Calcul itératif asynchrone à grande échelle sur des architectures hétérogènes et volatiles

Thèse pour obtenir le grade de
Docteur de l'Université de Franche-Comté
Spécialité Informatique Automatique et Productique
par

Jean-Claude Charr

Laboratoire d'Informatique de l'Université de Franche-Comté
UFR Sciences et Techniques
Université de Franche-Comté

Soutenue le 16 septembre 2009 devant la commission d'examen:

Rapporteurs	**Nabil Abdennadher**	Professeur à l'Université des Sciences Appliquées de Suisse Occidentale
	Christian Perez	Chargé de Recherche (HDR) à l'Inria Lyon
Examinateurs	**Nordine Melab**	Professeur à l'Université des Sciences et Technologie de Lille
	Laurent Philippe	Professeur à l'Université de Franche-Comté
Directeur de thèse	**Raphaël Couturier**	Professeur à l'Université de Franche-Comté
Co-encadrant	**David Laiymani**	Maître de Conférences à l'Université de Franche-Comté

Remerciements

Je voudrais d'abord remercier mon directeur de thèse, Raphaël Couturier, pour m'avoir choisi pour être son étudiant ce qui m'a permis de continuer mes études, et pour son soutien et amitié durant les trois dernières années. Je voudrais aussi remercier mon encadrant, David Laiymani, pour ses efforts, conseils, et expertises sans lesquels le succès de ce travail n'était pas possible. Je voudrais ajouter que travailler avec ces deux était un plaisir et une des plus importantes expériences que j'ai vécues à l'université de Franche Comté.

Je voudrais remercier professeur Nabil Abdennadher et chargé de recherche à l'INRIA Christian Perez pour avoir accepté d'être rapporteurs de ma thèse. Leurs remarques et commentaires étaient très utiles pour améliorer l'organisation et le contenu de ce manuscrit.

Je voudrais remercier les examinateurs, Nordine Melab et Philippe Laurent, pour m'avoir aidé à réaliser cet exploit. Je voudrais aussi remercier professeur Nordine Melab pour m'avoir recruté pour effectuer un stage post-doctoral au sein de son équipe de recherche.

Je voudrais remercier tous mes collègues et amis à l'équipe AND avec lesquels j'ai eu le plaisir de travailler pendant trois ans: Abdallah Makhoul, Jacques Bahi, Arnaud GIERSCH, Jean-Luc ANTHOINE, Husam ALUSTWANI, Laurence PILARD, Michel SALOMON, Sébastien MIQUEE, Stéphane DOMAS, Mirna ESKANDAR, Jean-François COUCHOT et Mourad HAKEM. Je voudrais remercier aussi Philippe Vuillemin qui m'a aidé à poursuivre le développent de la plate-forme JACP2P.

Finalement, je voudrais remercier tous mes amis et ma famille qui m'ont encouragé et soutenus pour que je poursuis mes études en France.

Introduction

Le domaine de l'analyse numérique est antérieur à l'invention des ordinateurs modernes par de nombreux siècles. Beaucoup de grands mathématiciens étaient préoccupés par l'analyse numérique. Cela est évident d'après la nomination de certaines méthodes numériques importantes comme la méthode de Newton, l'élimination de Gauss, ou la méthode d'Euler. Dans le dernier siècle, le calcul numérique a été et est encore largement utilisé dans tous les domaines de la génie et des sciences physiques. A cet époque, les sciences de la vie et même les arts ont également adopté des éléments de calcul scientifiques. Par exemple, la résolution de grands problèmes linéaires est essentielle à la psychologie quantitative et le calcul de la solution aux équations différentielles stochastiques et chaînes de Markov sont essentielles pour simuler les cellules vivantes pour la médecine et la biologie. En outre, le calcul numérique a chaque jour un impact direct sur nos vies. En effet, grâce au calcul numérique, nous sommes en mesure de prédire la météo, concevoir des voitures plus sûres, calculer efficacement la trajectoire que doit être prise par un satellite, de calculer la dose de rayons X nécessaires pour traiter les patients souffrant de cancer, etc. Nous dépendons en grande partie sur le calcul numérique pour améliorer notre mode de vie.

Le calcul scientifique simule souvent des phénomènes réels tels que le changement climatique ou la fusion nucléaire qui génèrent des grands et complexes problèmes numériques. Pour que les simulations soient utiles, elles doivent être exécutées pendant des périodes de temps finies et relativement courtes. Les grands problèmes numériques exigent de grandes puissance de calcul pour être résolus. Ainsi, leurs solutions ne peuvent pas être calculées à la main et nous devons utiliser les ordinateurs. De même, si le problème est trop gros pour être résolu sur un seul ordinateur, car il n'a pas suffisamment de puissance de calcul, ni assez d'espace mémoire, nous utilisons les architectures distribuées comme les supercalculateurs, les grappes locales, les grilles, etc. Ces architectures distribuées sont composées de plusieurs unités de calcul et ils combinent la puissance de calcul de tous leurs unités de calcul pour résoudre un problème donné. Pour utiliser une telle architecture distribuée, le problème numérique de grande taille doit être décomposé en plusieurs petites tâches et une méthode de résolution parallèle doit être utilisé pour résoudre chaque tâche sur une unité de calcul. La parallélisation d'une application n'est pas un problème trivial, surtout si

les tâches qui en résultent dépendent les unes des autres et si les nœuds de calcul n'ont pas un espace de mémoire partagé. Voici les problèmes que les programmateurs doivent aborder durant la parallélisation et l'exécution d'une application sur une architecture distribuée :

- **Gérer les communications entre les nœuds de calcul** : dans certaines applications scientifiques, les sous-problèmes résolus sur des nœuds de calcul distincts, dépendent de certaines composantes situées dans d'autres sous-problèmes. Quand on veut programmer des méthodes parallèles capables de résoudre de telles applications scientifiques, les communications entre les nœuds voisins doivent être gérées avec soin afin de ne pas diminuer les performances globales de l'application (en particulier si on utilise des environnements à grande latence où les communications sont très pénalisantes). Pour ces raisons, les algorithmes à gros grains sont préférés aux algorithmes à grains fins lors de l'utilisation des architectures à grande latence parce qu'ils n'ont pas besoin d'échanger beaucoup de données entre les nœuds de calcul.

- **L'hétérogénéité des nœuds, des réseaux et des tâches** : les nœuds formant une architecture distribuée sont généralement hétérogènes, en particulier lors de l'utilisation les architectures de grappes distribuées ou les architectures de calcul global. Chaque nœud a des spécifications différentes. Ainsi, le développeur d'une application parallèle doit prendre en considération les capacités de chaque nœud et doit distribuer une charge appropriée à chacun afin de ne pas perdre une partie de la puissance de calcul et afin d'avoir des performances optimales. Pour ces mêmes raisons, les développeurs doivent prendre en compte l'hétérogénéité des réseaux (bande passante et latence) qui inter-connectent les unités de calcul durant la conception d'une application parallèle. En outre, les tâches ne sont pas toujours similaires et elles pourraient exiger différents montants de puissance de calcul. La solution optimale doit attribuer chaque tâche de calcul à l'unité de calcul appropriée liée au réseau approprié.

- **Les synchronisations entre les nœuds de calcul** : Lors de l'échange de données ou du calcul d'une fonction de réduction ou de la détection de la convergence globale (si la méthode est itérative), les nœuds de calcul généralement se synchronisent localement ou globalement. Cela conduit à une perte de puissance de calcul au cours de ces synchronisations. En outre, comme les nœuds et les réseaux dans les architectures distribuées sont hétérogènes, les nœuds de calcul performants doivent attendre que les nœuds moins performants finissent leurs tâches avant de pouvoir communiquer avec eux. Ce qui résulte à des périodes d'inactivité pour les unités de calcul plus performantes.

- **Le calcul des fonctions de réduction** : Certaines méthodes numériques exigent le calcul de fonctions de réduction. Ces fonctions sont généralement calculées en utilisant des données provenant de tous les nœuds de calcul. La plupart des mécanismes pour le calcul des fonctions de réduction mènent à la synchronisation de tous les nœuds de calcul ou la centralisation de l'opération sur un seul nœud ce qui réduit la performance de l'application parallèle.

- **La détection de la convergence globale** : Ce problème concerne uniquement les méthodes itératives qui exécutent le même bloc d'instructions jusqu'à ce que leur résidu est inférieur à la précision demandée (ϵ). Plusieurs mécanismes ont été mis en œuvre pour détecter la convergence globale des méthodes itératives et parallèles. Cependant, la plupart d'entre eux sont centralisées et nécessitent la synchronisation de tous les nœuds ce qui réduit les performances des applications parallèles, surtout si elles sont exécutées sur des unités de calcul hétérogènes .

- **La volatilité des unités de calcul** : Les nœuds de calcul dans les architectures distribuées ne sont pas très stables, en particulier dans l'architecture de calcul volontaire où les nœuds de calcul sont généralement des machines publics, inutilisées qui peuvent être déconnecté à tout moment. Pour être capable d'exécuter des applications parallèles sur de tels environnements volatils, le développeur doit concevoir un mécanisme de détection des pannes et un mécanisme de restauration. Le mécanisme de détection des pannes est nécessaire pour détecter les nœuds de calcul déconnectés afin de les remplacer. Le mécanisme de restoration permet au nouveau nœud de calcul, qui a remplacé le nœud en panne, de poursuivre l'exécution de la tâche sans devoir recommencer dès le début. Ainsi, le système devient tolérant aux pannes. Il existe plusieurs systèmes pour assurer cette propriété, mais la plupart d'entre eux ont besoin de synchroniser tous les nœuds de calcul ou ont besoin d'un serveur de stockage stable pour sauvegarder régulièrement l'état et les données des nœuds de calcul. Ces méthodes ajoutent généralement une surcharge considérable sur les nœuds de calcul ce qui peut réduire les performances globales des applications parallèles.

- **Les centralisations** : Comme les architectures distribuées sont volatiles, la centralisation de n'importe quel mécanisme sur n'importe quel nœud est très dangereuse parce qu'elle crée des points faibles dans la plate-forme qui ne sont pas tolérants aux pannes. En outre, la centralisation limite l'extensibilité des applications parallèles. En effet, en utilisant un grand nombre de nœuds de calcul, les nœud centralisateurs seront surchargés et vont se planter. Donc, tous les mécanismes centralisés, tels que la détection de la convergence globale, la détection des pannes et la restauration des nœuds en panne, doivent être décentralisés afin d'avoir une application extensible et stable qui peut être exécuté en parallèle dans des environnements distribués avec des nœuds volatils.

Dans ce document, nous nous abordons tous ces problèmes en proposant la plate-forme JACEP2P-V2 et des méthodes itérative et parallèles basée sur le modèle d'itérations asynchrones. Le modèle d'itérations asynchrones est bien adapté aux architectures hétérogènes et volatiles: il élimine les synchronisations entre les nœuds de calcul et tolère la perte des messages de données.

JACEP2P-V2

Notre contribution principale dans ce document est la plate-forme JACEP2P-V2 (Java Asynchronous Computing Environment for P2P architectures Version 2), qui est une plate-forme décentralisée et tolérante aux pannes. Elle est dédiée à l'exécution des applications itératives parallèles basées sur le modèle d'itérations asynchrones sur des architectures hétérogènes et volatiles. Il offre toutes les fonctionnalités nécessaires pour pouvoir exécuter ce type d'algorithmes (comme le multi-threading, l'échange asynchrone des messages de données et un mécanisme décentralisé pour la détection de la convergence globale. En outre, l'utilisation de cette plate-forme avec le modèle d'itérations asynchrones nous permet de résoudre les problèmes mentionnés ci-dessus, comme éliminer les synchronisations et centralisations et résister aux pannes. Plusieurs expérimentations ont été menées sur des architectures distribuées et volatiles afin de confirmer l'efficacité de cette approche et d'évaluer sa robustesse, ses performances et son extensibilité.

La comparaison de plusieurs méthodes de résolution de problèmes à valeurs initiales

Nos recherches ont également porté sur la conception de méthodes itératives et parallèles pour la résolution des problèmes à valeurs initiales, bien adaptées aux architectures hétérogènes, distribuées et avec grande latence dans les communications. Étant donné que les communications dans de telles architectures sont très pénalisantes, il faut utiliser les méthodes à gros grains. Dans notre travail, nous avons étudié de nombreuses méthodes parallèles itératives à gros grains (par exemple la méthode de relaxation d'onde, le méthode de Multisplitting, etc) qui sont compatibles avec le modèle d'itérations asynchrones. Ces méthodes ont été comparées tout en résolvant de grands problèmes numériques sur des architectures hétérogènes et volatiles en utilisant la plate-forme JACEP2P-V2.

Le reste de ce document est divisé en deux parties : le contexte scientifique et les contributions. Chaque partie est aussi composée de deux chapitres.

- Dans le premier chapitre, nous présentons brièvement les méthodes directes et itératives. Puis, nous présentons les architectures distribuées utilisées couram-

ment dans le calcul numérique et les différents environnements mis en œuvre pour gérer ces architectures. Nous terminerons ce chapitre par la présentation de certains mécanismes de détection de pannes et de récupération après panne.

- Dans le deuxième chapitre, nous présentons le modèle d'itérations asynchrones et nous montrons ses avantages dans un environnement hétérogène et volatil. Puis, nous présentons deux plates-formes dédiées à la conception et l'exécution de méthodes parallèles et itératives programmées selon ce modèle. Les limites de ces plates-formes sont expliquées.

- Dans le troisième chapitre, nous présentons notre plate-forme JACEP2P-V2. C'est une évolution de la plate-forme JACEP2P avec de nombreuses améliorations. Ce chapitre est consacré à la présentation de l'architecture de la plate-forme et de ses différentes caractéristiques.

- Les expérimentations sont toutes regroupées dans le dernier chapitre. Elles sont divisées en deux séries. La première série d'expériences évalue la performance, la robustesse et l'extensibilité de JACEP2P-V2. La deuxième série d'expériences compare différentes méthodes parallèles et itératives pour la résolution de problèmes à valeurs initiales. La plupart des expérimentations sont effectuées sur des environnements volatils et hétérogènes.

On termine ce document avec une conclusion et quelques perspectives.

Acknowledgments

I would like first to thank my advisor, Raphaël Couturier, for taking me on as his student and allowing me to pursue my research interests, and for his support and friendship during the course of this work. I would also like to thank my collaborator David Laiymani, whose efforts, expertise and advice were essential for the success of this work. I would like to add that working with these two proved to be one of my best learning experiences at Franche-Comte's university.

I would like to thank professor Nabil Abdennadher and INRIA researcher Christian Perez for accepting to review on a short notice this document (During their summer vacation). Their suggestions and advice were very helpful in improving the organization and content of this document.

I would like to thank my committee members Nordine Melab and Philippe Laurent for helping me making this happen. I would also like to thank professor Nordine Melab for hiring me for a post-doctoral position in his research team.

I would like to thank my many colleagues and friends at the AND team, with whom I had the pleasure of working over three years. These include Abdallah Makhoul, Jacques Bahi, Arnaud GIERSCH, Jean-Luc ANTHOINE, Husam ALUSTWANI, Laurence PILARD, Michel SALOMON, Sébastien MIQUEE, Stéphane DOMAS, Mirna ESKANDAR, Jean-François COUCHOT and Mourad HAKEM. I would like to give a special thanks to Philippe Vuillemin who helped me understand the JACEP2P platform in order to continue its development.

Finally, I would like to thank all the people back home who encouraged and supported me. These include of course my family who motivated and inspired me to give my best efforts in this project.

Contents

List of Figures

Introduction

The field of numerical analysis predates the invention of modern computers by many centuries. Many great mathematicians of the past were preoccupied by numerical analysis, as is obvious from the names of important numerical methods like Newton's method, Gaussian elimination, or Euler's method. In the last century, numerical computing was and is still heavily used in all fields of engineering and physical sciences. Now, the life sciences and even the arts have also adopted elements of scientific computations. For example, solving large linear problems is essential to quantitative psychology and finding the solution to stochastic differential equations and Markov chains are essential in simulating living cells for medicine and biology. Moreover, numerical computing impacts our lives each day. Indeed, thanks to numerical computing we are able to predict weather, conceive safer cars, compute efficiently the trajectory that must be taken by a satellite, compute the dose of X-Ray required to treat a patient suffering from cancer, etc. We heavily depend on numerical computing to improve our lives.

Scientific computing often simulates real-world changing conditions such as climate change, nuclear fusion, which generate large and complex numerical problems. In order for the simulations to be useful, they must be executed in finite and relatively short periods of time. Large numerical problems require large amounts of computing power to be solved. Thus, their solutions cannot be computed by hand and we have to use computers. Furthermore, if the problem is too large to be solved on a single computer because it does not have sufficient computing power nor enough memory space, we use distributed architectures like supercomputers, local clusters, grids, etc. These distributed architectures are composed of many computing units and they combine the computing power of all their computing units to solve a given problem. To use such distributed architectures, the large numerical problem must be decomposed into small tasks and a parallel resolution method must be used to solve each task on a different computing unit. Parallelizing an application is not a trivial problem, especially if the resulting tasks depend on each others and if the computing nodes do not have a shared memory space. Here are the most common issues that developers face while parallelizing and executing an application on a distributed architecture:

- **Managing the communications between all the computing nodes**: in some scientific applications, the subproblems, being solved on each node, depend on the values of some components located in other subproblems. When implementing parallel methods that solve such scientific applications, the communication between computing neighbors must be managed carefully in order not to decrease the overall performances (especially in high latency environments where communications are highly penalizing). In this way, coarse grained algorithms are preferred over fine grained algorithms when using high latency architectures because they do not require exchanging a lot of data between the computing nodes (comparing to the volume of computations).

- **Heterogeneous computing nodes, networks and tasks**: the nodes that form a distributed architecture are usually heterogeneous, especially when using distributed clusters or global computing architectures. Each node has a specific computing power and memory space. Thus, the developer of a parallel application must take into consideration the capacities of each node and must distribute the appropriate load to each one in order not to lose significant computing power and to have optimal performances. In the same way, developers have to take into account the heterogeneity of the networks (bandwidth and latency) interconnecting the computing units while conceiving the parallel application. Moreover, the tasks are not always similar and they could require different amounts of computing power. The optimal solution must assign each task on the appropriate computing unit connected to the appropriate network.

- **Synchronizations between nodes**: when exchanging data or computing a reduction function or detecting the global convergence (if the method is iterative), the computing nodes usually synchronize locally or globally with each other. This leads to a loss of computing power during these synchronizations. Moreover, since the nodes and the networks in distributed architectures are heterogeneous, the fast computing nodes must wait for the slow ones to finish their tasks before being able to communicate with them. This also results in larger idle times for computing units.

- **Computing reduction functions**: some numerical methods require computing some reduction functions. These functions are usually computed using data from all the nodes. Most of the schemes that compute these functions lead to synchronizing all the nodes or centralizing the operation on one node which reduces the performance of parallel applications.

