity

The ad-hoc nature of connectivity has a major impact on all classes of P2P systems. In the field of distributed computing, parallel applications can not be run on every system indefinitely. This availability may vary from one system to another. P2P systems and distributed applications must take into account this ad-hoc nature and be able to manage the departure or joining of computational resources in the network. Although it can be considered as an exceptional event in distributed systems, it is seen as usual or frequent in P2P systems.

In P2P applications dedicated to content sharing, users expect to have access to content whenever they want, regarding the connectivity of content providers.

In the most reliable systems, with guarantees on quality of service, the ad-hoc characteristic is diminished through redundant service providers, but some of them may remain unavailable.

In collaborative P2P systems and applications, the ad-hoc nature of connectivity becomes more obvious. Users cooperating with each other are more and more interested in mobile devices, thus making them more connected to Internet and better suited for collaborative work. To face this situation, cooperative systems allow transparent communication delays from disconnected systems. this can be achieved through the use of proxies dedicated to message reception , or other types of relays which can temporarily suspend communications for a disconnected system.

Besides, not all the systems will be connected to Internet. Even if it were the case, ad-hoc groups of users must be able to create ad-hoc networks to cooperate. Existing infrastructures like 802.11b/g, Bluetooth and Infrared. Thus, P2P systems and applications must be designed to support frequent exits or entries of peers.

A.1.7 Performances and security.

When designing a P2P system, those two points have a great importance regarding extensibility/scalability of an application.

A.1.7.1 Intelligent routing and network topology.

To take advantage of the great potential of P2P networks, it is crucial to understand and explore the possible interactions between peers. One of the most advanced works about social interactions is the "small-world phenomenon" on the Milgram experiment [1967]. The goal of that experiment was to find a series of acquaintances (that is, two people who know each other) that could link any couple of people in the US who do not know each other. By using postcards, Milgram discovered that, in the 1960's, Americans were linked through a "path" of six people in average. Adamic *et al.* explored power-law distributions for P2P networks, and introduced local search techniques using nodes having a high degree and a scalability cost sub-linear with the network size[Adamic 1999].

Ramanathan *et al.* [Ramanathan 2002] determine "good" peers, based on interests and manipulate dynamically network connections between peers to guarantee that high degree nodes with similar interests are strongly linked to one another. Establishing a good set of peers reduces the number of broadcast messages on the network and the number of peers processing a request before the result is found. Some academic systems like Oceanstore and Pastry, improve performances by moving data inside the network in a proactive way. The main advantage of these approaches is that peers can choose who they connect to and when to open or close a connection , based only on local information.

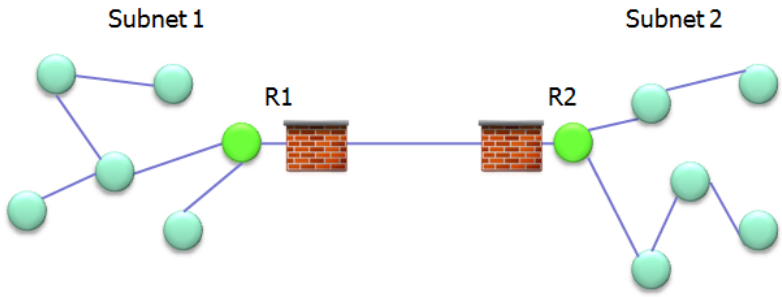

Figure A.2: Using relays to cross firewalls

A.1.7.2 Firewalls.

P2P applications naturally require direct connections between peers. However, in some administrative domains, internal networks are isolated from the external world, granting only restricted access to applications. For instance, most firewalls block incoming TCP connections. That is, a machine located behind a firewall shall not be accessible from outside the network. Worse, individuals often use, at home, a private IP address behind a router or a NAT (Network Address Translation) service to share an Internet connection between multiple machines. This leads to the issue of non-accessibility. However, as outgoing connections through port 80 (HTTP) are often allowed by firewalls, some mechanisms have been designed to allow communications between hidden machines (behind a firewall or a NAT system, unreachable from Internet). This procedure is somewhat limited as it requires the connection to be initiated by the "hidden" machine. When other communicating peers are located between different firewalls, the issue becomes even more complex. A "reflecting" server on Internet will be necessary to allow connections between hidden peers.(Figure A.2).

A.1.8 Fault Tolerance

One of the main characteristics of a P2P network is to avoid having any central weak points. Although most of (pure) P2P networks already do it, they still face some failures which are commonly linked to systems involving multiple hosts or networks : disconnections, unreachable hosts, network partitioning and nodes crashes. These failures can be more significant in some networks (for instance, wireless networks) than others (corporate cabled networks). It would be more interesting to pursue a collaborative work between peers still connected to the network when facing such failures. One relevant example could be an application, like Genome@Home [2001] running a distributed algorithm on multiple peers. Is it still possible to continue calculations if one of the peers disappears because of a network connection failure ? Similar questions have to be answered to by P2P systems willing to provide more than a "best-effort" Internet service.

In the past, client-server disconnections were studied for distributed filesystems including mobile devices (for instance, Coda [Satyanarayanan 1990]) , and one solution would be to have solvers proper to each application in order to deal with inconsistences after reconnections. Some recent P2P networks (Groove [Groove 2001],[Microsoft 2008]) define special nodes, called "relays", that temporarily memorize any update or communication until the receiver (here, another peer) reappears in the network. Others, like Magi [Bolcer 2000], delay messages at the source, until the receiving peer is detected.

A disconnection can result from a resources unavailability. This can occur when a machine becomes unreachable due to a network failure, or when the peer holding the resource crashes. Whereas the first case can be solved by routing information through an alternate path (a feature already provided by Internet), the second one requires greater attention. Crucial resources duplication can simplify the problem. Networks like Napster and Gnutella are systems allowing passive and uncontrolled duplication of data, based only on date's popularity. Depending on the application being run on these networks, it can be necessary to implement data persistency, by proposing a reliable data duplication policy.