- **Detecting the global convergence**: this problem is only related to iterative methods which execute the same bloc of instructions until the "residual vector" is lower than a requested precision (ϵ). Many schemes have been implemented to detect the global convergence of parallel iterative methods. However, most of

them are centralized and require synchronizing all the nodes which reduces the performance of parallel applications, especially if they are being executed on heterogeneous computing units.

- **Volatility of the computing nodes**: the computing nodes in distributed architectures are not very stable, especially in volunteer computing architectures where the computing nodes are generally public unused machines that can be disconnected at any time. To be able to execute parallel applications on such volatile environments, the developer must conceive a crash detection mechanism and a restoring mechanism. The crash detection mechanism is required to detect dead computing nodes in order to replace them. The restoring mechanism allows the new computing node that replaced the dead one to continue its task without restarting the task from the beginning. Thus, the system becomes fault tolerant. There are many schemes to ensure this property, but most of them require synchronizing all the computing nodes or need a safe storing area to store the status and data of computing nodes at regular time intervals. These methods usually add a considerable overhead on computing nodes and reduce the overall performance of parallel applications.

- **Centralization**: since distributed architectures are volatile, centralizing any mechanism on any node is very dangerous because it creates weak points in the platform which are not fault tolerant. Moreover, centralizing limits the scalability of the parallel application. Indeed, as the number of computing nodes increases, the centralizing node will eventually be overloaded and will crash. So all the centralized mechanisms, such as detection of global convergence, detection of dead nodes and restoring dead nodes, must be decentralized in order to have a safe scalable parallel application that runs well in volatile distributed environments.

In this document, we tackle all these issues by proposing the JACEP2P-V2 platform and some parallel iterative methods based on the asynchronous iteration model. The asynchronous iteration model is well adapted for heterogeneous volatile architectures: it eliminates synchronizations between the computing nodes and tolerates the loss of data messages.

JACEP2P-V2

Our main contribution in this document is JACEP2P-V2 (Java Asynchronous Computing Environment for P2P architectures Version 2) which is a decentralized fault tolerant platform dedicated to executing parallel iterative applications based on the asynchronous iteration model over volatile heterogeneous architectures. It provides all the functionalities necessary for running this type of algorithms like multi-threading, asynchronous message exchange and a decentralized global convergence detection.

Moreover, this platform combined with the asynchronous iteration model tackle many of the issues cited above, like eliminating synchronizations and centralizations and resisting to crashes. Many experiments have been conducted over distributed volatile architectures to prove the efficiency of this approach and to test its robustness, performances and scalability.

Comparative study between various resolution methods

Our researches were also focused on designing parallel iterative methods to solve linear and nonlinear systems that are well adapted to heterogeneous high latency distributed architectures. Since the communications in such architectures are very penalizing, only coarse grained methods must be used. In our work, we have studied many parallel iterative coarse grained methods (for example the Waveform Relaxation method, the Multisplitting method, etc.) which are compatible with the asynchronous iteration model. These methods were compared while solving large numerical problems over distributed heterogeneous volatile architectures using the JACEP2P-V2 platform.

The rest of this document is divided into two parts: scientific context and contributions. Each part is also composed of two chapters.

- In the first chapter, we briefly present the direct and the iterative methods. Then, we present the distributed architectures commonly used in numerical computing and the different environments implemented to manage these architectures. We end this chapter with the presentation of some mechanisms for fault detection and recovery.
- In the second chapter, we present the asynchronous iteration model and we show its benefits in a heterogeneous volatile environment. Then we present two platforms that are dedicated to designing and executing parallel iterative methods implemented according to this model. The limits of these platforms are discussed.
- In the third chapter, we present our platform JACEP2P-V2. It is an evolution of the JACEP2P platform with many improvements. This chapter is dedicated to presenting the architecture of the platform and its various features.
- The experimentations are all grouped in the last chapter. They are divided into two sets. The first set of experiments tests the performance, robustness and scalability of JACEP2P-V2. The second set of experiments compare various parallel iterative resolution methods while solving large numerical problems. Most of the experiments are executed over heterogeneous volatile environments.

We end this document with some conclusions and perspectives.

Part I

Scientific context

Chapter 1

Numerical computing on distributed architectures

1.1 Introduction

Humans are the only specie on earth capable of solving problems that require reflection and where the solution cannot be found by only using instincts. This intellectual capacity varies between individuals and can be developed if the individual receives a proper training which is usually done through education. The performance of this gift depends on the state of the individual: for example, it could be reduced or increased, if the individual is tired, sleepy or under a lot of pressure. Although, these intellectual capacities can be improved, they are definitely limited because the human brain can assimilate a relatively small amount of information at a small time period and it has a relatively small memory. Moreover, if a human being knows the method to solve a class of problems, it is almost impossible that he/she gives all the time the correct solution for problems belonging to this class because there is a high probability that he/she will make a common mistake when executing the resolution method. For all these reasons, humans have tried to create some sort of computing unit that is capable, once given the resolution method, of solving large and complex problems quickly and giving the correct solution without being affected by the surrounding environment. One of the first created computing units was the calculator. It only executed small mathematical operations but it already computed the solution of a mathematical problem a lot faster than a regular human being and it gave perfectly reliable solutions. As the problems get bigger, scientists developed faster computing units to solve them.
The first computing unit was a mechanical calculator built by Wilhelm Schickard in 1623. It was called the "Calculating Clock" because it used techniques such as cogs and gears that were at first developed for clocks. In the late 1940s, the first electronic computer was produced and it used vacuum tubes in the logic circuits which were later replaced by transistors in the 1950s. By the year 1958, the integrate circuit was discov-

ered by Jack Kilby. It allowed the integration of large numbers of tiny transistors on a small chip which reduced the cost and increased the performance of computers. This invention revolutionized the world of electronics. In accordance to Moore's law, the number of transistors placed on an integrated circuit has increased exponentially and in 2006 it was possible to fit about 1 million transistors per mm^2. In parallel to this huge evolution in the hardware capacities of computers, the resolution methods executed on these machines had to evolve in order to fully benefit from the increasing power of the computing units.

The aim of this chapter is to present an exhaustive state of the art on distributed numerical computing. In the second section, we present the various classes of numerical methods that are used to solve numerical problems. If the numerical problems are too large and cannot be solved on one computing unit, they must be parallelized and executed on parallel architectures. The various types of parallel architectures are described in section 3 where their advantages and drawbacks are also detailed. In section 4, we present many environments that allow users to easily execute parallel methods over the parallel architectures. These environments are classified according to which type of parallel architectures they are adapted. The fifth section presents the different mechanisms used to make the previously described environments fault tolerant which is very essential for architectures composed of volatile computing units. Finally, we end this chapter with a discussion about which type of parallel architecture suits well our research objectives.

1.2 Numerical methods

It has been said that almost all problems can be represented by some mathematical equations with relations between the different components that form the system. Many of these mathematical representations can be transformed into numerical problems and solved by numerical resolution methods. Numerical methods can be used to solve for example: systems of equations, eigenvalue problems, singular problems, differential equations... These methods can be divided into two classes: direct and iterative methods.

- **Direct methods** give the exact solution of a problem after executing a finite number of operations. We can cite for instance: the LU or the Cholesky method. Algorithm 1.1 presents the general steps for solving a linear system $Ax = b$ (where A is a n by n matrix and b and x are two vectors of dimension n) using the Cholesky direct method. It is composed of a finite number of operations and it returns the exact solution for the problem.

- **Iterative methods** compute many times the same block of operations until obtaining a good approximation of the solution. We then say that the method has

Algorithm 1.1 The Cholesky direct method

Require: Matrix $A_{n,n}$ and vector b
{Solve the linear system $Ax = b$ using the Cholesky direct method}
if A is symmetric and positive definite **then**
for k=1 step 1 until k=n-1 **do** {Compute the Cholesky decomposition $A = LL^T$ where L is a lower triangular matrix}
$l_{kk} \leftarrow \sqrt{a_{kk}}$
for s=k+1 step 1 until s=n **do**
$l_{sk} \leftarrow a_{sk}/l_{kk}$
end for
for j=k+1 step 1 until j=n **do**
for i=j step 1 until i=n **do**
$a_{ij} \leftarrow a_{ij} - l_{ik}l_{jk}$
end for
end for
end for
$l_{nn} \leftarrow \sqrt{a_{nn}}$
$y \leftarrow L^T x$
Compute vector y by solving the system $Ly = b$ using forward and back substitutions
Compute vector x by solving the system $L^T x = y$ using forward and back substitutions
end if
{Give the exact solution vector x after a finite number of operations}

converged to the solution of the problem. As an example, we can cite the Jacobi or the Conjugate Gradient method. Algorithm 1.2 presents the general steps for solving a linear system $Ax = b$ using the Jacobi iterative method. It executes until convergence the same bloc of instructions using the vector solution (x^{k-1}) computed at the previous iteration. At each iteration, it detects if the computed vector (x^k) has converged to the solution of the problem. At convergence the iterative process is terminated.

Algorithm 1.2 The Jacobi iterative method

Require: Matrix A, vector b, initial guess vector x^0 and the threshold ϵ
{Solve the linear system $Ax = b$ using the Jacobi iterative method}
for $k = 1$ step 1 until convergence **do**
for $i = 1$ step 1 until $i = n$ **do** {Compute vector x^k}
$x_i^{(k)} \leftarrow \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)})$
end for
Detect convergence
end for
{Give an approximation to the solution vector x after convergence}

To detect the local convergence of an iterative process, the residual R is computed at the end of each iteration. If the value of the residual is less than the required threshold (ϵ) then the iterative process has converged to the solution of the problem and the demanded precision has been met. There are many ways to compute the residual value. It depends directly on the distance measuring method used and on the components involved in this computation. A common scheme is to measure the distance between the solution vectors of two successive iterations using the Euclidean metric. In this approach, the residual value is computed as follows: $R = \sqrt{\Sigma_{i=1}^{n} |x_i^{k+1} - x_i^k|^2}$ where x_i^k denotes the value of the component i of the solution vector x at iteration k. The system converges when R is smaller than the requested threshold ϵ. To increase the precision of the results given by an iterative method, the developer must choose a smaller threshold.

Some problems can only be solved by iterative methods and when the problems are large, iterative methods are generally preferred over direct ones, especially if they give good approximations after small numbers of iterations.

Both classes of numerical methods presented in this section are called sequential methods because they are executed by a single computing unit. All the instructions are executed sequentially by the processor, the one after the other. These methods are easy to implement but they can benefit from the capacities of just one computing unit. Large problems cannot be solved by these methods on a single computing unit due to its limited computing power and memory. Therefore, many computing units must be used in parallel in order to solve large problems. A method that solves a given problem on many computing units is called a parallel method and the computing units, being used, form a parallel architecture. The different existing parallel architectures are presented in the next section.

1.3 Parallel architectures

Although desktop computers have become very powerful, many large complex numerical problems cannot be solved using only one computing unit, especially if those have resulted from the simulation of big natural phenomena like climate change. To solve these huge problems, a larger sources of computing power must be used. This could be ensured by a supercomputer or some sort of distributed architecture that combines the power of many small computing units.

- **Supercomputers** are known to have huge processing capacities. The first supercomputers were simply very fast scalar processors which reached about ten times the speed of the fastest machines available at that time. In the 1970s, supercomputers were transformed into vector processors and in the 1980s a supercomputer

could be composed by four to sixteen vector processors working in parallel. In the beginnings of the 21^{st} century, attention turned from vector processors to massive parallel processing systems with thousands of server-class microprocessors, such as the PowerPC, Opteron, or Xeon, all forming a single computing unit. The currently fastest supercomputer is the IBM Roadrunner [62], located at the Los Alamos national laboratory. It is composed of 6912 Opteron processors and 12960 PowerXCell processors. It is designed for a peak performance of 1.7 petaflops and it cost around 133 millions of US dollars. Supercomputers are very expensive and unaffordable by small research centers. This inconvenient pushed scientists into the development of new parallel architectures that use available low cost computing units.

- **Local clusters** are composed of regular computers that are located in a relatively small area. The computing units are usually homogeneous: they have similar specifications and similar configurations. All the nodes are interconnected via a local network with low latency and large bandwidth. Local clusters have limited computing power due to the limited number of regular computers that form them.

- **Distributed clusters** are composed of many interconnected local clusters that are geographically distant. These clusters are usually heterogeneous: two nodes from distinct clusters may have different specifications and different configurations. This distributed architecture is presented in Figure 1.1 where three heterogeneous clusters are interconnected via a wide area network to form a distributed clusters architecture. Although this architecture has huge computing resources, it suffers from high latency communications between two computing nodes from two distant clusters. Furthermore, when executing a lengthy application on many computing nodes, some volatility problems might occurred. So the platform executing parallel applications on such architectures must be fault tolerant.

  Grid'5000 [44] is a good example of this type of distributed architectures. This french national grid is dedicated for research experiments in large-scale parallel and distributed systems. It is currently composed of about 6320 heterogeneous cores (Nowadays, most of the processors are multicores. They combine two or more independent cores into a single package composed of a single integrated circuit, called a die. For example, a dual-core processor contains two cores, and a quad-core processor contains four cores) that are located in 9 sites in France: Bordeaux, Toulouse, Grenoble, Sophia Antipolis, Lyon, Lille, Rennes, Nancy and Orsay. The location of these sites in France is illustrated in Figure 1.2. Most of those sites have a Gigabit Ethernet Network for local machines. Links between the different sites range from 2.5 Gbps up to 10Gbps. Many softwares have been

developed in order to control and use the nodes forming this distributed architecture, for example OAR [59] which is a resource manager (or batch scheduler) for large distributed architectures. It allows users to submit or reserve nodes either in an interactive or a batch mode.

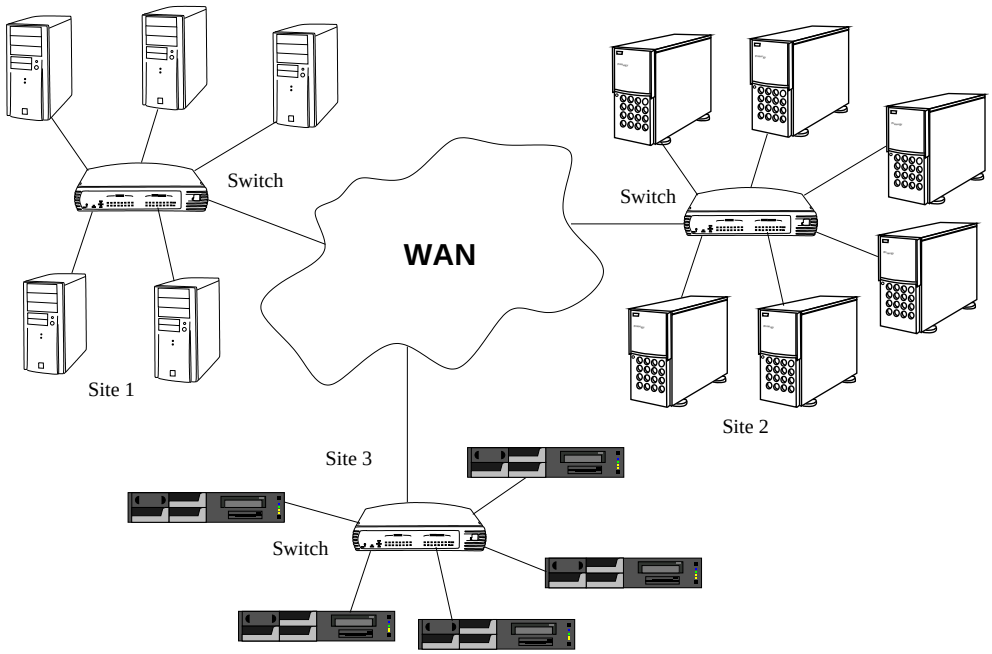

Figure 1.1: Distributed clusters architecture

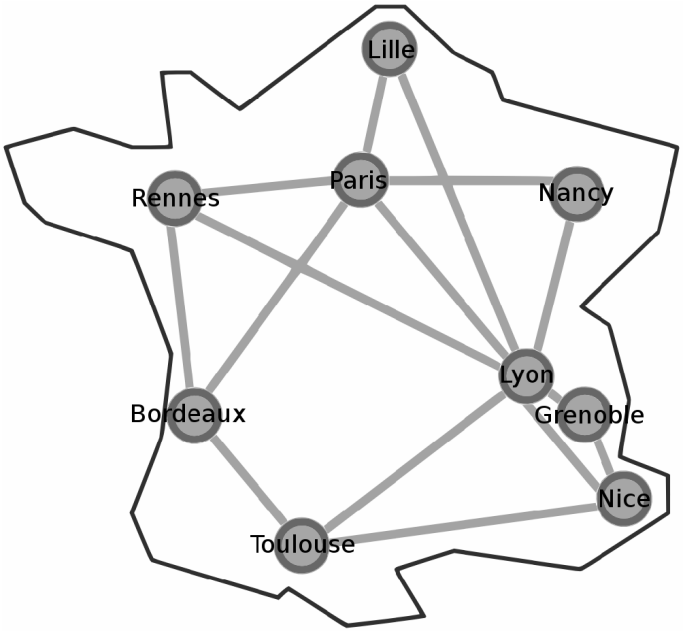

Figure 1.2: The location of Grid5000's sites in France

- **Global/volunteer computing architectures** are mainly composed of public unused computing units connected to the Internet. Figure 1.3 illustrates the heterogeneous components forming this architecture and their interconnection through the Internet. Nowadays, a small part of the computing power of the computing units is used. They spend a lot of idle times waiting for new instructions to execute. During these periods of inactivity, a lot of operations can be computed. If individuals donate the computing power of their computing units when they are inactive, it is possible to execute large distributed applications using unlimited free computing units from all around the world. This concept is called "cycle stealing". Although it seems the perfect solution for the lack of computing resources, many problems have to be considered when using this architecture. The computing units in this architecture are very heterogeneous: a computing unit could be a PDA with a small processor (around 400Mhz) and limited memory or it could be a server workstation equipped with many powerful multicores processors and a huge memory. Since, they are public computing units, they can disconnect from this architecture at any moment, in particular when executing a task. For this reason, the global/volunteer computing architecture is considered as a highly volatile environment and some sort of fault tolerance policy must be adopted when it is used. Moreover, since the nodes forming this architecture communicate through the Internet, problems of security and reliability are raised. Finally, this distributed architecture suffers from the high latency of the communications via the Internet.

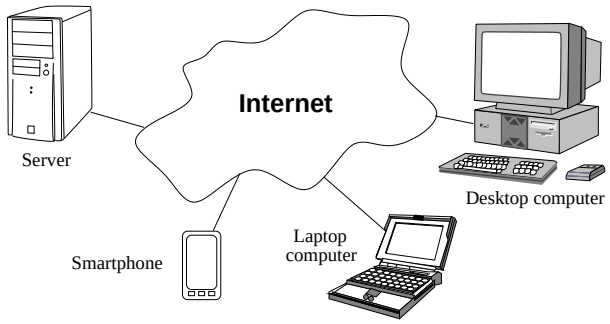

Figure 1.3: The global computing architecture

As described above, all the parallel architectures are composed of many computing units that are all connected together by some type of network. The complexity of the hardware forming these parallel architectures must be hidden from the users who would like to smoothly use the large number of resources as if they were executing an application over one computing unit. Therefore, many middlewares have been developed in order to fulfill this objective. They all present high-level APIs (Applica-

tion Programming Interface) that allow users to program and execute parallel applications over the distributed architectures without any direct intervention on the used resources. In the next section, we present some of these environments.

1.4 Environments

Many environments have been developed in order to use the computing power of the distributed architectures that were presented in the previous section. The conception of these environments are directly dependent on the architecture they run on. Therefore, in this section the environments are classified into three categories and according to the kind of distributed architecture they are designed for.

1.4.1 Middlewares for supercomputers and local clusters

Since supercomputers and local clusters are usually composed of homogeneous computing units interconnected via a fast network, the same middlewares can be used for manipulating both architectures. Many middlewares have been developed to control and run applications over these architectures. Here are the most common interfaces:

- **PVM** [40] for Parallel Virtual Machine, is a software package that allows heterogeneous computers interconnected via a network to be used as a single large parallel computing unit. Thus, it can solve large computational problems more cost effectively by using the aggregate computing power and memory of many computers. The PVM system is composed of two parts: the first part is a daemon that is executed on all the computers participating in the system. These daemons make up the virtual machine. The second part of the system is a library of PVM interface routines. It contains primitives that are needed for cooperation between the tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine. An application is decomposed into tasks which are executed on the daemons. If the tasks are interdependent, they can synchronize with each others or exchange data. The communications are based on the UDP and TCP protocols which are known for their portability. The PVM system currently supports C, C++ and Fortran languages for programming parallel applications. PVM is a simple environment but it lacks a fault tolerance policy which makes it vulnerable for crashes. Moreover, it does not support asynchronous non-blocking sending which is required in some parallel models to reduce synchronizations in high latency architectures.

- **MPI** [58] for Message Passing Interface, is a specification for an application programming interface (API) used to program parallel computers and based on

the message passing concept. The MPI library functions include, but are not limited to, send/receive operations, combining partial results of computations (gathering and reduction operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session. This specification has been implemented in many programming languages like C, Fortran and Java (for example, LAM/MPI [21], MPICH [46] and MPJ Express [18]). Nowadays, it is used to execute parallel applications over local and distributed clusters (if the same implementation of MPI exists on all the computing units). This specification does not include a fault tolerance policy. If a node crashes while executing a parallel application with MPI, the whole application is terminated. For this reason, researchers at the LRI laboratory (Paris XI) has developed MPICH-V [20] which is a fault tolerant implementation of MPI based on MPICH.

- **OpenMP** [30] for Open Multi-Processing, is an API for executing C, C++ or Fortran parallel applications on distributed architectures. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP is an implementation of the multithreading concept, a method of parallelization whereby the master thread forks a specified number of slave threads. The application is then divided among the slave threads. Every thread is allocated to a computing unit and they execute their tasks in parallel. The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will trigger the creation of the slave threads before the section is executed. Every slave thread executes the parallelized section of code independently. Once they finish their tasks, the slave threads join back into the master thread which continues the application. The runtime environment allocates slave threads to processors depending on usage, machine load and other factors. The number of threads to execute parallelized code parts can be assigned by the runtime environment based on environment variables or in the application's code using dedicated functions.
  This API facilitates parallelizing a serial application. It only requires the addition of preprocessor directives before the part of codes that need to be parallelized. However, it does not provide communications' primitives and thus it executes applications with dependent tasks only on parallel architectures with shared memory. For architectures with distributed memories, only independent code parts can be parallelized. On the other hand a new version of OpenMP, called cluster OpenMP, have been developed. It enables users to execute OpenMP's applications with dependencies between nodes in the parallelized sections over local cluster architecture. It simulates a shared memory between the different computing nodes in order to facilitate the development of parallel applications by the user. In practice, this API uses a message passing model but it is com-

pletely transparent to the user. For more information concerning this API, the reader can refer to [57]. Finally, both APIs do not offer any fault tolerance policy to resist to failures. Therefore, they are not adapted for volatile environments.

1.4.2 Middlewares for distributed clusters

Since distributed clusters are composed of heterogeneous distant clusters, the middlewares developed for running parallel applications on them must be capable to adjust to the heterogeneity of the computing units and to the high latency communications. Moreover, some fault tolerance policies must be implemented in these environments to resist to the volatility of computing units. Here are some examples of these platforms:

- **Globus** [37]: it is an open source Toolkit for grids. It allows people to securely share computing power, databases and other tools online. This project was developed in collaboration between the university of Chicago and the Information Sciences institute of South California and funded by DARPA. Its first version was created in 1997 and now the fourth version is available. The Globus toolkit includes software services and libraries for resource monitoring, discovery, and management, plus security, fault detection, communication and file management. It is packaged as a set of components that can be used either independently or together to develop parallel applications. Since resources such as data archives, computers, and networks are usually heterogeneous and thus incompatible, Globus allows transparent access to these resources which are referred to as virtual organizations, and removes obstacles that prevent seamless collaboration between them. The last version of Globus offers new tools and libraries for developing client-server applications according to the WSRF (Web Services Resource Framework) conventions. Due to its general functionalities and the various services it offers, this multipurpose toolkit is complicated to be used for specific computing tasks.

- **Legion** [27]: Legion is an object-based, meta-systems software project at the University of Virginia. It can manage huge amounts of distributed resources which are represented as a single worldwide virtual computer in order to hide the complexity of hardware and software systems to users. They can access from a single machine to all kinds of data and physical resources, such as digital libraries, physical simulations, cameras, linear accelerators, and video streams. Groups of users can construct shared virtual work spaces, to collaborate research and exchange information. This abstraction springs from Legion's transparent scheduling, data management, fault tolerance, site autonomy, and a wide range of security options.
  All the hardware and software resources in Legion are represented by Legion objects, which are active processes that respond to member function invocations

from other objects in the system. Every Legion object is defined and managed by its class object: class objects create new instances, schedule them for execution, activate and deactivate them, and provide information about their current location to client objects that wish to communicate with them. Legion contains default implementations of several useful types of classes but users can define and build their own class objects, especially if they do not meet the users' performance, security, or functionality requirements. Moreover, since Legion supports interoperability between objects written in multiple languages, users can run applications written in multiple languages. Finally, Legion is an open system that encourages third party development of new or updated applications, run-time library implementations, and core components. There is also a commercial product based on Legion, called AVAKI [45]. Since Legion is not dedicated to high performance computing, it implements only general multi-purpose functionalities and it concentrates on hiding the complexity of software and hardware to users, it is not very efficient in solving large numerical problems.

- **ProActive** [9]: ProActive is an Open Source Java library for parallel, distributed, and multi-threaded computing. With a reduced set of simple primitives, ProActive provides a comprehensive toolkit that simplifies the programming of parallel applications that are executed over distributed architectures like clusters, Internet grids and peer-to-peer intranets. It also provides a graphical environment for remote monitoring and steering of distributed and grid applications. This environment allows visualizing graphically the hosts, Java virtual machines, and active objects, including the topology and the volume of communications. ProActive includes many more services that makes parallel programming an easy task: timers chart views, high-level frameworks and C/MPI code integration. Furthermore, a lot of features like fault tolerance and load balancing can be added to the applications. Therefore, ProActive is extensible, making the toolkit open for adaptations and optimizations.
  Although this environment provides direct communications between nodes using the RMI technology (Remote Method Invocation), when two nodes communicate, they must be synchronized (even if the concept of future objects exists). In consequence, the asynchronous iteration model (see next chapter) cannot be used on this platform. Furthermore, to resist to nodes failures, ProActive uses a global checkpointing mechanism, that requires synchronizing all the nodes, or message logging. The two methods are centralized and are not well suited for large scale volatile environments.

- **Padico** [31]: Padico is a software environment, designed for high performance parallel and distributed computing. It targets applications based on the concept of parallel CORBA [1] objects. The main feature of this environment is that it provides the user with fast CORBA, MPI and Java RMI implementations (available

at the same time) over high performance networks such as Myrinet, Infiniband or Quadrics QsNet. PadicoTM, for Padico's Task Manager, offers a framework that allows different communication middlewares to efficiently cohabit within the same process. This framework has a modular design and is composed of core modules like thread management and network multiplexing, and service modules like middlewares and runtimes. The main benefit of this platform is that it eliminates conflict between different middlewares using the same high speed network. Although this platform adds a new layer between the parallel distributed applications and the network resources, experiments show that the additional overhead is insignificant. On the other hand, this platform is not fault tolerant and thus it is not well adapted for volatile environments. Moreover, in our work we did not study the use of two middlewares at the same time. But it would be very interesting to be able to couple two codes that use two different middlewares without having to modify them.

1.4.3 Middlewares for global/volunteer computing

The global/volunteer computing architecture is composed of public computing units distributed all around the world. It uses the cycle stealing concept to benefit from public machines that are connected to Internet and that do not use all their computing powers at the time being. Any middleware developed for such architecture have to mainly tackle the following issues: the high volatility and heterogeneity of the computing units and the high latency of communications. Here are some examples of existing platforms for this type of architectures:

- **Seti@home** [7]: it stands for "Search for Extra Terrestrial Intelligence". It is an american project that aims at discovering if there is some life forms in outer-space. It analyzes radio telescope data using Internet-connected computers. It has a client/server architecture: the server divides the signal captured by the radio telescope into small portions then sends them to the clients via Internet. The clients analyze these small portions and return the result to the server. Each portion is analyzed by many clients in order to verify the authenticity of the results and overcome a client failure. The server compares the results and detects if two clients have computed different results for the same task. In this case, it suspects that the results have been modified by a malicious client. Therefore, it executes the same task over some new clients. The amazing success this platform have achieved, helped the creation of similar platforms like Folding@home [54] (simulating the folding of proteins to discover their functionalities and the effect of misfoldings on the human body) and Genome@home [54] (designing new genes to generate new protein sequences that do not exist naturally in order to understand how natural genomes have evolved and how natural genes and proteins work). Condor [56] and Boinc [6] (Berkeley Open Infrastructure for Network

Computing) can be seen as a generalization of Seti@home. They are independent of the problems they are solving. They also offer some fault tolerance policies and can now compute tasks faster by using the clients' GPUs in addition to CPUs. However, in all these platforms, the clients cannot communicate with each other. So these platforms cannot execute a parallel computing application with dependencies between tasks.

- **XtremWeb** [35]: it is a french project developed at the LRI laboratory. XtremWeb provides general services for global computing. Resources participating in this environment can have two modes:

  – In user mode, a complete environment is dedicated to designing and programming parallel applications. Then, the final application can be submitted to the system for execution.

  – In volunteer mode, the computing resource is used by the system for executing submitted applications when it is not used by its owner.

  XtremWeb is composed of three entities: clients, workers and coordinators. Most of the time they interact as follows: the client submits its job to the coordinator and the coordinator send the resulting tasks to the workers who execute them. Clients and workers never communicate directly. All the communications pass by the coordinator. Therefore, the coordinator has a very important role in the architecture of XtremWeb and it has many duties:

  – It handles the clients' requests and distributes the tasks to free workers.

  – It monitors the execution of tasks over workers.

  – It detects the eventual crashes of workers and restarts the corresponding tasks on other free workers.

  – It retrieves and saves the tasks' results when workers finish them. Then it forward them to the client.

  The coordinator can be duplicated on many resources and the duties previously described can be equally distributed on the coordinators in order to reduce the load of the coordinator and the bottlenecks.

  All the communications use the RPC (Remote Procedure Call) concept. In particular, they use the RMI Java and the XML-RPC implementations. Moreover, the communications in XtremWeb can bypass security measures and firewalls, in particular when transferring the application's code to a new computing unit. However, the workers are not able to communicate with each others and thus computing applications with dependencies between tasks is impossible.

- **XtremWeb-CH** [4] It is an improved version of the XtremWeb environment. This new version is more adapted to peer-to-peer architectures. It can execute parallel and distributed applications with dependencies between tasks. Indeed the parallel application is decomposed into modules that can communicate with each others by exchanging files: for example a module may require the output of another module before beginning the execution of its task. Therefore, the modules can communicate indirectly in the XWCH-sMs platform (sMs: slave-master-slave) where the output of one module is sent to the master and then the master transfers it to the module waiting for that output. Or directly in the XWCH-p2p platform where the module signals to the master that it has finished its task, then the signal is passed to the worker waiting for the output of that task and finally it requests the output file from the first worker. The data flow graph of the parallel distributed application is represented in an XML file which should be created by the client before launching its application. This file explicitly describes the input and output file of each module, and their interdependence. This platform is not well suited for parallel iterative algorithms because interdependent tasks cannot be executed in parallel as it is required in such algorithms. To overcome this problem the developer must create a module for each iteration in each task and a module for computing the global residual value (convergence-detection) at the end of each iteration. After executing an iteration for all the tasks by the computing modules, the convergence-detection module must compute the global residual value using the output of all the computing modules. If the the global residual value is higher than the requested precision, the convergence-detection module must dynamically generate new modules that will execute the next iteration for all the tasks using the output of the previous ones. This implementation model is not trivial. Moreover, since there is no recovery mechanism implemented in XWCH, it is not well adapted for highly volatile environment. A dead task must be computed all over again at each crash.

- **JXTA** [43]: it is an open-source project introduced by Sun Microsystems Inc. It is composed of a set of peer-to-peer protocols which are defined as a set of XML messages that allow any device (cell phone to PDA, PC to server) connected to a network to exchange messages and to collaborate independently of the underlying network topology. JXTA peers create a virtual network where any peer can directly interact with other peers and resources.
  JXTA is composed of the following primary components:

  - Peers are any networked device that implements one or more of the JXTA protocol.
  - Peer group is a collection of peers that have agreed upon a common set of services.

  - Network services are offered by peers and group of peers.
  - Modules are an abstraction used to represent any piece of code used to implement a behavior in the JXTA world.
  - Pipes are used by peers to exchange messages. Pipes are an asynchronous and unidirectional non reliable message transfer mechanism.
  - Advertisements can represent any resource. They are XML documents that describe and publish the existence of peer resources.

  These components use the following basic protocols to interact:

  - Peer Discovery Protocol (PDP) is used by peers to advertise their own resources and discover resources from other peers.
  - Peer Information Protocol (PIP) provides a set of messages to obtain peer status information (uptime, state, etc.).
  - Peer Resolver Protocol (PRP) enables peers to send generic query requests to other peers . PDP and PIP listed above are built using the PRP.
  - Pipe Binding Protocol is used by peers to establish communication channel or pipe between one or more peers.
  - Endpoint Routing Protocol defines a set of request/query messages that are used to find routing information.
  - Rendezvous Protocol is a mechanism by which peers can subscribe or be a subscriber to a propagation service. It is used by the PRP and PBP in order to propagate messages.

  Using these protocols developers are able to create any kind of peer-to-peer application based on the Java technology (e.g. file sharing, messenger, distributed computing...). JXTA has a strong supportive community and a lot of interesting peer-to-peer protocols have been implemented using this technology. However, JXTA is a low level platform, so much of the task management and the support for different computing models is left to the developer of the application to implement. As a consequence, JXTA is rather too complicated to be used and offers a lot of general functionalities that are not well adapted for executing complex computing applications.

- **P2P-MPI** [42] for Peer-to-Peer Message Passing Interface, is a middleware framework that runs parallel applications over peer-to-peer architectures. P2P-MPI is developed in Java and thus it can be used across all platforms. This message passing library is similar to MPJ-Express. However, it is fault tolerant: it implements a transparent process replication mechanism that allows nodes to resist to an eventual crash. Each group of replicas is composed of one master and many

slaves. When a master crashes, its slaves detect this failure and elect a new master from the slaves to replace the dead one. The "Binary Round Robbin Gossip" style is used for detecting failures and the computing units communicate through Java's TCP sockets. P2P-MPI consists of four processes:

- The File Transfer Service (FT) is in charge of transferring the executable code and inputs from the submitter (the node requesting the execution of the parallel program) to the computing nodes when they need it.
- The Message Passing Daemon (MPD) is executed on each computing unit participating in the P2P-MPI environment. It allows peers to communicate with each others.
- The Fault Detection Service (FD) monitors the resources executing an application and detects failures.
- The Reservation Service (RS) reserves the required resources to compute a given parallel application.

Since when receiving a data message from a neighbor, the master have to broadcast it onto its slaves, we fear that the network will be overloaded with redundant messages, especially if the nodes are communicating via a high latency network. Moreover, since the number of replica are defined when launching the application and a dead replica cannot be replaced, we feel that this restriction in number of crashes makes P2P-MPI not well adapted for highly volatile environments.

Even if some of these middlewares are not meant to be used over parallel architectures with volatile nodes, they must all have some type of fault tolerance policy. Indeed, it is very probable that some type of crash occurs while executing a lengthy parallel application over many computing nodes. In this case, the middleware, supervising the execution of this parallel application, must be able to overcome this failure and to continue the execution of the application. In the next section, we present the most common fault tolerance policies while discussing their advantages and drawbacks.

1.5 Fault Tolerance

Many distributed architectures are composed of volatile computing units. To be able to execute large and lengthy applications over these architectures, a fault tolerance policy must be implemented in those parallel applications. These policies are usually composed of a fault detection mechanism and a restoring mechanism.

1.5.1 Fault detection mechanisms

Fault detection is an essential mechanism for fault tolerance. Its aim is to detect if a computing unit participating in the execution of the application has crashed. An efficient fault detection mechanism must quickly discover the dead nodes, in order to trigger as fast as possible the restoring mechanism and to reduce the effect of this crash on the performance of the application. Moreover, it must not overload the network interconnecting the computing nodes with status messages. Finally, it must not significantly reduce or affect the performance of the computing process.
This mechanism have been implemented in many ways. Here are some examples:

- **Standard fault detection mechanism**: it uses the Push or the Pull model [36]. In the Push model, each computing unit that is executing the parallel application, have to regularly send heartbeat messages to a dedicated server. This server monitors the state of the computing nodes. If it does not receive a heartbeat message from a computing unit for a defined period of time, it suspects that this computing node is dead and triggers the restoring mechanism. If the architecture used suffers from high latency in communications between nodes, the heartbeat messages will be delayed which increases the probability of detecting a false crash by the server. On the other hand, in the Pull model, the server regularly sends liveness requests to the computing nodes. The computing nodes answer back and confirm that they are alive. If a computing node does not answer to the server's request after a predefined time period, the latter considers it as dead and triggers the restoring mechanism. The Pull model requires double the number of messages used by the Push model because each heartbeat message must be answered back but it is more precise than the Push model in detecting crashes. Therefore a third model, resulting from the combination of the two previous models, have been created. It is called the Dual model and has the advantages of both previous models. The Dual model consists of two phases: in the first one, the Push model is applied. When the server suspects that a computing node is dead because it has not been sending heartbeat messages for a while, it triggers the second phase where the pull model is used. Indeed, the server sends a liveness requests to the suspected computing node. If it does not respond, the server considers it as dead and triggers the restoring mechanism. On the other hand, if it responds to the request then the Push model is applied again. Figure 1.4 illustrates the Dual model where a computing node sends every T1 time period a heartbeat message to the server. If after $2 * T1$ time period the server does not receive a heartbeat message, it sends a liveness request to the suspected dead node. If after $2 * T1$ time period it does not receive an answer, it activates the restoring mechanism.