Anonymous broadcasting services like Freenet [Clarke 2001] and Publius [Waldman 2000] ensure data availability through controlled replication. Oceanstore [Kubiatowicz 2000] introduces a system with two replication layers, and, through monitoring administrative domains, avoids to send replicas to sites with a high probability of failure. However, as a resource into a P2P network can be more than just a file (like proxies on Internet, online storage space, or shared computational power), the concepts of distributed data replication must be extended to other types of resources. Some solutions for distributed computing on grids (for instance, Legion [Grimshaw 1997]) propose fault tolerance for nodes by restarting calculations on other nodes.

One of the main challenges for a P2P system is that it manages its own maintenance, a task distributed over all the peers, to ensure resources availability. This is different from client-server systems, where resources availability is ensured by the server.

A.2 Existing Implementations

In this section, we propose an overview of the main existing P2P networks : Avaki, SETI@Home, Groove, Magi, FreeNet, Gnutella, JXTA, .NET, NaradaBrokering, Pastry.

Avaki [Avaki 2002] is designed to allow to see a network of heretogeneous resources as a single virtual machine. This is a classical example of "*meta-computing*" which is applied to networks ranging from corporate systems and data grids to

global computational grids over Internet. This is an object-oriented system. Each entity is considered as an addressable object with a set of methods and interfaces. It is mostly inspired from Mentat [Grimshaw 1993].

SETI (Search for ExtraTerrestrial Intelligence)) is a collection of projects aiming to discover extraterrestrial civilizations. One of these projects, SETI@Home, analyzes radio signals received from space and collected by the giant telescope in Arecibo, by using the computing power of millions of unused machines. [Anderson 2002].

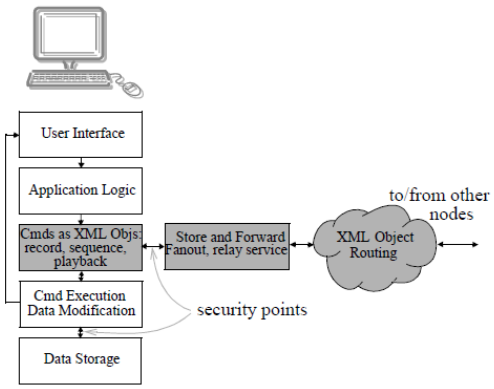

Figure A.3: Groove Architecture

Groove [Microsoft 2008] is a collaborative P2P system, which can be also considered as a platform. Groove mainly addresses to Internet users as well as intranet ones. It can also be used on mobile devices like PDAs, mobile phones or touchscreen tablets (Figure A.3). It enables communications, content sharing and proposes tools for joint activities. For instance, all Groove applications inherit from the integrated security mechanism without the need to redefine algorithms linked to security. This allows the deployment of safe applications.

Magi [Bolcer 2000] is a P2P infrastructure for designing secure, portable and collaborative applications. It uses standardized protocols like HTTP, WebDAV and XML to allow communications between applications inside corporate networks or Internet (Figure A.4). Magi Enterprise, the final product, builds an infrastructure which links computers of project teams to enable file sharing, instant messaging ....

FreeNet [Freenet 2001] is a P2P file sharing systems based on a model proposed by Ian Clarke [Clarke 2001]. The prime objective of FreeNet is to guarantee user's anonymity. That is, when logging into the system, a user must be able to submit requests without anyone discovering sender's identity. a FreeNet user has no knowledge about the content stored on its hard disk.

Gnutella [Osokine 2002] is a file sharing protocol. Applications implementing Gnutella allow users to search and download content from other users connected to Internet.

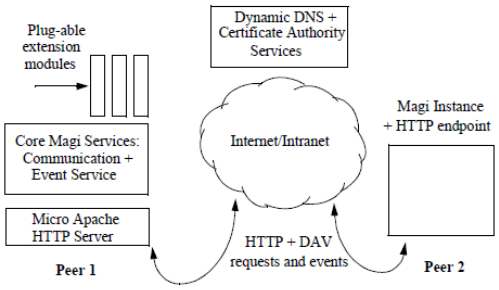

Figure A.4: Magi Architecture

The objective of the **JXTA** project [Daniel Brookshier 2002] is to propose an open and innovative collaboration platform which supports a wide range of distributed computing applications and allows to harness computing power from any device connected to the network [Gong 2001]. JXTA provides features on multiple layers, including basic mechanisms and concepts, high-level services that extend these mechanisms to a wide range of applications, which demonstrates the platform's potential.

NaradaBrokering [Fox 2005],[Pallickara 2003] is a technology that takes advantage from two distributed application systems : P2P systems and grid-based systems. Both systems have complementary advantages. This technology can be interfaced with a JXTA P2P network. The main idea behind NaradaBrokering is to provide users with a high quality network. To do so, NaradaBrokering uses techniques from distributed objects, web services and message-oriented middlewares. The architecture used is made of local networks of peers that are linked at large scale through a "core" interconnection network. To access the services, "brokers" are introduced. Messages-oriented middlewares are mainly used on this part of the network. Brokers are considered as the smallest unit of this network. They are in charge of processing and routing messages intelligently. The NaradaBrokering network is considered as a network made of brokers which cooperate with one another, whether they are located on client machines or not. To avoid having a loosely connected network, NaradaBrokering integrates a inter-broker communication protocol to manage links between brokers and manage the addition (or substraction) of brokers into/from the network. The network's organization is hierarchical, each broker belonging to a sub-group, forming greater groups with other sub-groups and so on. These first sub-groups have strongly interconnected brokers to guarantee at least one link even in case of failure. Thus, we have many small networks connected to one another. Network traffic increases by a logarithmic trend and not exponential like disorganized networks. NaradaBrokering also introduces Brokers Network Maps (BNM) that allow the intelligent routing feature described here above.

Pastry [Gendron 1994],[Castro 2002], [Rowstron 2001a] is a P2P system whose main characteristic is to propose a topology and a routing algorithm that allows to have a network traffic increasing logarithmically with the number of peers and avoids the definition of another type of peers as in NaradaBrokering.

Tapestry [Zhao 2001] is very similar to Pastry, the only difference being a variant in the routing algorithm. In our works, we inspired ourselves from Pastry, which we detail more thoroughly.

Publications

Book Chapters

- **M. Djamaï**, B. Derbel and N. Melab, "***Large Scale P2P-Inspired Problem Solving: A Formal and Experimental Study***", Large Scale Network-Centric Computing Systems, Wiley, 2012, *To be published.*

International Conferences

- **M. Djamaï**, B. Derbel and N. Melab, "***Impact of overlay properties upon a P2P approach for parallel B&B***", The 1st International Conference on Systems and Computer Science, August 29^{th}-31^{st} 2012, Villeneuve d'Ascq, France.

- **M. Djamaï**, B. Derbel and N. Melab, "***A Large-Scale Pure P2P approach for the B&B algorithm***", The Fourth IEEE International Scalable Computing Challenge (SCALE 2011) held in conjunction with The 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'2011), Newport Beach, USA, May 23^{rd}-26^{th}, 2011.

International Workshops

- **M. Djamaï**, B. Derbel and N. Melab, "***Distributed B&B : A Pure Peer-to-Peer Approach***", In Proceedings of the 25^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS) – Workshop on Large-Scale Parallel Processing (LSPP'11), Anchorage, Alaska, USA May 16^{th}-20^{th}, 2011.

- **M. Djamaï**, B. Derbel and N. Melab, "***Experimental Study of a P2P B&B approach on top of Grid'5000***",In Grid'5000 Workshop and Spring School, Reims, France, April 18^{th}-21^{st}, 2011.

- **M. Djamaï**, B. Derbel and N. Melab, "***Distributed Branch-and-Bound Algorithm : A Pure Peer-to-Peer Approach***" , In Grid'5000 Workshop and Spring School, Lille, France, April 6^{th}-9^{th}, 2010.

Bibliography

[Abadi 2000] I.N. Kamal Abadi, Nicholas G. Hall and Chelliah Sriskandarajah. *Minimizing Cycle Time in a Blocking Flowshop*. Operations Research, vol. 48, no. 1, pages 177–180, January/February 2000. (Cited on page 70.)

[Adamic 1999] Lada A. Adamic. *The Small World Web*. In 3^{th} European Conference on Research and Advanced Technology for Digital Libraries (ECDL'99), pages 443–452, London, UK, 1999. Springer-Verlag. (Cited on page 128.)

[Aida 2002] K. Aida and Y. Futakata. *High-performance parallel and distributed computing for the BMI eigenvalue problem*. In 16^{th} International Parallel and Distributed Processing Symposium, (IPDPS'02), pages 71 –78, 2002. (Cited on page 17.)

[Aida 2005] K. Aida and T. Osumi. *A Case Study in Running a Parallel Branch and Bound Application on the Grid*. In SAINT '05: The 5^{th} IEEE Symposium on Applications and the Internet, pages 164–173, Washington, DC, USA, jan. 2005. (Cited on page 17.)

[Allahverdi 1999] Ali Allahverdi, Jatinder N.D Gupta and Tariq Aldowaisan. *A review of scheduling research involving setup considerations*. Omega, vol. 27, no. 2, pages 219 – 239, 1999. (Cited on page 70.)

[Allahverdi 2004] Ali Allahverdi and Tariq A. Aldowaisan. *No-wait flowshops with bicriteria of makespan and maximum lateness*. European Journal of Operational Research, pages 132–147, 2004. (Cited on page 70.)

[Anderson 2002] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky and Dan Werthimer. *SETI@home: an experiment in public-resource computing*. Communications of the ACM, vol. 45, no. 11, pages 56–61, November 2002. (Cited on pages 11 and 20.)

[Andrés 2005] Carlos Andrés, José Miguel Albarracin, Guillermina Tormo, Eduardo Vicens and José Pedro Garcia-Sabater. *Group technology in a hybrid flowshop environment: A case study*. European Journal of Operational Research, vol. 167, no. 1, pages 272 – 281, 2005. (Cited on page 70.)

[Avaki 2002] Avaki. Avaki grid software: Concepts and architecture. Using Comprehensive Grid Software from AVAKI to Provide Wide-Area Access to Processing Power, Applications, and Data, Mars 2002. (Cited on page 130.)

[Awerbuch 1985] B. Awerbuch. *Complexity of network synchronization*. Journal of the ACM, vol. 32, no. 4, pages 804–823, 1985. (Cited on pages 29 and 36.)

[Bakken 2002] Dave Bakken. Paradigms for distributed fault tolerance, Février 2002. (Cited on page 94.)

[Barabasi 1999] Albert-Laszlo Barabasi and Reka Albert. *Emergence of Scaling in Random Networks.* Science, vol. 286, no. 5439, pages 509–512, 1999. (Cited on pages 85 and 87.)

[Barrat 2000] A. Barrat and M. Weigt. *On the properties of small-world network models.* The European Physical Journal B, Volume 13, Issue 3, pp. 547-560 (2000)., vol. 13, pages 547–560, January 2000. (Cited on page 85.)

[Bendjoudi 2009] 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, vol. 1, no. 2, pages 159–168, 2009. (Cited on pages 2, 3 and 19.)

[Bendjoudi 2011] Ahcene Bendjoudi, Nouredine Melab and El-Ghazali Talbi. *Fault-Tolerant Mechanism for Hierarchical Branch and Bound Algorithm.* 2011. (Cited on page 19.)