  This centralized mechanism is not scalable because the server could be overloaded with heartbeat messages sent from numerous computing nodes. Moreover, this mechanism creates a new critical point which is the server itself. In-

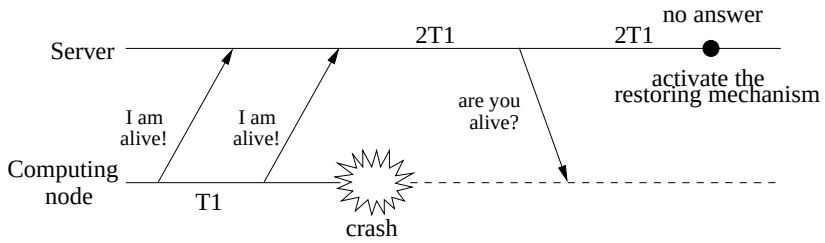

Figure 1.4: The Dual fault detection model

deed, if the server crashes the whole fault tolerance policy becomes invalid. Furthermore, if many nodes are participating in executing a parallel application, the server has to monitor the state of all these computing units. Thus a large amount of time is required to test the state of all the nodes which increases the detection time of a dead node and reduces the performances of this mechanism. Therefore, this basic mechanism is not well suited for large scale applications. To tackle the drawbacks of this mechanism, some variants have been proposed where the server is duplicated many times and each replica monitors a subgroup of the computing units and the rest of the replicas. Figure 1.5 illustrates the centralized heartbeating model with multiple servers. The servers are represented by rectangles and the computing nodes by circles. The computing nodes are divided into two groups. Each one is monitored by a server. For this reason, each server receives heartbeat messages (represented by arrows) from its group of nodes. Furthermore, we can notice that the servers also exchange heartbeat messages. This allows the servers to detect if one of them has crashed. In this case a restoring mechanism for servers must be implemented to handle the crash of servers.

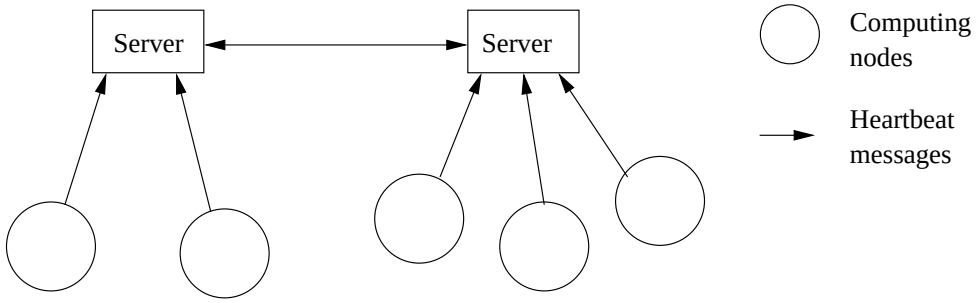

Figure 1.5: The centralized heartbeating model with multiple servers

- **Gossip protocol** [61]: using this protocol, each node maintains a table that contains for each known member its address and an integer which is the heartbeat counter. Every T_{gossip}, each computing node increments its heartbeat counter and sends its table to a randomly selected node. Upon receiving such gossip message, a node merges the table in the message with its own and adopts the maximum heartbeat counter for each member. If a node A detects that since $T_{cleanup}$ the

heartbeat counter of node B has not increased, then node A suspects that B is dead. Afterward a consensus phase is required to be sure that B is dead. This protocol is completely decentralized and so it is fully scalable and well suited for large scale volatile architecture. In practice, random gossip evens the communication load amongst the network links but has the disadvantage of being non-deterministic. It is possible that a node receives no gossip message for a period long enough to cause a false failure detection. To minimize this risk, the system's developer can increase $T_{cleanup}$ at the cost of a longer detection time. In order to make gossip messages traffic more uniform, at every round, each node sends its table to a neighbor while using the "Round Robin" strategy. The rank of the destination is computed as follows:

$$d = (r + s)\ mod\ n,\ 0 \leq s < n,\ 1 \leq r < n$$

where r is the current round, s the source's rank and n the number of nodes in the architecture. This protocol guarantees that all nodes will receive the updated heartbeat for every node within a bounded period of time. Thus, it is possible to deduce the minimum $T_{cleanup}$ required to reduce the probability of false failure detections. Another variant of this protocol uses the Binary Round-Robin strategy. It aims to minimize the bandwidth used for gossiping by eliminating redundant gossip messages. This elimination alleviates the network's load and accelerates the propagation of updates for nodes' heartbeat counters at the cost of increasing the risk of false failure detections. For more information on this or other variants of this protocol (like the double Binary Round-Robin gossip protocol) the reader can refer to [41].

1.5.2 Restoring mechanisms

Once a failure is detected, it is capital to trigger some sort of reaction to overcome this crash. These reactions are known as the restoring mechanism. However, to be able to resist to a failure, the system cannot jut react after the occurrence of a crash. It has to be well prepared for this problem. Thus it has to regularly execute some sort of scheme that makes the system fault tolerant. The actions undertaken after a crash are directly dependent of these schemes. Therefore, we consider that these schemes form an essential part of the restoring mechanism. In the listing below we present the most common restoring mechanisms and we show their advantages and drawbacks.

- **Checkpointing** [60]: this is the most common scheme. It regularly saves for each node the essential data required to continue the application after a crash. This procedure is executed on precise checkpoints which are usually specified by the application's developer. Nowadays a good fault tolerant platform automatically

detects the checkpoints without any intervention from the user, we say it provides transparent fault tolerance services. Checkpointing can be divided into two groups:

1. **Coordinated checkpointing** [26]: when using this method, at each checkpoint, all the computing nodes in the system must be synchronized with a barrier call and a snapshot of all the system is taken and saved. If a node crashes, all the other nodes, participating in the execution of the application, are blocked until the dead node is replaced. Once the replacement node retrieves the last backup to continue the task, all the nodes must also rollback to the last checkpoint and continue their tasks from that checkpoint.
2. **Uncoordinated checkpointing**: using this method, after a checkpoint, each node saves its data without any synchronization with the others. This allows the nodes to have backups at various times. When a node crashes and is replaced by a new one, the rest of the nodes, participating in the execution of the application, do not have to rollback to the last checkpoint and continue their tasks from that checkpoint.

Most of the fault tolerant platforms that use the checkpointing concept save their backups on a server. This requires to have at least one reliable station. This centralization may cause a bottleneck and limit the scalability of the system. To reduce the effect of this problem, stations with high computing capacities and large bandwidth are used as backup servers. This method is not well adapted to distributed clusters and peer-to-peer environments where all the nodes are volatile and have very limited capacities. In such environments, it is preferred to use the distributed redundant checkpointing method where the backups are saved on the computing nodes: each node saves its backups on its neighbors, so the number of backups per node is equal to the number of backup neighbors (which is specified by the user). This method does not make the platform fully tolerant to all types of crashes. Indeed, if a node and all its saving neighbors die at the same time, the backups for this node are lost and the platform cannot replace the dead node which will terminate the application. To make the platform more resistant to crashes, the user can increase the number of backup nodes, but this could reduce the performances of the platform. On the other hand, this scheme is very scalable because it is completely decentralized.

Finally, the backups saved either on a server or on computing nodes can be stored in the memory or in files (XML files, binaries ...) on hard disks. If the size of the backups is too big, it is preferable to store them in files to prevent overloading the memory. However, this method slows down the platform because it has to access the hard disk every time a node demands a backup from its neighbors.

- **Process Replication** [34]: using this scheme, each computing process is executed

on many nodes. A master is chosen to represent each computing process and the others are considered as slaves. All the replications of a computing process execute the same process at the same time, but only the master communicates with the other masters. After receiving a message, it broadcasts it to its slaves which enables them to continue their tasks. If a master crashes, it is replaced by a slave that continue the task and becomes the new master for this computing process. The dead node cannot be replaced by a new one because there is no backup to retrieve. So after many crashes a computing process may be terminated if all its replications are dead. To increase the robustness of this scheme the number of replicas per computing process must be increased. On the other hand, this reduces the performance of the masters because they have to broadcast to a larger number of slaves each message received. Moreover, this scheme is not very practical because if an application requires m computing processes, the platform has to reserve $n * m$ computing units (where n is the number of replicas per computing process). This method has been implemented in the P2P-MPI platform.

- **Message logging** [5]: using this scheme, each node saves the data that it receives into a log. When a node crashes, it is replaced by a new one which retrieves the log and executes from the beginning all the operations using the data saved in the log. This method saves the logs on a reliable server and requires to restart the computing process of the dead node from the beginning. If the application is large, the log may become full. For this reason, most of the time, message logging is used in conjunction with the uncoordinated checkpointing method: each node saves in a local log the messages it sent to its neighbors. When the log becomes too big, the node's data are saved on a reliable server. If a node crashes, the replacement node has to retrieve the backup from the backup server and the logs from its neighbors, then it executes all the operations that happened after the backup using the data stored on the logs. If a neighbor's log does not contain the required data, the neighbor must retrieve its backup and rollback [33] to that state then it has to execute again all the operations executed after the backup. So, one crash may cause a domino effect and oblige many nodes to retrieve their old backups and rollback to that state. This method guarantees that the system will tolerate all the crashes that affect the computing nodes but it takes a lot of time to replace a dead node. This method is not adapted to volatile environments because it requires a reliable backup server which is impossible to guarantee in such environments. Since, it saves its backups or logs on a single server, this centralization limits the scalability of this scheme. This scheme has been implemented in the MPICH-V2 platform.

1.6 Conclusion

In this chapter we have presented in general the purpose of simulating numerically natural problems which can be solved by direct or iterative methods. However, large and complex problems cannot be solved by a sequential numerical method executed on a single computing unit. Supercomputers or distributed architectures are required to solve these large problems. Due to the expensive price of supercomputers, many researchers are using distributed architectures which can be classified into three categories: local clusters, distributed clusters and global/volunteer computing architectures. Many middlewares have been developed in order to allow users to easily execute parallel applications over distributed architectures. These middlewares hide the hardware complexity of the distributed architectures and create a virtual machine that is more easy to use. Some middlewares adapted for each category of parallel architectures have been briefly presented in the fourth section of this chapter. The main issues encountered when using parallel architectures are the heterogeneity of the computing units, the high latency of interconnecting networks and the most important one, the volatility of the computing nodes. To tackle the problem of nodes' volatility, a fault tolerance policy must be implemented into the middleware managing the resources. These policies are usually composed of a fault detection mechanism and a restoring mechanism. We have presented some schemes for implementing these two mechanisms and emphasized their advantages and disadvantages.

In our research, we are interested in solving large and complex numerical problems which require huge amounts of computing power. Therefore, we prefer to use the distributed clusters and the global computing architectures. These parallel architectures are not crash free which imposes implementing a fault tolerant policy in the middleware that is managing the resources. It must provide a highly scalable fault detection mechanism because we compute large applications using a lot of computing units. Thus, the implementation of this mechanism must be decentralized and it has to detect failures as fast as possible (independently of the number of computing nodes) and without overloading the network with heartbeat messages. It is also crucial that while using this mechanism the probability of false failure detection is as low as possible. Finally, it is preferred that it does not impose many constraints for its implementation (like to be able to communicate with all the nodes or it requires some information on the topology of the architecture because this mechanism could be used on peer-to-peer architectures where such informations are not available). The fault tolerance policy must also provide a restoring mechanism which is highly scalable. Thus, it has to be decentralized and must not reduce significantly the performances of the system. Therefore, it has to be a light process that is executed on each computing unit in parallel with the computing process and that does not consume a lot of computing power nor a large bandwidth. Finally, the restoring mechanism must recover

the system from a failure as fast as possible without affecting other computing nodes.

In the next chapter, we present the different models for parallelizing sequential iterative numerical methods. In particular, we introduce the asynchronous iteration model. Then we describe two environments for executing asynchronous parallel iterative programs and we explain the various issues that these platforms have to overcome.

Chapter 2

The Asynchronous Iteration Model

2.1 Description

Large and complex problems cannot be solved by sequential resolution methods that use just one computing unit, because it usually does not have sufficient computing power nor enough memory. Therefore, developers decompose the large problems into smaller ones and most of the time, they try to parallelize the resolution methods in order to solve the small problems in parallel on many computing units. This concept is called "SPMD" for Single Process on Multiple Data. As mentioned in the previous chapter, distributed architectures, composed of many computing units, can be divided to three classes depending on their resources' interconnection and localization. They all offer huge amounts of computing powers that can be harnessed to quickly solve large and complex problems. In our work, we are interested in the parallelization of iterative methods because they are very popular and the resulting parallel iterative methods are more efficient than direct ones.

Let F be a linear or nonlinear mapping from E to E, whose domain of definition is $D(F)$,

$$F : D(F) \subset E \rightarrow E.$$

Consider a sequential iterative algorithm associated to F, i.e. a sequential algorithm defined by

$$X^{k+1} = F(X^k) \qquad \textit{Given an arbitrary } X^0 \in D(F)$$

and where $X = (X_1, X_2, ..., X_n)$ is the vector of unknowns. If the sets of X^k generated in algorithm 2.1 converge and F is a continuous function then $X^* = F(X^*)$ is verified and x^* is a fixed point of F.

To parallelize this iterative algorithm and solve this problem on m computing units, the components of vector X must be decomposed into m groups, $X^k = (X_1^k, ..., X_m^k)$. Each group of components is solved on one computing unit as follows:

$$X_i^{k+1} = F_i(X_1^k, ..., X_m^k) \qquad \textit{where} \quad i = 1, ..., m \tag{2.1}$$

Algorithm 2.1 the iterative sequential algorithmic model

Given an arbitrary X^0
for $k = 1$ step 1 until convergence **do**
  $X^{k+1} = F(X^k)$
end for

X_i^k is the group of components evaluated on the i^{th} computing unit at iteration k. The algorithmic model will be modified as presented in algorithm 2.2.

Algorithm 2.2 the parallel iterative algorithmic model

Given an arbitrary $(X_1^0, ..., X_m^0)$
for $k = 1$ step 1 until convergence **do**
  for $i = 1, ..., m$ **do**
    $X_i^{k+1} = F(X_i^k)$
  end for
end for

According to F_i some group of components may be dependent of each others. Thus, they must exchange data at each iteration. According to the type of communications, synchronous or asynchronous, used to exchange data, three models for parallelizing iterative sequential algorithms can be identified: the SISC, SIAC and AIAC models [11].

In the rest of this chapter we present these three models for parallelizing iterative sequential algorithms and we discuss their advantages and drawbacks. In particular, we explain the benefits of AIAC algorithms in heterogeneous and volatile environments. In the second and third sections, we present JACE and JACEP2P-V1 which are two environments dedicated for executing AIAC algorithms. Their architectures and characteristics are explained and their weaknesses are also pointed out in order to expose the problematic. We end this chapter with a brief summary and a conclusion.

2.2 The SISC model

SISC stands for Synchronous Iterations and Synchronous Communications. It is the standard model for parallelizing iterative sequential algorithms with dependencies between tasks. Once the components of a problem are divided between the computing nodes, as illustrated in Figure 2.1, each computing unit executes an iteration (represented by a dark rectangle) and synchronously sends its dependency values (represented by arrows) to the nodes that require them. Afterward, it waits the reception of all the data messages that the other computing units sent to it and then executes the next iteration while using the newly acquired data in its computation.

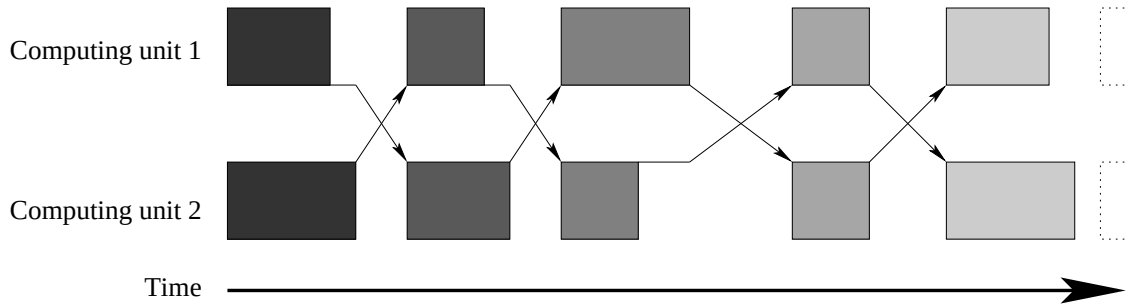

Figure 2.1: The Synchronous Iteration and Synchronous Communication model

These synchronizations may generate large periods of idle times (represented by white spaces in Figure 2.1) between two successive iterations. Indeed, if a fast computing unit quickly executes an iteration, it must wait for the other slow ones to finish their iterations to be able to synchronously send its dependency data to them. It then waits again the reception of data messages from its neighbors. This procedure wastes huge amounts of computing power. The length of idle periods is increased when the computing nodes are very heterogeneous, in particular when the load, given to each computing node, is not adapted to its computing capacities. Moreover, if a data message, intended for a computing unit, is lost due to a network's congestion or failure, the computing node will keep on waiting for ever for this message and the whole application will be blocked. In the same way, if a computing node crashes, all its neighbors will be blocked while waiting for data messages from the dead node. Therefore, this parallel iterative model do not seem to be well adapted for volatile environments if no checkpointing mechanism, which could be very penalizing, is implemented.

2.3 The SIAC model

SIAC stands for Synchronous Iterations and Asynchronous Communications. It is similar to the SISC model but here the data messages are exchanged asynchronously. As illustrated in Figure 2.2, after computing an iteration, a computing node asynchronously sends its dependency values to its neighbors, without synchronizing with them and without waiting for them to finish their iterations. Moreover, it does not wait for its neighbors to receive the data messages that it has sent to them but it waits that it receives data messages from all it neighbors. In this way the communications are partially overlapped by the computing process and therefore the idle times between iterations are reduced but not entirely eliminated. Indeed, fast computing nodes still have to wait for slow ones to send their data messages.