[Bendjoudi 2012] Ahcène Bendjoudi. *Scalable and Fault-Tolerant Hierarchical B&B Algorithms For Computational Grids.* PhD thesis, Université A.MIRA-BEJAIA Faculté des Sciences Exactes Département Informatique, 2012. (Cited on page 113.)

[Bendjoudi 2013] A. Bendjoudi, N. Melab and E-G. Talbi. *FTH-B&B: a Fault-Tolerant Hierarchical Branch and Bound for Large Scale Unreliable Environments.* IEEE Transactions on Computers, 2013. (Cited on page 119.)

[Bindel 2002] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, Ben Zhao and John Kubiatowicz. *OceanStore: An Extremely Wide-Area Storage System.* Rapport technique, Berkeley, CA, USA, 2002. (Cited on page 127.)

[Bittorrent 2005] Bittorrent. *Bittorrent Protocol Specification*, 2005. (Cited on pages 20 and 113.)

[Blumofe 1996] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou. *Cilk: An Efficient Multithreaded Runtime System.* Journal of Parallel and Distributed Computing, vol. 37, no. 1, pages 55–69, August 25 1996. (Cited on page 32.)

[Bolcer 2000] G. A. Bolcer. *Magi: an architecture for mobile and disconnected workflow.* vol. 4, no. 3, pages 46–54, May–June 2000. (Cited on pages 130 and 131.)

[Bonney 1976] M C Bonney and S W Gundry. *Solutions to the constrained flowshop sequencing problem.* Operational Research Quarterly, no. 27, page 869, 1976. (Cited on page 70.)

[Cabani 2007] A. Cabani, S. Ramaswamy, M. Itmi and J.-P. Pécuchet. *PHAC: An Environment for Distributed Collaborative Applications on P2P Networks*. In 6th International Conference on Distributed Computing and Internet Technology (ICDCIT), pages 240–247, 2007. (Cited on page 19.)

[Cappello 1997] Peter Cappello, Bernd Christiansen, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser and Daniel Wu. *Javelin: Internet-Based Parallel Computing Using Java*, 1997. (Cited on page 20.)

[Caromel 2007] Denis Caromel, Alexandre di Costanzo, Laurent Baduel and Satoshi Matsuoka. *Grid'BnB: a parallel branch and bound framework for grids*. In Proceedings of the 14th international conference on High performance computing, HiPC'07, pages 566–579, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 20.)

[Castro 2002] Miguel Castro, Peter Druschel, Y. Charlie Hu and Antony Rowstron. *Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks*, 2002. (Cited on pages 124 and 133.)

[Chandra 2000] R Chandra, R.M. Lefever, M Cukier and W.H. Sanders. *Loki: a state-driven fault injector for distributed systems*. In Proceeding International Conference on Dependable Systems and Networks. DSN 2000, pages 237–242. IEEE Comput. Soc, 2000. (Cited on page 107.)

[Clarke 2001] Ian Clarke, Oskar Sandberg, Brandon Wiley and Theodore W. Hong. *Freenet: A Distributed Anonymous Information Storage and Retrieval System*. Lecture Notes in Computer Science, vol. 2009, pages 46–50, 2001. (Cited on pages 20, 122, 124, 125, 126, 130 and 131.)

[Claudel 2009] Benoit Claudel, Guillaume Huard and Olivier Richard. *TakTuk, adaptive deployment of remote executions*. In Proceedings of the 18th ACM international symposium on High performance distributed computing, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM. (Cited on page 68.)

[Cung 1994] Dowaji Cung, Mautor Le Cun and Chris Roucairol. *Parallel and distributed branch-and-bound/A* algorithms*. Rapport technique, PRiSM Laboratory, Technical Report n94/31, Octobre 1994. (Cited on page 12.)

[Dakin 1965] R. J. Dakin. *A tree-search algorithm for mixed integer programming problems*. The Computer Journal, vol. 8, no. 3, pages 250–255, March 1965. (Cited on page 8.)

[Daniel Brookshier 2002] Brendon Wilson Daniel Brookshier Sing Li. *JXTA : P2P Grows Up*. Java Sun Technical Articles, vol. 1, pages 1–6, December 2002. (Cited on page 132.)

[Dawson 1996] Scott Dawson, Farnam Jahanian and Todd Mitton. *ORCHESTRA: A fault injection environment for distributed systems.* Ann Arbor, pages 1–30, 1996. (Cited on page 107.)

[Di Constanzo 2007] A. Di Constanzo. *Branch-and-bound with peer-to-peer for large-scale grids.* PhD thesis, Ecole doctorale STIC, Sophia Antipolis, France, Octobre 2007. (Cited on pages 2, 17, 18 and 20.)

[Dijkstra 1980] Edsger W. Dijkstra and C. S. Scholten. *Termination detection for diffusing computations.* Information Processing Letters, vol. 11, no. 1, pages 1–4, August 1980. (Cited on page 33.)

[Dingledine 2001] Freedman M. Rubin A. Dingledine R. Peer-to-peer. harnessing the power of disruptive technologies, chapitre Free Haven, pages 159–187. Oram A., 2001. (Cited on pages 124 and 126.)

[Drummond 2006] Lúcia M.A. Drummond, Eduardo Uchoa, Alexandre D. Gonçalves, Juliana M.N. Silva, Marcelo C.P. Santos and Maria Clícia S. de Castro. *A grid-enabled distributed branch-and-bound algorithm with application on the Steiner Problem in graphs.* Parallel Computing, vol. 32, no. 9, pages 629–642, October 2006. (Cited on pages 2 and 18.)

[Druschel 2001] Peter Druschel and Antony I. T. Rowstron. *PAST: A large-scale, persistent peer-to-peer storage utility.* pages 75–80, 2001. (Cited on pages 20 and 124.)