As the SISC model, the SIAC is not well suited for volatile environments because it does not tolerates the loss of data messages nor the failure of a computing unit. In case a node crashes, all the computing nodes are blocked until some restoring mechanism

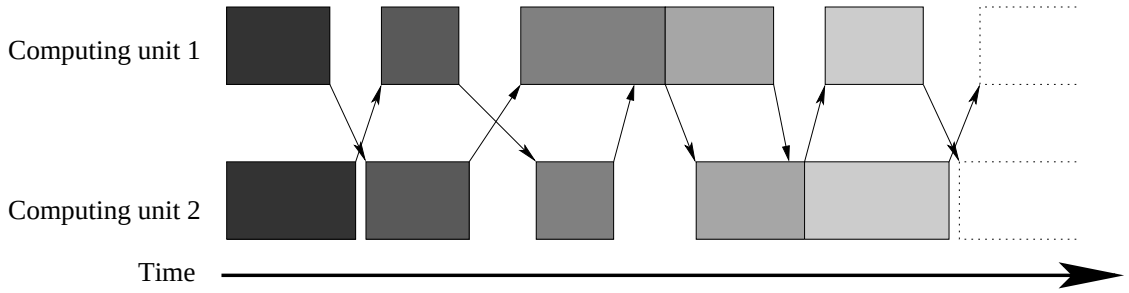

Figure 2.2: The Synchronous Iteration and Asynchronous Communication model

is executed. On the other hand, since the communications are asynchronous, the effect of the high latency of the network on the application's performance are reduced.

2.4 The AIAC model

AIAC stands for Asynchronous Iterations and Asynchronous Communications and it is also called the asynchronous iteration model [11, 10]. This model is illustrated in Figure 2.3 where it is clear that there is no idle time at all between iterations (no white spaces between the dark rectangles). In fact, in this model and at the end of every iteration, computing processes asynchronously send their data messages to their neighbors and do not wait for the reception of the new data messages that are sent by their neighbors. They compute the next iteration using just the last received data from their neighbors.

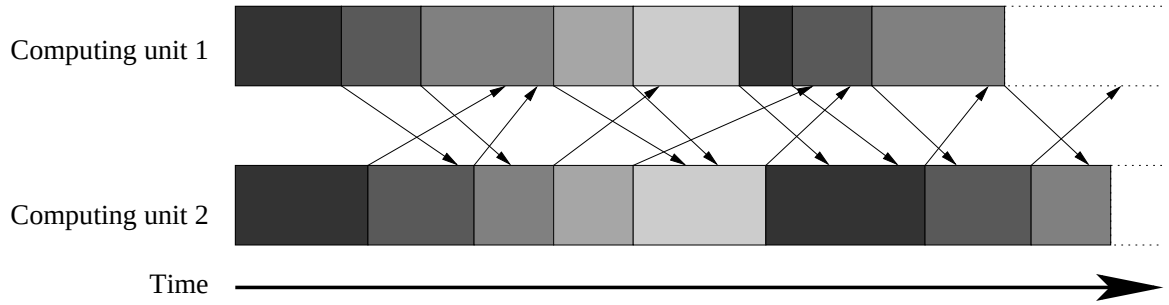

Figure 2.3: The Asynchronous Iteration and Asynchronous Communication model

2.4.1 Algorithmic model

In the asynchronous iteration model, vector X which contains all the components, is partitioned between m computing nodes in the same way as for the synchronous model. However, the main difference between the two models is that the m block-components are not updated at each iteration in the asynchronous iteration model

because the communications and the iterations are asynchronous.

The asynchronous model is based on the following ideas:

- the m block-components may be updated in a random order on the m computing nodes and it is possible that some block-components may not be updated for a while. Nevertheless, no block-component is permanently idle.
- at time t, each node checks if one of its dependencies' components have been updated. If this is the case it updates its own block-components using the last received information from its dependencies. Otherwise, it does nothing at time t.

A computing node can then continue to execute its task without waiting for its neighbor results. As a consequence, nodes may compute different iterations at the same time t.

In the classical model of those algorithms, we denote by $J(t)$ the set of peers updated at time t, also called the strategy of the algorithm, and by $X_j^{s_j^i(t)}$ the state of the group j available for the group i at time t. $s_j^i(t)$ is the state of the data from group j available on group i at time t. It is defined by $s_j^i(t) = t - r_j^i(t) \leq t$, where $r_j^i(t)$ denotes the delay of the group j with respect to the group i at time t.

The classical hypotheses, assumed over the $s_j^i(t)$ in order to ensure that the process actually iterates and then evolves, are the following ones:

Definition Consider the X^k vector decomposed into m block-components and the strategy $J = \{J(t)\}_{t\in\mathbb{N}}$, the sequence of non-empty subsets of the m block-components.

For $i \in \{1, ..., m\}$, let $S^i = \{s_1^i(t), ..., s_m^i(t)\}_{t\in\mathbb{N}}$ be a sequence of $\mathbb{N}^m$, such that

(h1) There exists a positive integer B, used to bound the delays, such as $\forall i, j \in \{1, ..., m\}$ and $\forall t \in \mathbb{N}$, we have: $t - B < s_j^i(t) \leq t$. As $s_j^i(t) = t - r_j^i(t)$ like defined above, we can also deduce the following expression concerning the delays: $0 \leq r_j^i(t) < B$.

(h2) On average no block-component may be neglected by the updating rule. This condition is called "fair sampling condition" and is equivalent to:
$\forall i \in \{1, ..., m\}$, $\text{Card}\left(\{t \in \mathbb{N}, i \in J(t)\}\right) = \infty$.

Then, the asynchronous dynamic of the m-nodes network associated to the given transition function F and activation set J and with initial configuration $X^0 = \left(X_1^0, ..., X_m^0\right)$ is described by Algorithm 2.3 in page 38.

We precise that in this model, t does not necessary correspond to real time, it could simply be an artificial variable used to index the events of interest (e.g., the times at which some variables are updated). We can also remark that a synchronous iterative algorithm is a particular case of asynchronous iterative algorithms in which:

Algorithm 2.3 The asynchronous iteration model

Given an initial state $X^0 = (X_1^0, ..., X_m^0)$
for each time $t = 0, 1, ...$ **do**
  for each block-component $i = 1, ..., m$ **do**
    if $i \in J(t)$ **then**
      $X_i^{t+1} = F_i(X_1^{s_1^i(t)}, ..., X_m^{s_m^i(t)})$
    else
      $X_i^{t+1} = X_i^t$
    end if
  end for
end for

- $r_i^j(t) = 1$ for all the neighbors, whatever the value of t,
- t represents the k variable of the synchronous algorithm given in equation (2.1).

Hypothesis (h1) indicates that the delays are always bounded during the execution and cannot become infinite as t increases. In a practical case, this means that a node should always be able to communicate its results to the other nodes. In order to ensure this hypothesis, the environment handling the parallel asynchronous iterative application must automatically delete from the set of computing nodes each machine that cannot reach other nodes for a predefined period of time.

Concerning hypothesis (h2), it indicates that all components are eventually updated after a finite period of time. This hypothesis must also be ensured by the platform executing the parallel asynchronous iterative application.

The global convergence of the parallel asynchronous iterative method depends on the scientific problem considered and is ensured if certain conditions are satisfied with respect to the data of the problem. For example, some scientific applications are described by systems of differential equations which leads, after discretization using a finite difference scheme, to linear systems that can be solved with a asynchronous method. For more information concerning the convergence conditions of asynchronous iterative algorithms, interested readers can can refer to, for example [65, 19, 38, 39] and the references therein.

In the parallel asynchronous iterative methods, the nodes do not use the Euclidean metric to compute the residual because it requires updated components at each iteration. Usually, the residual is evaluated using the following formula:

$$R = max_i(|X_i^{k+1} - X_i^k|)$$

where X_i^k denotes the value of the component i of array X at iteration k. This norm is also called the max norm.

2.4.2 Advantages

The AIAC model is not easy to implement but it offers many benefits:

- **No idle time.** Since at the end of each iteration, the computing unit does not wait the reception of the data messages from their neighbors, there is no idle times between two successive iterations and the communications are totally overlapped by the computing process.

- **No synchronization.** Since data messages are asynchronously exchanged between neighbors and the computing units do not have to wait for the reception of data messages, the synchronizations between the computing neighbors are totally eliminated. Therefore, iterations are asynchronous and fast computing nodes can execute different iterations from the slow ones at any given moment: for example, in Figure 2.3, the first computing unit executes the seventh iteration and begins the eight while the second computing unit is still computing the sixth iteration. This property makes parallel asynchronous iterative applications much less sensitive to the heterogeneity of communication and computing resources than conventional synchronous parallel iterative algorithms are.

- **Tolerance of data message loss.** The loss of data messages is totally tolerated in the AIAC model because each computing process does not wait for the reception of new data messages. It uses the last received data to compute the next iteration and it is not blocked if a message is lost or delayed. In the same way, if a computing node crashes the neighbors are not affected by this crash. They continue their iterations using the last received data from that dead neighbor. To better illustrate this advantage, in figure 2.4, we present the execution of a parallel iterative synchronous algorithm over a volatile environment. The parallel application is composed of three tasks that are interdependent. To resist to the volatility of the computing nodes, we consider that at the the end of each iteration a global checkpoint is executed and the whole backup image is saved in a safe storage server. Therefore, at the end of an iteration all the nodes are synchronized which results in idle times between iterations as shown in figure 2.4. Moreover, in figure 2.4, processor P1 disconnects while executing the third iteration which blocks the rest of the computing nodes. The application will stay blocked until processor P1 is replaced by a new one which also results in large idle times. Once the dead node is replaced, all the nodes must retrieve the last backup (in this case, it is the backup saved at the end of the second iteration) and continue the tasks from this checkpoint.

  On the other hand, figure 2.5 presents the execution of the asynchronous implementation of the same parallel iterative algorithm illustrated in figure 2.4 and it is also executed on a volatile environment. Since the asynchronous iteration model

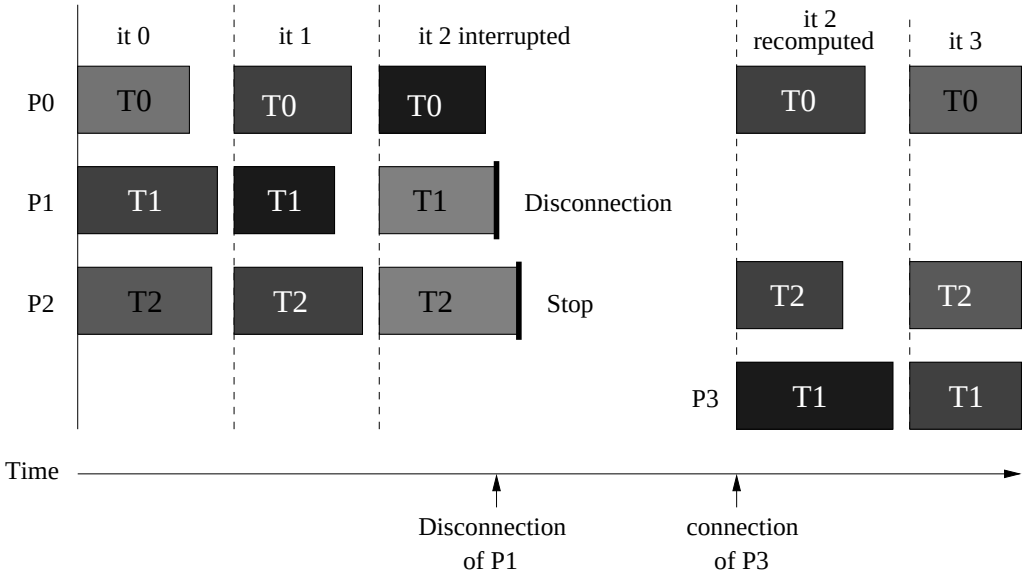

Figure 2.4: The execution of a parallel iterative synchronous algorithm over a volatile environment

is used the uncoordinated checkpointing mechanism is used at the end of each iteration. Therefore, there is no synchronizations between the computing nodes, thus the idle times between successive iterations are eliminated. Furthermore, when a computing node is disconnected while executing a task, for example in figure 2.5 while executing the third iteration of the second task T1, the rest of the nodes continue their tasks and are not affected by this failure. When the dead node is replaced by a new one, only the new node retrieves the last backup and continue its task from this last checkpoint. This comparison shows the huge usefulness of the asynchronous iteration model over volatile environment.

For all these reasons, the AIAC model is well suited for volatile distributed architectures that are composed of heterogeneous computing nodes which are interconnected via a high latency network.

2.4.2.1 Disadvantages

Although the AIAC model might seem the perfect solution to tackle the issues of volatile and heterogeneous architectures, this model also has its own weaknesses and disadvantages:

- **Compatibility.** This model cannot be applied on all types of iterative methods (like GMRES, Gradient Conjugate [3]...) because they will not converge if they are implemented according to the asynchronous iteration model. Indeed, some iterative methods require receiving the data messages from their neighbors at each iteration in order to ensure the convergence of the iterative algorithm in a

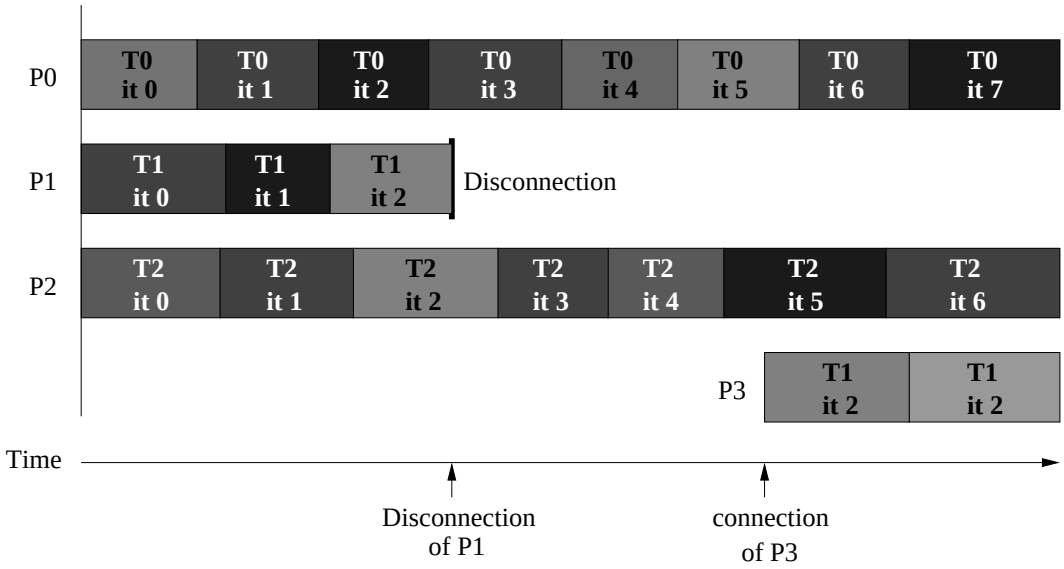

Figure 2.5: The execution of a parallel iterative algorithm based on the asynchronous iteration model over a volatile environment

finite number of iterations. Otherwise, they iterate forever without converging and they could even crash.

- **Execution time.** As shown in [11], when using the asynchronous iteration model, parallel iterative algorithms usually require more iterations than in synchronous models to converge to the solution of the problem. This increase in the number of iterations may lengthen the total execution time of the parallel application, in particular if it is executed over a parallel architecture interconnected via a fast network like a local cluster where the synchronizations are not very penalizing.

- **Convergence detection.** Since in the AIAC model, the tasks do not receive at each iteration the data messages from their neighbors, the standard mechanisms for detecting the global convergence of synchronous parallel iterative algorithms cannot be used for the asynchronous ones. In fact a computing process can execute many iterations without receiving any data messages from its neighbors. In the absence of new data messages, the values of the computing node's components may not vary between successive iterations which could lead it to detect a false local convergence (its local residual is less than the required threshold). This fake convergence is contradicted at the reception of the first data message because the local subsystem will locally diverges after computing the next iteration. Therefore, special mechanisms are required for detecting the global convergence of a parallel iterative algorithm implemented according to the asynchronous iteration model.

In our work, we are interested in solving large linear/non linear systems over distributed clusters and global computing architectures. These numerical problems can

be solved by iterative methods that are compatible with the AIAC model. Moreover, it has been proven in [10] that parallel iterative algorithms that are implemented according to the AIAC model, solve a given problem faster than the synchronous methods over distributed volatile architectures that are composed of heterogeneous computing nodes which are interconnected via a high latency network. For the convergence issue, we will present, in the next chapter, a decentralized algorithm that detects efficiently the global convergence of parallel iterative asynchronous algorithms. Due to the significant advantages that the AIAC parallelization model presents, we have decided to adopt it in our work.

2.5 JACE

The environments described in the previous Chapter do not provide the functionalities required for executing AIAC algorithms over distributed architectures. Therefore, our colleagues in the AND ("Algorithmique Numérique Distribué") team, Kamel Mazouzi and Stéphane Domas, have developed a platform, called JACE [16] for Java Asynchronous Computation Environment, which is optimized for developing and executing parallel asynchronous iterative algorithms. JACE is completely developed in Java which ensures its portability over most of the existing operating systems. This platform is not just another message passing interface. It provides many mechanisms that are essential for executing AIAC algorithms. In particular, it separates the computing process from the communications. Indeed, this platform is multithreaded which allows it to allocate some threads that are dedicated for computing purposes and other ones for communicating with neighbors. This dissociation allows the computing units to asynchronously exchange their data messages with their neighbors and eliminates idle times during communications. Thus, the communications are fully overlapped by the computing process. On the other hand, JACE also allows executing synchronous iterative algorithms.
In the next subsections, we present the architecture of JACE, the centralized global convergence detection mechanism and its asynchronous communication mechanism.

2.5.1 JACE's architecture

JACE is composed of three entities:

- **The spawner**: when a user wants to execute a parallel application over a distributed architecture using JACE, it launches a spawner with the appropriate parameters (the number of required computing units, the location of the Java parallel application and the list of computing units). The spawner then sends the corresponding task to every computing unit using an hypercube propagation scheme. Afterward, the tasks are then executed over the computing units.

- **The daemon**: it runs on each computing node participating in the execution of the parallel application. The daemon executes the task that it received from the spawner and allows the computing units to communicate with each others via one of the following three communications protocols: Sockets, RMI or NIO. Therefore, each daemon has the identifiers of all the computing units participating in the execution of the application. Figure 2.6 illustrates the architecture of the daemon in JACE. It is mainly composed of two layers, the application and the communication layers. The first one handles the tasks, developed by the user, and the second one manages the exchange of data messages using the different communications protocols.