[Eckstein 2000] Jonathan Eckstein, Jonathan Eckstein, Cynthia A. Phillips, Cynthia A. Phillips, William E. Hart and William E. Hart. *PICO: An Object-Oriented Framework for Parallel Branch and Bound.* Rapport technique, Rutgers University, Piscataway, NJ, 2000. (Cited on page 18.)

[Erdös 1959] P. Erdös and A. Rényi. *On random graphs, I.* Publicationes Mathematicae (Debrecen), vol. 6, pages 290–297, 1959. (Cited on pages 85 and 87.)

[Erdos 1960] P. Erdos and A. Renyi. *On the evolution of random graphs.* Publ. Math. Inst. Hung. Acad. Sci, vol. 5, pages 17–61, 1960. (Cited on pages 85 and 87.)

[Fedak 2003] Gilles Fedak. *XtremWeb : une plate-forme pour l'étude expérimentale du calcul global pair-à-pair.* PhD thesis, Université Paris XI, 2003. (Cited on pages 11 and 12.)

[Finkel 1987] Raphael Finkel and Udi Manber. *DIB—a distributed implementation of backtracking.* ACM Trans. Program. Lang. Syst., vol. 9, no. 2, pages 235–256, 1987. (Cited on page 20.)

[Foster 1997] Ian Foster and Carl Kesselman. *Globus: A Metacomputing Infrastructure Toolkit.* International Journal of Supercomputer Applications, vol. 11, pages 115–128, 1997. (Cited on page 11.)

[Fox 2005] G. Fox and S. Pallickara. *Deploying the NaradaBrokering Substrate in Aiding Efficient Web and Grid Service Interactions*. vol. 93, no. 3, pages 564–577, March 2005. (Cited on pages 20 and 132.)

[Freenet 2001] Project Freenet. *Understand Freenet*, 2001. (Cited on page 131.)

[Frey 2002] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster and Steven Tuecke. *Condor-G: A Computation Management Agent for Multi-Institutional Grids*. Cluster Computing, vol. 5, no. 3, pages 237–246, 2002. (Cited on page 11.)

[Gabber 1999] Eran Gabber, Phillip B. Gibbons, David M. Kristol, Yossi Matias and Alain Mayer. *Consistent, yet anonymous, Web access with LPWA*. Commun. ACM, vol. 42, no. 2, pages 42–47, 1999. (Cited on pages 125 and 126.)

[Gangadharan 1993] Rajesh Gangadharan and Chandrasekharan Rajendran. *Heuristic algorithms for scheduling in the no-wait flowshop*. International Journal of Production Economics, vol. 32, no. 3, pages 285–290, 1993. (Cited on page 70.)

[Gendron 1994] B. Gendron and T. G. Crainic. *Parallel Branch-And-Bound Algorithms : Survey and Synthesis*. Operations Research, vol. 42, no. 6, pages 1042–1066, Nov-Dec 1994. (Cited on pages 8, 12, 14 and 133.)

[Grid'5000] Grid'5000. *Grid'5000 Website*. (Cited on page 62.)

[Grimshaw 1997] Andrew S. Grimshaw, Wm. A. Wulf and CORPORATE The Legion Team. *The Legion vision of a worldwide virtual computer*. Commun. ACM, vol. 40, no. 1, pages 39–45, 1997. (Cited on page 130.)

[Groove 2001] Networks Groove. *Groove Networks Prodict Backgrounder*. 2001. (Cited on page 130.)

[Hall 1996] Nicholas G. Hall and Chelliah Sriskandarajah. *A Survey of Machine Scheduling Problems with Blocking and No-Wait in Process*. Operations Research, vol. 44, no. 3, pages 510–525, May/June 1996. (Cited on page 70.)

[Huffaker 2002] Bradley Huffaker, Marina Fomenkov, Daniel J. Plummer, David Moore, K. claffy, Bradley Huffaker Marina Fomenkov and A. Background. *Distance Metrics in the Internet*. In in IEEE International Telecommunications Symposium, pages 200–2, 2002. (Cited on pages 92 and 119.)

[Iamnitchi 2000] A. Iamnitchi and I. Foster. *A problem-specific fault-tolerance mechanism for asynchronous, distributed systems*. In Proceedings of the 29^{th} International Conference on Parallel Processing, pages 4–13, 21–24 Aug. 2000. (Cited on pages 20 and 22.)

[Johnson 1954] S. M. Johnson. *Optimal two- and three-stage production schedules with setup times included.* Naval Research Logistics Quarterly, vol. 1, no. 1, pages 61–68, 1954. (Cited on page 8.)

[KING 1980] J. R. KING and A. S. SPACHIS. *Heuristics for flow-shop scheduling.* International Journal of Production Research, vol. 18, no. 3, pages 345–357, 1980. (Cited on page 70.)

[Kleinberg 2000a] Jon Kleinberg. *The Small-World Phenomenon: An Algorithmic Perspective.* pages 163–170, 2000. (Cited on page 85.)

[Kleinberg 2000b] Jon M Kleinberg. *Navigation in a small world.* Nature, vol. 406, no. 6798, page 845, 2000. (Cited on page 85.)

[Kubiatowicz 2000] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells and Ben Zhao. *OceanStore: an architecture for global-scale persistent storage.* SIGPLAN Not., vol. 35, no. 11, pages 190–201, 2000. (Cited on pages 20, 124, 127 and 130.)

[Kumar 1984] V. Kumar and L. N. Kanal. *Parallel Branch-and-Bound Formulations for And/or Tree Search.* IEEE Transactions on Pattern Analysis and Machine Intelligence,, vol. PAMI-6, no. 6, pages 768 –778, nov. 1984. (Cited on pages 12 and 22.)