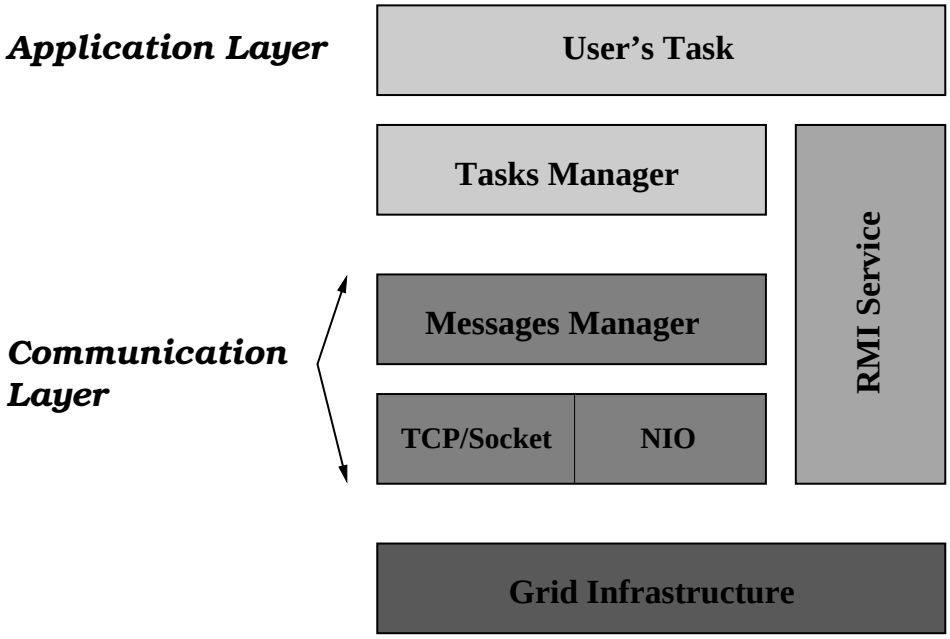

Figure 2.6: JACE daemon architecture

- **The computing task**: a parallel application that is implemented with the JACE's API (similar to the MPI's API) is composed of many computing tasks. The tasks cooperate and exchange data messages in order to solve a given problem. Each task is a light process, a thread, which allows the separation of the computing and the communication processes. Moreover, this property allows the execution of many tasks on the same computing resources.

2.5.2 Centralized Global convergence detection

The detection of the global convergence of parallel iterative algorithms have been studied for a long time. Many papers have been published in this domain. In particular we can cite [19, 63, 12]. In this subsection, we present the centralized global convergence detection mechanism implemented in JACE. This mechanism is divided into two

phases: the detection of the local convergence then the detection of the global convergence. Each daemon checks at the end of an iteration if its iterative computing process has converged locally. As described in the previous chapter, this is detected by evaluating the local residual value and comparing it to the threshold defined by the user. If the local residual is smaller than the required precision, the daemon considers that its task has locally converged and send a convergence message to the computing unit responsible of the convergence detection. Besides computing its task, this node has to detect the global convergence of the parallel application that is executed over JACE. Therefore, it checks regularly if all the daemons participating in executing the application have sent to it a convergence message. If this is the case, it declares that the system has converged globally. This mechanism is used for detecting the convergence of synchronous iterative parallel applications and is not adapted for AIAC algorithms because if a node does not receive a data message from its neighbors for period of time which is typical while executing AIAC algorithms, it could detects a false local convergence which could lead to the detection of a false global convergence. To reduce the probability of a false global convergence detection, when a node detects the local convergence of its task, it executes a predefined number of iterations to make sure that its residual is still under the threshold before sending a convergence message to the node responsible of detecting the global convergence of the system. This scheme increases the execution time of the application. Finally, this centralized global convergence detection mechanism could overload the daemon responsible of the detection and drastically slows its computing process. Therefore, in the next chapter, we present a decentralized global convergence detection algorithm that is developed by our colleagues in the laboratory, Jacques Bahi, Raphael Couturier and Sylvain Contassot. This algorithm is well suited for AIAC applications and is very scalable due to its distributed nature.

2.5.3 Asynchronous communication mechanism

As mentioned before, daemons in JACE can exchange data by passing messages. The reception and the emission of messages can be done in a synchronous or asynchronous manner. The asynchronous communication mechanism in JACE is composed of many entities:

- **Message**: it is the data structure that is composed of the following fields:
  - *Destination*: it contains the identifier of the computing unit to which this message has to be sent.
  - *Source*: it contains the identifier of the computing unit that have send this message.
  - *Tag*: it is composed of the iteration and the step numbers of the task during which this message was created on the sender.

  - *Data*: it contains the data that must be transfered from the source to the destination.

  The message is serialized before being sent to its destination.

- **Send queue**: it stores the messages before their asynchronous transfer to their destinations.
- **Reception buffer**: it stores the received messages before they are consumed by the computing process.
- **Sender thread**: it is the communicating process. It asynchronously sends the messages from the Send queue of the source to the Reception buffer of the destination.

All these entities are necessary to asynchronously exchange data messages between daemons in JACE. After computing an iteration, the daemon puts in the Send queue the messages that it intended to send to its neighbors. If the queue already contains a message that has the same destination and step number but a lower iteration number than the new message that is being added, the old message in the queue is replaced by the new one as illustrated in Figure 2.7. In the asynchronous iteration model, only the most recent data messages are sent to the neighbors. After adding a message to the queue, the Sender thread is notified and thus awoken. It retrieves the messages from the queue and sends them one after the other to their corresponding receivers. When a neighbor receives a data message, it stores it in the Reception buffer. Again if this buffer contains an older message that has the same source and step number but a lower iteration number, the old message is replaced by the new one in the buffer. Usually in an iterative parallel algorithm, after sending its data messages to the neighbors, the computing node has to wait to receive the data messages from its neighbors. This means all the computing nodes are synchronized. However, in JACE and in the asynchronous iteration model the computing nodes are not synchronized. Instead of waiting for the dependency messages of their neighbors, they get the new messages from the recipient buffer. If a certain data message has not been received yet and does not exist in the recipient buffer, they use the most recent data message they already have. As a consequence, the computing nodes do not execute each iteration using fresh data dependencies from their neighbors. But they are not blocked nor do they suffer from idle times when waiting to receive a data message from a dead or slow neighbor.

JACE is a very powerful platform for executing parallel iterative asynchronous algorithms over crash free distributed architectures. However, in reality, there are no crash free architectures and the probability for a failure to occur is proportional to the number of nodes required to compute the application and the time it takes to solve

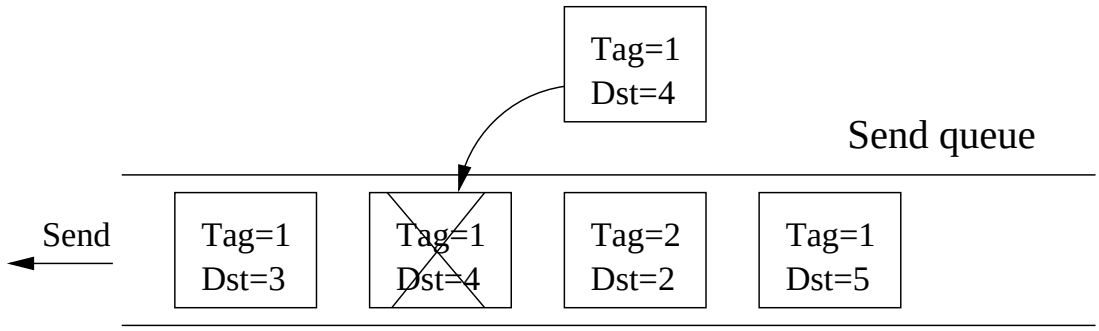

Figure 2.7: Old messages are replaced by new similar messages in the Send queue

the given problem. Therefore, JACE requires some fault tolerance policy because it is vulnerable to failures and the whole application could be terminated if a daemon crashes during the computation. In the next section, we present another platform, called JACEP2P, which implements some fault tolerance policies.

2.6 JACEP2P

JACEP2P [15] stands for Java Asynchronous Computation Environment for Peer-to-Peer architectures (Version 1) and it is an evolution of the JACE environment. It has also been developed by our colleagues at the AND team, Philippe Vuillemin and Raphael Couturier, and it is fully implemented in Java for platform independence purposes. As for JACE, JACEP2P executes parallel iterative asynchronous applications with dependencies between computing nodes. Thus any group of nodes executing a given parallel application can communicate asynchronously with each others. The main objective for creating this environment was to harness the unlimited free computing power of the public unused computing units that are interconnected via the Internet and to execute AIAC algorithms over this distributed and volatile architecture. Therefore, it was essential for this platform to implement a fault tolerance policy in order to resist to the eventual disconnection of public computing units from the distributed architecture when being used by their respective owners. On the other hand, this platform is not only dedicated for peer-to-peer architectures, it can be also used to safely execute AIAC algorithms over most of the distributed architectures like local clusters and distributed clusters.

In the next subsections, the architecture of JACEP2P is presented, its fault tolerance policy is detailed and all its advantages and drawbacks are demonstrated.

2.6.1 JACEP2P's architecture

Figure 2.8, presents the architecture of JACEP2P and the different components that form the platform. There are three kinds of entities in JACEP2P and each one has a defined role:

- The first entity is called the "super-node" (represented by a big circle in Figure 2.8). They are the access portals to the platform for the computing units that wish to participate in the computations and for the users who want to execute their parallel applications. Each super-node stores in its "register" the identifiers (IP address) of all the computing nodes that are connected to it and are not executing an application. The super-node regularly receives heartbeat messages (represented by doted lines in Figure 2.8) from the computing nodes connected to it. If the super-node does not receive a heartbeat message from a computing node included in its register for a given period of time, it declares that this computing node is dead and deletes its identifier from the register.

- The second entity is the "spawner" (represented by a square in Figure 2.8). When a user wants to execute a parallel application, it launches a spawner with the required parameters: the number of computing nodes N necessary to execute the application, the location of the application's files (e.g. their location on a Web server) and the eventual parameters for the application. The spawner then contacts a super-node to reserve N computing nodes. If the super-node does not have the sufficient amount of daemons, it contacts an other super-node and reserves the demanded daemons (the reserved daemons are removed from the super-node's register), then returns to the spawner a register containing the identifiers of the reserved daemons. When the spawner receives the register, it creates a task for each computing node and starts the execution of the tasks on the respective computing nodes. The spawner also sends its register to all the computing nodes in order for them to be able to communicate with each other. Moreover, the spawner is responsible for detecting the disconnection of a computing node that was executing a part of the application. Indeed, when the computing nodes are reserved by the spawner, they change the destination of their heartbeat messages from the super-node to the spawner. If the spawner detects that a computing node have not sent to it a heartbeat message for a while, it declares that this computing node is dead. Then, it contacts the super-node and reserves a new computing node in order to replace the dead one. The spawner initializes the new daemon, which retrieves the last backup of the dead node (see next paragraph) and continues the computing task from that checkpoint. Finally, the spawner is also responsible for detecting the global convergence of the parallel iterative application. As in JACE, this process is centralized but here it is executed over the spawner because it is considered safe and crash free and all the

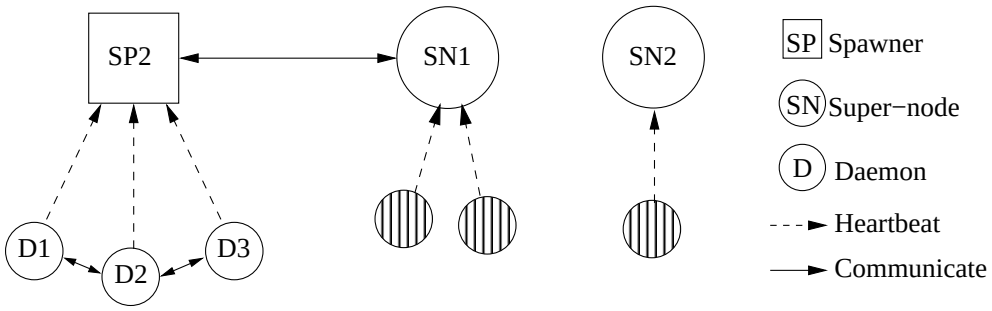

Figure 2.8: JACEP2P's architecture and different components.

daemons are volatile. This mechanism is executed exactly as the one described in JACE and suffers from the same drawbacks. Moreover, the assumption that the spawner is crash free is not real and it is always safer to integrate a fault tolerance mechanism for the spawner.

- The third entity is the "daemon" or the computing node (represented in Figure 2.8 by a hashed small circle if it is free and by a white small circle if it is executing an application). Once launched, it connects to a super-node and waits for a task to execute. During the execution of the parallel application, the daemons can communicate with each others and they regularly save their state on their neighbors. After the end of a task, the daemons reconnect to the super-node.

2.6.2 Checkpointing and restoring mechanisms

To resist to computing nodes' crashes, JACEP2P uses an uncoordinated transparent distributed checkpointing mechanism where each node regularly saves its data on its backup neighbors. Since JACEP2P executes parallel asynchronous iterative applications, the computing nodes do not have to synchronize with each others when saving their status. The frequency of the backups can be predefined by the user. When a node wishes to save its data, it creates a backup object, selects a backup neighbor using the "Round-Robbin" strategy and sends the backup to it. Figure 2.9 illustrates the "Round-Robbin" strategy in the checkpointing mechanism: at iteration i, it saves its data on the first neighbor (represented by a circle in Figure 2.9). After n iterations, it sends its new backup on the next neighbor and so on.

If the spawner detects that a daemon, executing a task, is not sending heartbeat messages to it, it considers that the daemon is dead and triggers the restoring mechanism. The spawner then contacts the super-node and reserves a new daemon to replace the dead one. The new daemon retrieves from the dead node's neighbors its last backup and continue its task from that last checkpoint. During the restoring mechanism, the rest of the daemons that are executing the same application, continue their

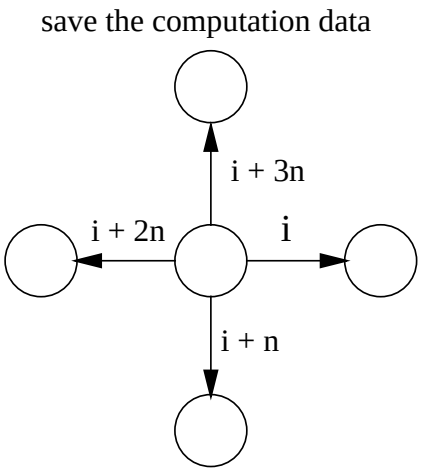

Figure 2.9: A daemon saving its data each *n* iterations on a neighbor using the "Round-Robbin"strategy in JACEP2P.

computations and they are not affected by the disconnection of the dead daemon (see section 2.4.2).

2.6.3 JACEP2P's limitations

In [15], the experiments results proved that the first version of JACEP2P performs well and presents a relatively small overhead. Nevertheless, this version had some important limits:

- JACEP2P is not fully fault tolerant. Indeed, in this version, the spawners crashes are not tolerated . Moreover, while executing the global convergence process, the platform does not resist well if a daemon goes down.
- As mentioned above, JACEP2P has a centralized failure detection mechanism. If the application is being executed by a large number of daemons, the spawner will be overloaded with heartbeat messages. This will delays the detection of a dead daemon and could even lead to a false crash detection. Moreover, if many daemons die successively and there is only one spawner to handle the dead daemons, then the spawner will take a lot of time to replace them. This may reduce the performances of the platform.
- JACEP2P has a centralized global convergence detection mechanism which is not well adapted for executing asynchronous parallel iterative algorithms on volatile architectures. The daemons executing such applications does not receive dependencies messages from their neighbors at each iteration. This may lead to a false local convergence and thus resulting to false global convergence detection. Furthermore, the spawner could be overloaded by convergence messages, if many daemons converges locally at the same time.

- JACEP2P has many centralized mechanisms like launching the application, detecting the global convergence and detecting the dead nodes. These centralizations limit the scalability of JACEP2P and create new weak points in the platform.
- In JACEP2P, each daemon receives the whole register which contains the identifiers of all the daemons executing the application. If a daemon crashes and is replaced by a new one, the spawner has to notify the modifications to all the daemons in order to update their registers. This could overload the spawner and increase the congestion of messages in the network.

2.7 Conclusion

In this chapter, we have presented three common models for parallelizing iterative methods: SISC, SIAC and AIAC. The advantages and the drawbacks of each one were explicitly demonstrated. The AIAC seems to be the most suitable model for heterogeneous volatile distributed architectures. It separates the computing process from the communications and eliminates the synchronizations between the computing units. It allows the computing process to overlap the communications in order to eliminate idle times. Afterward, we have presented two environments, JACE and JACEP2P, that are dedicated to designing and executing AIAC algorithms and explained their architecture and characteristics. They are both developed in Java and implements all the functionalities necessary for executing AIAC algorithms like asynchronous message passing and multithreaded environment. However, the main difference between the two platforms is that JACE is not fault tolerant and cannot execute parallel applications over volatile architectures while JACEP2P is fault tolerant during the computing process (under some hypothesis).
Both architectures have their limitations and weaknesses. A lot of improvements are required in order to have a reliable platform for executing AIAC algorithms over heterogeneous volatile distributed architectures. To overcome all these problems, we have implemented a new version of JACEP2P which is more optimized and scalable than this old version and offers many new features that makes it a much more reliable and robust platform. This new version is detailed in the next chapter.

Part II

Contributions

Chapter 3

JACEP2P-V2

3.1 Overview

JACEP2P-V2 is a major evolution of JACEP2P. This new version tackles almost all the issues, described in the previous chapter and that undermined the performance, stability and scalability of JACEP2P. JACEP2P-V2 contains all the functionalities that are required to execute parallel iterative applications implemented according to the asynchronous iteration model, like asynchronous messaging and multi-threading. JACEP2P-V2 also offers many new innovative mechanisms that makes it a very interesting and efficient platform for designing and executing parallel iterative asynchronous applications over distributed volatile heterogeneous architectures such as distributed clusters and global/volunteer computing architectures. JACEP2P-V2 main characteristics are:

- It is platform independent because it is implemented with the JAVA programming language which makes it usable in heterogeneous environments.
- It is completely fault tolerant and weak points free. Thus, it is usable in volatile environments.
- It is completely decentralized in order to be as much as possible scalable.
- It executes parallel iterative asynchronous applications with dependencies between the computing nodes.
- It uses the message passing model to asynchronously exchange data between the computing nodes.

JACEP2P-V2 presents many more features which are explained in details in the third section. However before detailing any mechanism implemented in JACEP2P-V2, we present in the second section the architecture of JACEP2P-V2 in order to describe its components and their general behavior.

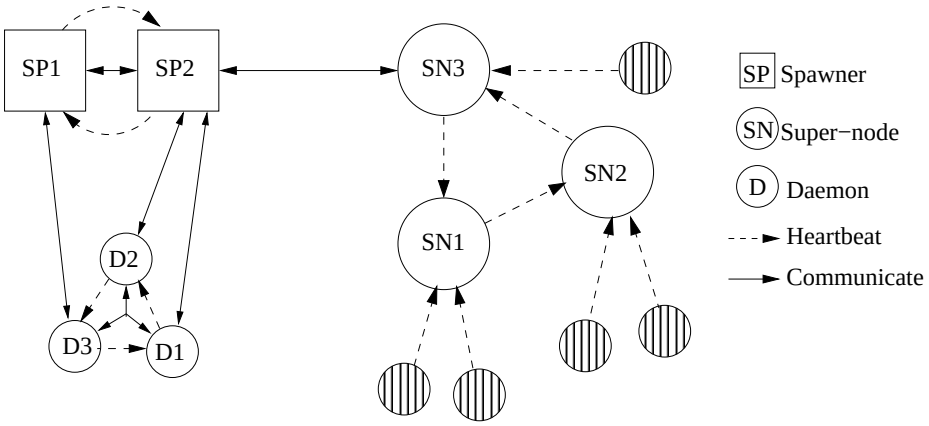

Figure 3.1: JACEP2P-V2's architecture and its different components.