[Lageweg 1978] B. J. Lageweg, J. K. Lenstra and A. H. G. Rinnooy Kan. *A General Bounding Scheme for the Permutation Flow-Shop Problem.* Operations Research, vol. 26, no. 1, pages 53–67, 1978. (Cited on page 8.)

[Lalami 2012] Mohamed Esseghir Lalami and Didier El-Baz. *GPU Implementation of the Branch and Bound Method for Knapsack Problems.* 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, vol. 0, pages 1769–1777, 2012. (Cited on page 119.)

[Land 1960] AH Land and AG Doig. *An automatic method of solving discrete programming problems.* Econometrica: Journal of the Econometric Society, vol. 28, no. 3, pages 497–520, 1960. (Cited on page 8.)

[Little 1963] G. Little, K. Murty, D. Sweyney and C. Karel. *An algorithm for the travelling salesman problem.* In Operations Research, 1963. (Cited on page 8.)

[Litzkow 1988] M. J. Litzkow, M. Livny and M. W. Mutka. *Condor-a hunter of idle workstations.* In Proc. th International Conference on Distributed Computing Systems, pages 104–111, 13–17 June 1988. (Cited on page 11.)

[Mans 1995] Bernard Mans, Thierry Mautor and Catherine Roucairol. *A parallel depth first search branch and bound algorithm for the quadratic assignment*

problem. European Journal of Operational Research, vol. 81, no. 3, pages 617 – 628, 1995. (Cited on pages 2 and 21.)

[Mattern 1987] Friedemann Mattern. *Algorithms for Distributed Termination Detection*. Distributed Computing, vol. 2, no. 3, pages 161–175, 1987. (Cited on page 34.)

[Maymounkov 2002] P. Maymounkov and D. Mazieres. *Kademlia: A peer-to-peer information system based on the XOR metric*. 2002. (Cited on pages 20 and 113.)

[Mehdi 2011] Malika Mehdi. *Parallel hybrid optimization methods for permutation based problems*. PhD thesis, Université Lille1 - Sciences et Technologies, Université du Luxembourg, 2011. (Cited on page 119.)

[Melab 1996] Nordine Melab, Nathalie Devesa, Marie-Paule Lecouffe and Bernard Toursel. *Adaptive Load Balancing of Irregular Applications - A Case Study: IDA* Applied to the 15-Puzzle Problem*. In IRREGULAR '96: Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems, pages 327–338, London, UK, 1996. Springer-Verlag. (Cited on page 15.)

[Melab 1997] Nouredine Melab. *Gestion de la granularité et régulation de charge dans le modèle P3 d'évaluation parallèle des langages fonctionnels*. PhD thesis, Lille 1, Grenoble, 1997. Th. : informatique. (Cited on page 15.)

[Melab 2005] Nouredine Melab. *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul*, Novembre 2005. HDR. (Cited on pages 11 and 12.)

[Melab 2012] Nouredine Melab, Imen Chakroun, Mohand-Said Mezmaz and Daniel Tuyttens. *A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem*. CoRR, vol. abs/1208.3933, 2012. (Cited on page 119.)

[Mezmaz 2005] M. Mezmaz, Melab N. and E.-G Talbi. *Towards a Coordination Model for Parallel Cooperative P2P Multi-objective Optimization*. In European Grid Conference (EGC'2005), pages 305–314, 2005. (Cited on pages 19, 20, 21, 22, 26, 30, 31 and 83.)

[Mezmaz 2007a] M. Mezmaz. *Une approche efficace pour le passage sur grilles de calcul de méthodes d'optimisation combinatoire*. PhD thesis, Université des Sciences et Technologies de Lille 1, Novembre 2007. (Cited on pages 1, 70 and 119.)

[Mezmaz 2007b] M. Mezmaz, N. Melab and E.-G. Talbi. *A Grid-enabled Branch and Bound Algorithm for Solving Challenging Combinatorial Optimization*

Problems. In 21st International Parallel and Distributed Processing Symposium, 2007. (IPDPS'07)., pages 1–9, March 2007. (Cited on pages 2, 19, 20, 21, 22, 23, 62, 79 and 90.)

[Microsoft 2008] Microsoft. *Groove Protocols Review*, Décembre 2008. (Cited on pages 130 and 131.)

[Miller 1993] D.L. Miller and J.F. Pekny. *The Role of Performance Metrics for Parallel Mathematical Programming Algorithms*. In ORSA J. Computing, volume 5, pages 26–28, 1993. (Cited on page 12.)

[Milojicic 2008] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins and Z. Xu. *Peer-to-Peer Computing*. Rapport technique, 2008. (Cited on pages 11, 113, 121 and 122.)

[Mittal 2004] Neeraj Mittal, S. Venkatesan and Sathya Peri. *Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies*. In Proceedings of the 18th Symposium on Distributed Computing (DISC, pages 290–304, 2004. (Cited on page 33.)

[Moran 2000] S. Moran and S. Snir. *Simple and efficient network decomposition and synchronization*. Theoretical Computer Science, vol. 243, no. 1-2, pages 217–241, 2000. (Cited on page 36.)

[Nguyen 2012] The Tung Nguyen and Didier El Baz. *Fault Tolerant Implementation of Peer-to-peer Distributed Iterative Algorithms*. In Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on, pages 137 –145, dec. 2012. (Cited on pages 2, 67, 82, 92 and 119.)

[Osokine 2002] S. Osokine. *Search Optimization in the Distributed Networks*. Internet, Octobre 2002. (Cited on page 131.)

[Pallickara 2003] Shrideep Pallickara and Geoffrey Fox. *NaradaBrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids*. In Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware, pages 41–61, New York, NY, USA, 2003. Springer-Verlag New York, Inc. (Cited on page 132.)

[Papadimitriou 1998] CH Papadimitriou and K Steiglitz. Combinatorial optimization: algorithms and complexity. 1998. (Cited on page 8.)