3.2 Architecture

JACEP2P-V2 is composed of the same components as JACEP2P: the spawner, the super-node and the daemon. However, their functionalities and topology have been modified in the new version. Figure 3.1 shows the architecture of JACEP2P-V2 where we notice that there are two spawners handling the execution of a single application and each group of entities (spawners, daemons and super-nodes) forms a circular network. The functionalities of each component are defined as follows:

- **Super-nodes**. They now form a circular network and store in an equally distributed manner the identifiers of all the computing nodes that are connected to the platform and that are not executing any application. Figure 3.2 illustrates the architecture of a super-node. Each super-node is composed of:
  - a register which contains the identifiers of the daemons connected to the super-node.
  - a status table which contains the number of computing nodes connected to each super-node.
  - five threads: the RMI service for communicating, the heartbeat thread for signaling its status to the next super-node, the token thread (see next paragraph) and two scan threads for fault tolerance issues (see next section).

  All the super-nodes share a "token" that is passed successively from a super-node to the next one. Once a super-node has the token, the token thread is notified. The token thread executes the algorithm 3.1 where it computes the average load of the super-nodes (*avg*) using the status table. If the load of the local super-node is higher than *avg*, the extra load is distributed on the super-nodes with loads lower than *avg*. Then the status tables of all super-nodes is updated and the local super-node passes the Token to the next super-node in the circular network.

Algorithm 3.1 Token thread

1: {n = number of computing nodes connected to the super-node}
2: {avg = the average number of computing nodes connected to a super-node}
3: **if** has Token **then**
4: Compute avg using the status table
5: **if** $avg < n$ **then**
6: Send the identifiers of $n - avg$ nodes to the super-nodes that have the number of computing nodes connected to them less than avg.
7: Update the status tables of all the super-nodes.
8: **end if**
9: Send the Token to the next super-node
10: **else**
11: Wait for Token
12: **end if**

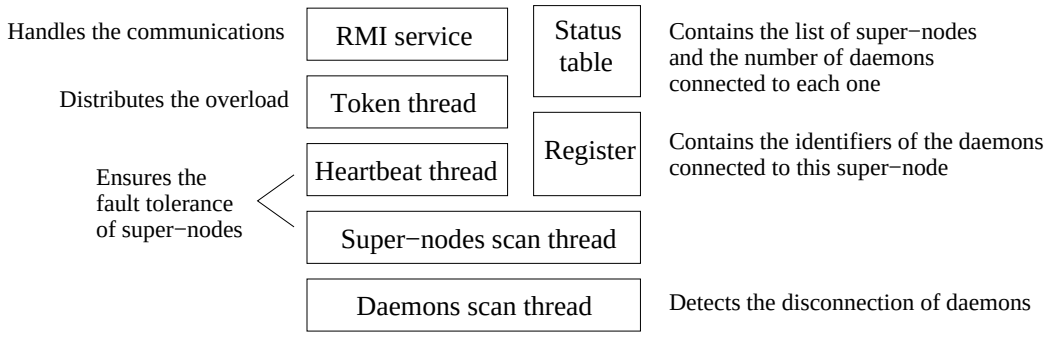

Figure 3.2: The super-node's architecture in JACEP2P-V2.

Figure 3.3 illustrates three super-nodes, represented by circles and forming a circular topology. The first super-node has the token, so it computes the avg which is equal to $\frac{26+34+0}{3} = 20$. The first super-node has six nodes more than the avg. Therefore, it sends the identifiers of the six extra nodes to super-node 3 which has fewer computing nodes connected to it than the average. The six extra nodes will start heartbeating super-node 3 and their ids will be removed from the register of super-node 1. Then, super-node 1 informs the rest of the super-nodes of this transfer. This distribution reduces the overload of the super-nodes.

- **Spawners**. Figure 3.4 illustrates the new architecture of the spawner. It is composed of an RMI service to handle the communications, the heartbeat and the spawner scan threads for fault tolerance issues (see next section), a spawners table containing the identifiers of the spawners, a register as the super-node and a task manager to assign the tasks to the reserved daemons. When a user wants to execute a parallel application that requires N computing nodes, he or she launches a spawner. The spawner contacts a super-node to reserve the N computing nodes plus some extra nodes in order to transform them into spawners for fault tolerance issues. When the spawner receives the register from the super-

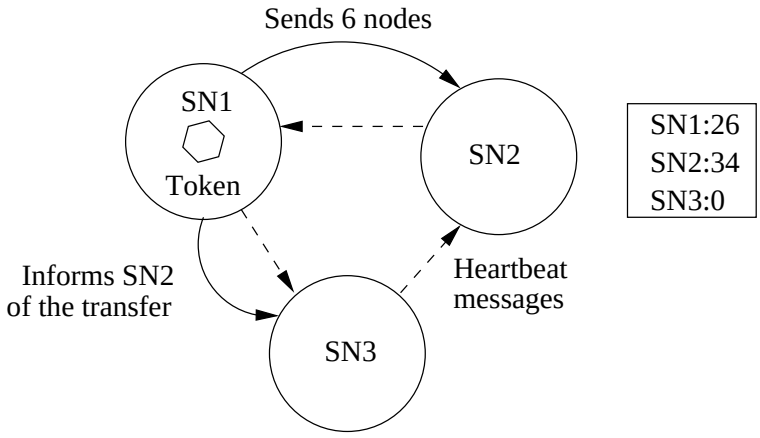

Figure 3.3: The distribution of the overload between the super-nodes in JACEP2P-V2.

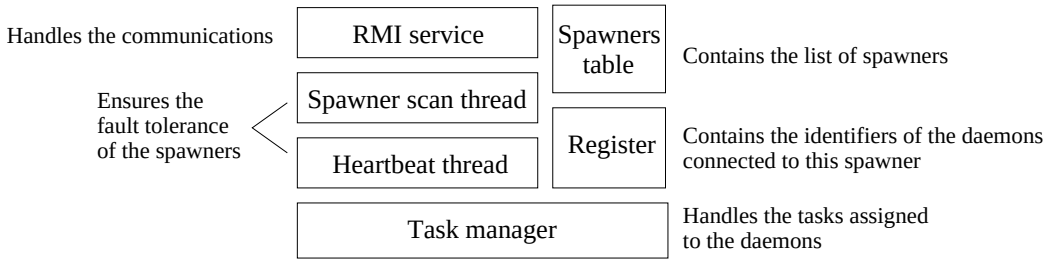

Figure 3.4: The spawner's architecture in JACEP2P-V2.

node, it transforms the extra daemons into spawners and stores the identifiers of the rest of the daemons in its own register. Once the extra nodes are transformed into spawners, they form a circular network and they receive the register containing the identifiers of all the computing nodes. Then each spawner becomes responsible for a subgroup of computing nodes. It then starts the tasks on these computing nodes and sends a specified register to them. So each computing node receives a specified register that only contains the identifiers of the daemons it interacts with and that depends on the application being executed. These specified registers reduce the number of messages sent by the spawners to update the register of the daemons after the crash of a daemon because just a small number of daemons is usually affected by a crash.

- **Daemons**. As in JACEP2P, the daemon executes the tasks that it receives from the spawner. Figure 3.5 illustrates the architecture of the daemon. It mainly includes:
  - The task thread which executes the task.
  - The sender thread which asynchronously sends via the RMI service the data messages stored in the message queue, to their respective destinations,
  - The message buffer which receives the data messages from the neighbors and stores them in order to be consumed by the task thread.

– the backups threads which save the data on the neighbors.

More information on each component are given in the next section. When executing an application, the daemons also form a circular network. This topology is only used for the failure detection mechanism and each daemon can directly communicate with other daemons if it has their identifiers in its register.

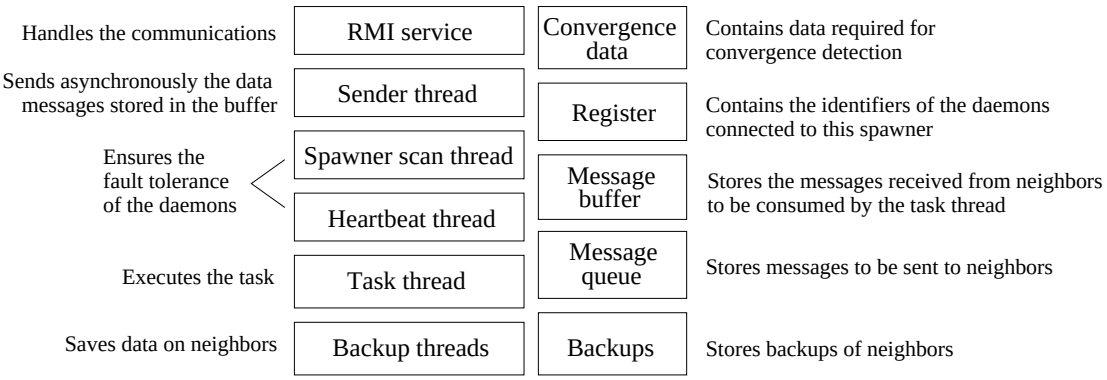

Figure 3.5: The daemon's architecture in JACEP2P-V2.

3.3 Characteristics and functionalities

After describing the architecture of JACEP2P-V2 and the role of each component, in this section we present in details the different new functionalities and characteristics implemented in JACEP2P-V2.

3.3.1 A Completely fault tolerant platform

In the section describing JACEP2P, we mentioned that the platform only tolerates the disconnection of daemons during the computing phase and that failures are prohibited for daemons during the convergence detection phase. The crashes are also never tolerated for spawners. However, since the user have no control over the disconnection of the computing nodes and there are no stable nodes in a volatile environment, the assumptions described above can never be met. Therefore, we have introduced into JACEP2P-V2 some new mechanisms that make all three entities that form the core of JACEP2P-V2, fault tolerant. In particular, we have implemented in the three entities a decentralized crash detection mechanism. It enables the neighbors of a node to detect if it is dead or alive and is based on the Dual model explained in Section 1.5.1. Each group of entities forms a circular network. This organization is required to apply the decentralized crash detection mechanism. As shown in the architectures' figures in the previous section, each entity has a "heartbeat thread" that signals regularly to the next node in the circular network that the sender is still alive and another thread, the "scan

thread" , that tests at each iteration if the previous node in the circular network has recently sent a heartbeat message. Figure 3.6, illustrates three nodes forming a circular topology and executing the mechanism described above. However, in Figure 3.6, node 1 crashes and thus it stops sending heartbeat messages to node 2. So, the scan thread in node 2 detects after a given period of time that it is not receiving heartbeat messages from the previous one and suspects that the previous node is probably dead. It then tries to contact the dead node. If it does not answer, node 2 triggers the restoring mechanism to handle this disconnection. The restoring mechanism directly depends on the type of the dead node (daemon, spawner or super-node). Indeed, each entity has a restoring mechanism which is also dependent on the saving mechanism used for each type of entity.

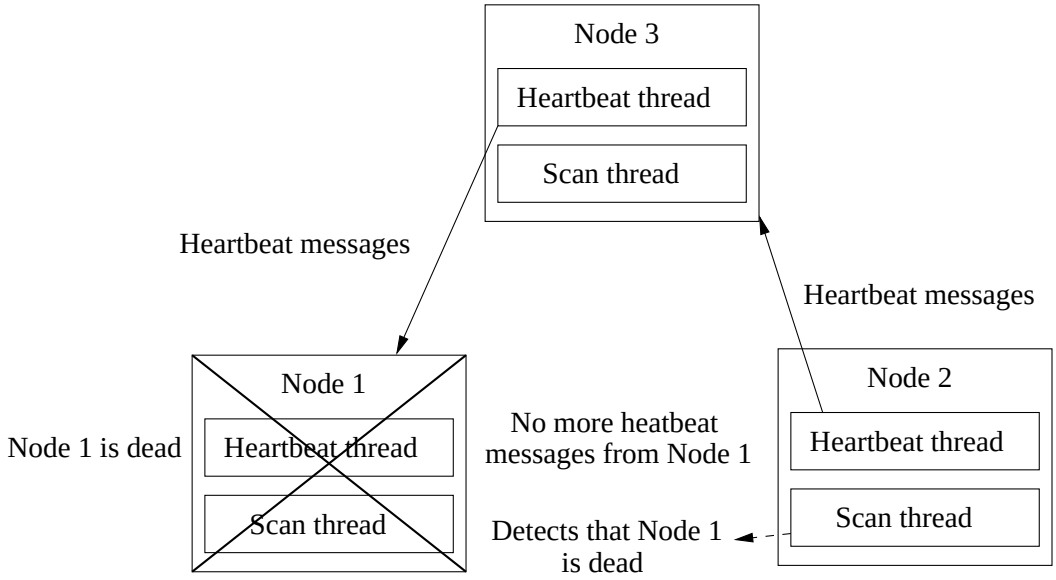

Figure 3.6: The fault detection mechanism.

- For the daemons, we use the distributed backup mechanism described in Section 2.6.2 and we have implemented two types of backup in JACEP2P-V2. The first backup, called the data backup, contains all the information concerning the state of a node (convergence data) and its computing process (solution vector). This backup is saved each N iterations (N is given by the user and usually depends of the length of an iteration) on a different neighbor using the "Round-Robin" strategy. On the other hand, the second backup, called the status backup, only contains the status data. This backup has a smaller size and it is saved when the status of a daemon has changed, especially when it concerns the global convergence detection mechanism. This backup is saved on all the backup neighbors simultaneously. Once a daemon detects that the previous daemon is dead, the daemon signals it to the spawner responsible for it. The spawner contacts a super-node and acquires a new daemon. The new daemon replaces the dead one

and retrieves the last status backup and the last data backup from the neighbors of the dead node. Once it has the backups, it continues the task from that last checkpoint. During all this operation, all the other daemons continue their tasks normally and they are not affected by the disconnection of their neighbor.

- For the spawners, we use the replication mechanism. The spawner is duplicated into many spawners which also form a circular network. All the spawners have all the information concerning all the daemons and each one only manages a subgroup of daemons. If a spawner dies, the next spawner in the circular network detects it (using the same scheme described before). Then, that spawner contacts a super-node, reserves a new daemon and transforms it into a spawner. Once it becomes a spawner, it receives the register containing the identifiers of all the daemons executing the application, it identifies its subgroup of daemons, it informs them that it is the new spawner responsible of them and it is reintegrated into the circular spawner network.

- For the super-nodes, there is no backup mechanism. They do not contain very valuable information. When a dead super-node is detected by the next super-node in the circular network, it is removed from the circular super-node network and all the daemons that were connected to it will reconnect to another super-node.

3.3.2 Completely decentralized

JACEP2P-V2 is completely decentralized. In fact, all the tasks are divided between the entities of the same type. For example, a daemon can be connected to any super-node and the group of super-nodes shares equally the control of the free daemons that are connected to the super-node network. The spawners are also decentralized: once a spawner is launched to execute an application, it quickly duplicates itself into several spawners (depending on the number of daemons required to execute the parallel application) by transforming some daemons into spawners. Each spawner becomes responsible for starting the application on a subgroup of daemons and handling the needs of that subgroup. For the computing nodes, each one executes a part of the application and the sum of their work gives the solution of the global problem. Moreover, they save their data on their neighbors. This decentralized backup mechanism reduces bottlenecks and does not overload the backup nodes. Furthermore, all the three types of entities, implements a decentralized fault detection mechanism and a decentralized global convergence detection mechanism, presented in the next subsection. This distribution of tasks, allows JACEP2P-V2 to solve very large problems and thus to become very scalable with theoretically no limiting conditions.

3.3.3 Multi-threaded

JACEP2P-V2 is multi-threaded which enables it to take advantage of multi-cores computing units which became very common. Indeed, in JACEP2P-V2, the computing process is never blocked by the exchange of data messages between daemons and each functionality in JACEP2P-V2 (communicating, detecting crashes, saving and computing) has its own thread. There are the sender thread for exchanging data messages, the heartbeat thread and the scan thread for detecting a node's disconnection or failure and a new thread is created each time a daemon needs to save its data on a backup neighbor. For the the computing process, the solver is usually chosen by the user and depends on the numerical problem he is trying to solve. There are a lot of numerical solvers that are multi-thread and that take benefits of the multi-cores processors. For example, in our experiments we used MTJ [2] (Matrix Toolkits for Java) which offers a very wide range of high-performance data structures and algorithms for numerical computing. MTJ is natively multi-threaded.

3.3.4 The decentralized global convergence detection algorithm

Since the centralized mechanism for detecting the global convergence of parallel asynchronous iterative algorithms is not efficient nor scalable, we have implemented the decentralized global convergence detection algorithm [12, 11], designed by Bahi et al., into JACEP2P-V2. However, this algorithm is not well adapted for volatile environment. Therefore, we have modified this algorithm in order to make it fault tolerant. In the next paragraph, we present a general description of the algorithm (that we called DCD) and afterwards we list the modifications we made to let it be fault tolerant.

3.3.4.1 Description

Before beginning the global convergence detection mechanism, each daemon has to detect if its subsystem has converged locally: after each iteration, each daemon compares its "local residual vector" which is computed using the max norm ($R = max_i(|X_i^{k+1} - X_i^k|)$ where X_i^k denotes the value of the component i of array X at iteration k), to the precision requested by the user (ϵ). If the residue is smaller than ϵ we say that the system has converged locally. However, the residue does not always decrease uniformly, especially for asynchronous iterative parallel algorithms because a computing node does not receive fresh dependencies from all neighbors at each iteration. So, the residue may oscillates around the threshold which can lead to a false detection of the global convergence. To limit early false local convergence detection, we use the *pseudo-period* concept. It is the smallest number of iterations where a node receives at least one dependency message from each of its neighbors and executes an iteration using these new data. We say that a node has converged locally if its residue stays under the threshold (ϵ) during one or many pseudo-periods. The pseudo-period

concept is illustrated in Figure 3.7 where a node (Node 1) locally converges and then applies the pseudo-period concept. It waits two iterations to receive fresh dependency messages from its two neighbors and then executes a third iteration using these dependencies to be sure that its subsystem is still converged.