[Pfitzmann 1987] A Pfitzmann and M Waidner. *Networks without user observability*. Comput. Secur., vol. 6, no. 2, pages 158–166, 1987. (Cited on page 125.)

[Prieditis 1998] Armand Prieditis. *Depth-First Branch-and-Bound vs. Depth-Bounded IDA**. Computational Intelligence, vol. 14, no. 2, pages 188–206, 1998. (Cited on page 21.)

[Ralphs 2003] T.K. Ralphs, L. Ladanyi and M.J. Saltzman. *Parallel branch, cut, and price for large-scale discrete optimization*. Mathematical Programming, vol. 98, no. 1-3, pages 253–280, September 2003. (Cited on page 8.)

[Ramanathan 2002] Murali Krishna Ramanathan, Vana Kalogeraki and Jim Pruyne. *Finding Good Peers in Peer-to-Peer Networks*. In IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium, page 158, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on page 128.)

[Ratnasamy 2001] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp and Scott Schenker. *A scalable content-addressable network*. In SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, volume 31, pages 161–172. ACM Press, October 2001. (Cited on page 126.)

[Reddi 1972] S. S. Reddi and C. V. Ramamoorthy. *On the Flow-Shop Sequencing Problem with No Wait in Process[dagger]*. J Oper Res Soc, vol. 23, no. 3, pages 323–331, September 1972. (Cited on page 70.)

[Reiter 1998] Michael K. Reiter and Aviel D. Rubin. *Crowds: anonymity for Web transactions*. ACM Trans. Inf. Syst. Secur., vol. 1, no. 1, pages 66–92, 1998. (Cited on page 125.)

[Reza Hejazi 2005] S. Reza Hejazi and S. Saghafian. *Flowshop-scheduling problems with makespan criterion: a review*. International Journal of Production Research, vol. 43, no. 14, pages 2895–2929, 2005. (Cited on page 70.)

[Rhea 2001] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon and John Kubiatowicz. *Maintenance-Free Global Data Storage*. IEEE Internet Computing, vol. 5, no. 5, pages 40–49, 2001. (Cited on page 127.)

[Röck 1984] Hans Röck. *The Three-Machine No-Wait Flow Shop is NP-Complete*. J. ACM, vol. 31, no. 2, pages 336–345, March 1984. (Cited on page 70.)

[Rowstron 2001a] A. Rowstron and P. Druschel. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*. In Lecture Notes in Computer Science, pages 329–350, 2001. (Cited on pages 20, 127 and 133.)

[Rowstron 2001b] A. Rowstron and P. Druschel. *Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility*, 2001. (Cited on page 126.)

[Saffre 2003] Fabrice Saffre and Robert Ghanea-Hercock. *Beyond anarchy: self-organized topology for peer to peer networks*. Complexity, vol. 9, no. 2, pages 49–53, 2003. (Cited on page 19.)

[Sato 1997] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi and Satoshi Matsuoka. *Ninf: A network based information library for global world-wide computing infrastructure.* pages 491–502, 1997. (Cited on page 17.)

[Scarlata 2001] V. Scarlata, B.N. Levine and C. Shields. *Responder anonymity and anonymous peer-to-peer file sharing.* pages 272–280, Nov. 2001. (Cited on page 126.)

[Seymour 2002] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee and Henri Casanova. *Overview of GridRPC: A Remote Procedure Call API for Grid Computing.* pages 274–278, 2002. (Cited on page 17.)

[Shabtay 1994] Lior Shabtay and Adrian Segall. *Low Complexity Network Synchronization.* Proceedings of the 8th International Workshop on Distributed Algorithms.(WDAG '94), pages 223–237, 1994. (Cited on pages 29 and 36.)

[Shields 2000] Clay Shields and Brian Neil Levine. *A protocol for anonymous communication over the Internet.* In CCS '00: Proceedings of the 7th ACM conference on Computer and communications security, pages 33–42, New York, NY, USA, 2000. ACM. (Cited on page 125.)

[Stoica 2001] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications.* In SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, pages 149–160, New York, NY, USA, 2001. ACM. (Cited on pages 124 and 126.)

[Stott 2000] D.T. Stott, B Floering, D Burke, Z Kalbarczpk and R.K. Iyer. *NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors.* In Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000, pages 91–100. IEEE Comput. Soc, 2000. (Cited on page 107.)

[Stutzbach 2006] Daniel Stutzbach and Reza Rejaie. *Understanding churn in peer-to-peer networks.* Proceedings of the 6th ACM SIGCOMM on Internet measurement - IMC '06, page 189, 2006. (Cited on page 113.)

[Syverson 1997] Paul F. Syverson, David M. Goldschlag and Michael G. Reed. *Anonymous Connections and Onion Routing*, 1997. (Cited on page 125.)

[Taillard 1993] E. Taillard. *Benchmarks for basic scheduling problems.* European Journal of Operational Research, vol. 64, no. 2, pages 278 – 285, 1993. Project Management and Scheduling. (Cited on page 19.)

[Tanaka 2003] Y Tanaka, H Nakada, S Sekiguchi, T Suzumura and S Matsuoka. *Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing*. Journal of Grid Computing, vol. 1, no. 1, pages 41–51, 2003. (Cited on page 17.)

[Tanenbaum 2002] AS Tanenbaum and M Van Steen. Distributed systems: principles and paradigms. Prentice Hall PTR, 2002. (Cited on page 94.)

[V.K. Janakiram 1988] D.P. Agrawal V.K. Janakiram and R. Mehrotra. *A Randomized Parallel Branch-and-Bound Algorithm*. In in Proc. of Int. Cont. on Parallel Processing, pages 69–75, Août 1988. (Cited on pages 2 and 12.)

[Waldman 2000] Marc Waldman, Aviel D. Rubin and Lorrie Faith Cranor. *Publius: a robust, tamper-evident, censorship-resistant web publishing system*. In SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium, pages 5–5, Berkeley, CA, USA, 2000. USENIX Association. (Cited on pages 126 and 130.)

[Watts 1998] Duncan J. Watts and Steven H. Strogatz. *Collective dynamics of "small-world" networks*. 1998. (Cited on page 85.)

[Y. Xu 2005] L. Ladanyi M.-J. Saltzman Y. Xu T. K. Ralphs. *ALPS : A Framework for Implementing Parallel Search Algorithms*. pages 319–334, 2005. (Cited on page 18.)

[Zhang 2000] Weixiong Zhang. *Depth-first branch-and-bound versus local search: A case study*. In In Proc. 17th National Conf. on Artificial Intelligence (AAAI-2000, pages 930–935, 2000. (Cited on page 21.)

[Zhao 2001] Ben Y. Zhao, John D. Kubiatowicz and Anthony D. Joseph. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*. Rapport technique, Berkeley, CA, USA, 2001. (Cited on pages 127 and 133.)

Peer-to-Peer Branch-and-Bound Algorithms for Computational Grids.

Abstract:

In the field of Combinatorial Optimization, the resolution to optimality of large instances of optimization problems through the use of Branch-and-Bound algorithms require a huge amount of computational resources. Nowadays, such resources are available from computing grids, which are sets of computing nodes geographically distributed over multiple sites. These parallel environments introduces multiples challenges related to the scalability, the heterogeneity of resources and the fault tolerance. Most of the existing approaches for the Branch-and-Bound algorithm are based on the Master-Slave paradigm where a central entity shares work units among slave entities in charge of processing them. Such an architecture represents an obstacle to scalability. In this thesis, we propose to face the challenges of grid environments and overcome this limitation by proposing an innovative and fully distributed approach based on the Peer-to-Peer paradigm. This architecture is based on a unique type of entity, a peer which is in charge of exploring its own local work pool and broadcasts global information to the network. We provide mechanisms to deal with the main tasks of the Branch-and-Bound algorithm : the load balancing, the diffusion of the best solution and the detection of the termination. Along with extensive experiments conducted on the Flow-Shop Scheduling Problem using the Grid'5000 Experimental Grid, we propose a formal proof of the correctness of our approach.In addition to this, we tackle a central issue when designing a Peer-to-Peer application : the impact of the P2P network topology on the performance of our approach. This aspect is often ignored in most of existing works, where only a predefined organization is chosen for the peers. The obtained results showed that the approach allows to deploy computing networks at extreme scales, involving hundreds of thousands of computing cores. Our final contribution consists in a Fault-Tolerant approach to deal with the dynamicity of the network (the volatility of computational resources). Results indicate that it faces efficiently various real-case and failure-intensive situations.

Keywords: Peer-to-Peer Computing, P2P, Parallel Branch-and-Bound, Fault Tolerance, Grid Computing, Large Scale Networks, Topologies, Network Protocols, Flow-Shop Scheduling Problem, Grid'5000, Termination Detection, Combinatorial Optimization, Exact Methods.

Algorithmes Branch-and-Bound Pair-à-Pair pour grilles de calcul.

Résumé:

Dans le domaine de l'Optimisation Combinatoire, la résolution de manière optimale de problèmes de grande taille par le biais d'algorithmes Branch-and-Bound requiert un nombre très élevé de ressources de calcul. De nos jours, de telles ressources sont accessibles grâce aux grilles de calcul, composées de grappes de clusters réparties sur différents sites géographiques. Ces environnements parallèles posent de nombreux défis scientifiques, notamment en termes de passage à l'échelle, de la prise en compte de l'hétérogénéité des ressources ainsi qu'en termes de tolérance aux pannes. La plupart des approaches existantes pour l'algorithme Branch-and-Bound parallèle sont basées sur une architecture de type Maître-Esclave, où un processus maître répartit les tâches à accomplir auprès de processus esclaves en charge de les traîter. L'utilisation d'une telle entité centrale constitue un obstacle majeur en ce qui concerne le passage à l'échelle. Dans cette thèse, nous proposons de relever ces défis ainsi que de surmonter cet obstacle grâce à une approche innovante et complètement distribuée, basée sur une architecture Pair-à-Pair (P2P). Celle-ci repose sur un seul type de processus (le pair), qui a pour mission d'explorer son propre ensemble de tâches, de le partager avec d'autres pairs et de diffuser l'information globale. Nous définissons des mécanismes adaptés en lien avec l'algorithme Branch-and-Bound, qui traitent de la répartition de la charge, de la diffusion de la meilleure solution trouvée et de la détection de la terminaison des calculs. En plus de multiples expérimentations sur le problème d'ordonnancement du Flow-Shop sur la grille de calcul Grid'5000, nous proposons une preuve formelle de la correction de notre approche. Par ailleurs, nous traîtons une problématique souvent ignorée dans les travaux relatifs au calcul P2P, qui est l'importance de la topologie du réseau P2P. Généralement, une topologie très simple est utilisée. Les résultats obtenus montrent que notre approche permet le déploiement de réseaux de calculs à de très grandes échelles, constitués potentiellement de centaines de milliers de coeurs de calcul. Notre dernière contribution consiste en une approche Pair-à-Pair tolérante aux pannes afin de prendre en compte la nature généralement très volatile des ressources de calcul. Les résultats obtenus prouvent la robustesse de l'approche dans des environnements à la fois réalistes et sujets à de nombreux dysfonctionnements.

Mots-clés: Calcul Pair-à-Pair, P2P, Branch-and-Bound Parallèle, Tolérance aux pannes, Traitement réparti sur grille, Réseaux large-échelle, Protocoles réseau, Problème d'ordonnancement du Flow-Shop, Grid'5000, Détection de la terminaison, Optimisation Combinatoire, Méthodes exactes.