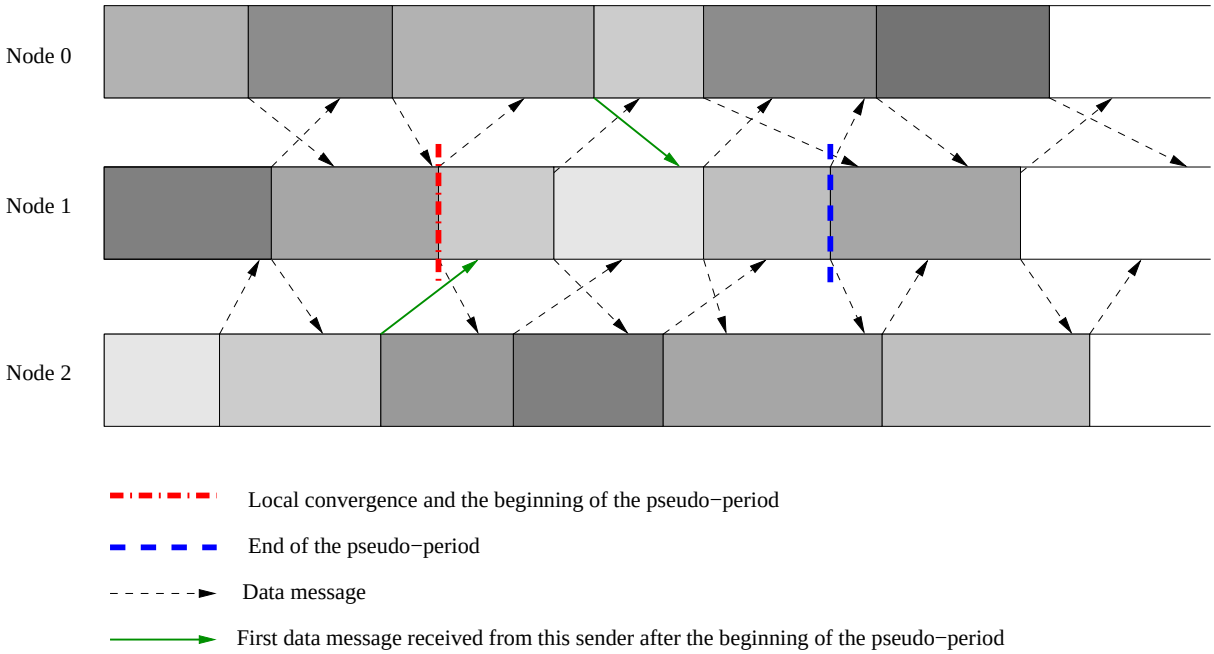

Figure 3.7: The pseudo-period concept.

The global convergence detection algorithm is composed of two phases, the global convergence detection phase and the verification phase. The first phase is based on the "Leader Election" algorithm [8, 32] which chooses dynamically a node to execute a specified task (here, it is the detection of the global convergence). In order to apply this algorithm, we transform the computing graph into a non cyclic graph. Then, when a computing node converges locally, it checks how many of its neighbors have not converged yet (this number is noted as s):

- If $s > 1$, the node continues its iterative process.
- If $s = 1$, it means that the node is a leaf in the tree or that all its neighbors but one have converged (that implies that its neighbors sub-trees have also converged). In this case, the node sends a *convergence message* to its neighbor that did not converge yet. When the neighbor receives this message, it decrements by one its s.
- If $s = 0$, it means that all the neighbors of the node (and all their sub-trees) have converged. Therefore, it is the last node to converge locally and it is elected as *Leader* in order to detect the global convergence of the system.

Figure 3.8 illustrates the global convergence detection phase. The computing graph have been transformed into a non cyclic graph and besides each computing node (represented by a circle) appears the number of its neighbors that did not converge yet. As the daemons compute their iterations, the system evolves and some subsystems converges locally (represented with filled circles). Each time a subsystem converges locally, the Leader election algorithm described above is applied. Finally, the successive applications of this algorithm lead to the election of a Leader as shown in the last sub-figure in the Figure 3.8. After electing a Leader, the verification phase begins. Fig-

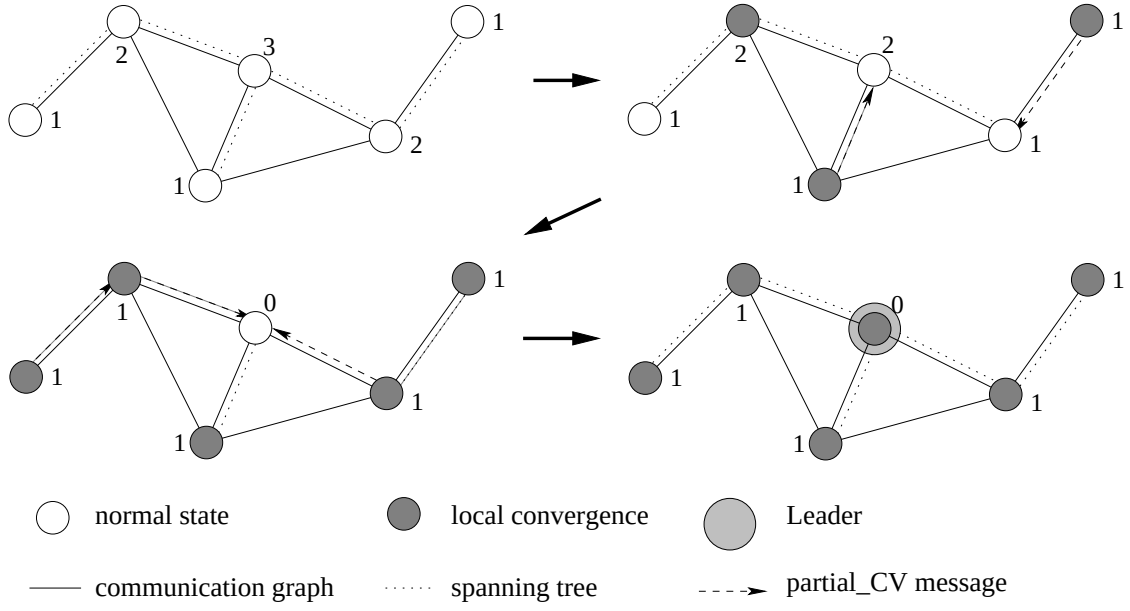

Figure 3.8: The global convergence detection phase of the DCD algorithm.

ure 3.9 illustrates the different steps of this phase. There are four steps to perform in order to be sure that all the nodes are still in a local convergence state:

1. The Leader broadcasts a *verification message* to all its neighbors which also transmit the message to all their neighbors. In this way, the verification message is propagated to all the computing nodes and signals the beginning of the verification phase. This step is illustrated in the first sub-figure of Figure 3.9.

2. After receiving the verification message, each node begins a new pseudo-period in which it waits for new data (computed after the leader has been chosen) from all its neighbors. Then, it computes an iteration using these new dependencies. If, since sending the convergence message until the end of the pseudo-period, the residue does not ever cross over the threshold, the response to the verification phase is positive, otherwise it is negative. This phase allows the system to verify if the state of the nodes has evolved.

3. If a node computes a negative response, it sends it directly to the neighbor that sent him the verification message which propagates it to the Leader. On the other hand, if the response is positive, the node has to wait until its neighbors send him positive responses. Then it propagates the response to the Leader.

4. If the Leader receives or computes a negative response, it broadcasts directly a negative *verdict message* to all its neighbors which also propagates the verdict to all the nodes. When a node receives a negative verdict, it starts the whole global convergence mechanism all over again. This step is illustrated in the two sub-figures on the left hand of Figure 3.9. On the other hand, if the Leader develops and receives from all its neighbors positive responses, it broadcasts a positive verdict to all the computing nodes. This positive verdict means that the system has converged globally. When a node receives a positive verdict, it ends its iterative process. This step is illustrated in the two sub-figures on the right hand of Figure 3.9.

3.3.4.2 Critical procedure and backups

To be able to transform the DCD algorithm into a fault tolerant algorithm, we have defined a critical procedure as follows:*"each sequence of instructions that affects two neighbors and blocks one of them if that node crashes before saving its state"*. Thus, in the DCD algorithm, if a node $n1$ locally converges and all its neighbors but one ($n2$) have also converged, $n1$ has to send a convergence message to $n2$. This message signals to $n2$ that $n1$ has converged. $n2$ decrements by one the number of its neighbors that did not converge yet and returns an acknowledge message (see subsection 3.3.4.3) to $n1$ which means that $n2$ has well received the convergence message. Once $n1$ receives the acknowledge message, it waits for the verification phase and stops sending convergence messages. Such a sequence of instructions is a critical procedure because it affects two neighbors ($n1$ and $n2$), modifies their states and if one of the two nodes dies during this procedure, the algorithm will be blocked indefinitely. In fact, if $n2$ dies or disconnects from the platform, the spawner detects that this node has not been sending any heartbeat message for a while, so it considers that it is dead. Then, the spawner tries to replace this dead node: it contacts a super-node and requests an available node. The reserved node $n3$ replaces $n2$ and to be able to continue the task, retrieves the last backup. If $n2$ died after returning the acknowledge message to $n1$, all the modifications, made after receiving the convergence message, are lost and $n3$ have retrieved an old backup with a false number of neighbors that have not converged yet. $n3$ will wait indefinitely for a convergence message from $n1$. To solve this problem, $n2$ must save its data before sending the acknowledge message to $n1$.

On the other hand, if $n1$ receives the acknowledge message and dies just before saving, the daemon replacing the dead node will retrieve an old backup. Thereafter, it

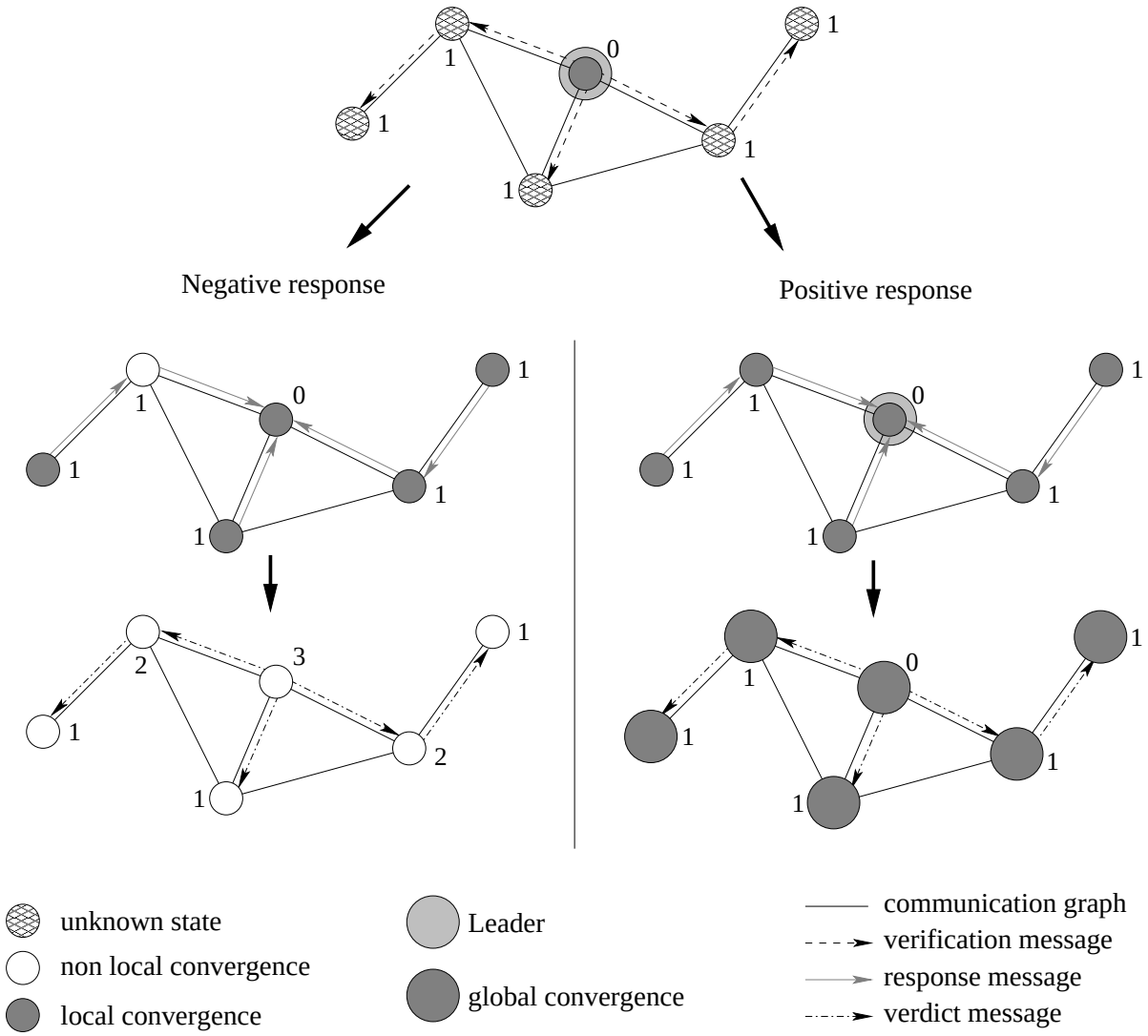

Figure 3.9: The verification phase.

detects again the local convergence of its task and sends a new convergence message to $n2$. The recipient notices that it has already received this message, so it ignores it but returns an acknowledge message to $n1$ so it can execute the next phase.

Identifying a critical procedure while detecting the global convergence is very difficult. The easiest approach is to save data after the execution of each instruction concerning the global convergence detection. This solution is not practical given its execution cost. Thus, it is necessary to determine the critical sequences of instructions and to minimize their numbers (see Section 3.3.4.4, for more information on the different critical procedures in the DCD algorithm).

Since the DCD algorithms requires saving the convergence data after a critical procedure, we have created a new backup to save the convergence data that contains all

the values necessary to detect the global convergence and maintains the system's coherence. Distinguishing between the computing and the convergence backups reduces the bad influence of checkpointing on the computing process. Actually, the size of the convergence backup is very small compared to the size of the computing backup. So the convergence data can be saved more frequently than the computing data without having a great impact on the performance of the computing process.

3.3.4.3 Acknowledge messages

JACEP2P-V2 tolerates the loss of data messages when it executes parallel asynchronous iterative algorithms. However, as described in the previous paragraph, JACEP2P-V2 has to ensure the right reception of the convergence messages by the receivers in order to ensure a coherent system. Indeed, The DCD algorithm is composed of many phases and if a crash prevents a daemon from receiving a message that informs it that it has to pass to the next phase, the daemon and its sub-trees nodes will be stuck in this phase while the others will pass onto the next one. Therefore, we have to make sure that all the notifications concerning the global convergence detection are well received by their recipients. To accomplish that, we propose two solutions. First, the recipient treats the message and saves the results of that treatment on its neighbors. Then it returns the acknowledge message to the sender. Second, the recipient just saves the information sent in the message and returns the acknowledge message to the sender then handles the saved information afterward. Each approach has its advantages and drawbacks:

- When using the first solution, while the recipient treats the convergence message and saves the results on its neighbors, the sender remains blocked. However the recipient has to save just once after processing the message's data.

- When using the second solution, the recipient saves the message's data and quickly returns the acknowledge message to the sender. However, after processing the message's data, it saves the results again on its neighbors.

To be able to choose between those two methods, we must evaluate the execution time cost for these two solutions. Unfortunately, it is very hard to achieve this test because it depends on many parameters like the network's latency, the processing speed of nodes and the number of components solved on each node. In practice we can use the first method for messages that do not require a large time to be processed (like local convergence messages) and the second method for the broadcasting messages because they could block the sender for a long time (while the message is propagated to all the sub-trees nodes).

3.3.4.4 Overview of the fault tolerant algorithm

In the implementation of the DCD algorithm into JACEP2P-V2, we organized the computing nodes into an hypercube topology to reduce the broadcast time of the convergence messages. If n is the number of computing nodes, the maximum number of nodes that a message has to cross is equal to x with x equal to the smallest integer that verifies $n < 2^{x+1}$. In our implementation of the DCD algorithm, each node discovers its convergence neighbors by executing Algorithm 3.2.

Algorithm 3.2 Discovering convergence neighbors for an hypercube tree

1: $\{n = number\ of\ computing\ nodes\}$
2: $\{id = rank\ of\ the\ current\ computing\ node\}$
3: $d \leftarrow 0$
4: **while** $2^d < n$ **do**
5: **if** $id < 2^d$ and $id + 2^d < n$ **then**
6: $id + 2^d$ is a convergence neighbor
7: **end if**
8: **if** $id < 2^{d+1}$ and $id > 2^d$ **then**
9: $id - 2^d$ is a convergence neighbor
10: **end if**
11: **end while**

As described before, the decentralized convergence detection algorithm is decomposed into many phases. In this paragraph, we will show how each phase has been modified in order to make the whole algorithm fault tolerant.

- **Before local convergence:** each node computes its iterative task and at the end of an iteration it asynchronously exchanges dependencies vectors with its neighbors then evaluates its residue in order to detect the local convergence. Throughout this phase only computation data are saved on neighbors because in the asynchronous iteration model the loss of dependencies messages is tolerated and does not affect the convergence of the parallel iterative application (i.e., the solution of the studied problem). This phase corresponds to line 5 to 9 in Algorithm 3.3.

- **After detecting the local convergence:** this is the first critical procedure. When a node detects that its residue is smaller than the requested precision, it computes a pseudo-period. It waits to receive new dependencies vectors from all its neighbors, then it computes a new iteration using these new data. If the residue evaluated after this iteration is still smaller than the threshold, the node declares its local convergence. This phase corresponds to line 12 to 16 in Algorithm 3.4. As mentioned before in the DCD algorithm, the locally converged node tests the number of its unconverged neighbors ($NbNeighboursNotConv$): if $NbNeighboursNotConv = 1$, it executes the sequence of instructions illustrated in Figure 3.10 and from line 26 to 33 in Algorithm 3.4. If $NbNeighboursNotConv = 0$

the node declares itself as Leader and propagates the beginning of the verification phase on all its neighbors as in Figure 3.11 and from line 17 to 24 in Algorithm 3.4. If a node involved in this procedure fails, the node recovery mechanism described in Section 3.3.4.2 is applied.

- **Verification phase:** when a node receives a verification message it executes the sequence of instructions illustrated in Figure 3.12 and in the *ReceiveVerification* procedure in Algorithm 3.5 where it begins the verification phase. During the verification phase, each node waits for new dependencies tagged with the verification tag. Afterward it computes a new iteration using those new data. If during all these operations the residue is still under the threshold, the node elaborates a positive response, otherwise it elaborates a negative response.

- **Responding to the verification phase:** if a node elaborates or receives a negative message, it directly sends it to the sender of the verification message as detailed in Algorithm 3.4 from line 72 to 79. On the other hand, if it receives from all its neighbors (except from the sender of the verification message) positive responses and elaborates a positive response, it positively responds to the sender of the verification message. This case is presented in Algorithm 3.4 from line 81 to 87 and sending a response is illustrated in Figure 3.10. If the response is positive, the sender saves its computing and convergence data on its neighbors. This phase is also considered as a critical procedure. As above, if a node, involved in this procedure, fails, the node recovery mechanism described in Section 3.3.4.2 is applied.

- **Declaring a verdict:** when the Leader receives or elaborates a negative response, it broadcasts a negative verdict to all its neighbors as in Figure 3.11 and in Algorithm 3.4 from line 53 to 60. Then it restarts the convergence detection algorithm all over again. When the Leader elaborates and receives from all its neighbors positive responses, it sets its state to global convergence and broadcasts a positive verdict to all its neighbors, as illustrated in Figure 3.11 and in Algorithm 3.4 from line 61 to 68. This is also a critical procedure and the node recovery mechanism described in section 3.3.4.2 may be applied, if a node involved in this procedure fails.

- **Receiving a verdict:** when a node receives a verdict, it executes the sequence of instructions illustrated in Figure 3.12 and in the *ReceiveVerdict* procedure in Algorithm 3.5. If the verdict is negative it reinitializes its convergence variables and restarts the convergence detection algorithm for the same step. On the other hand if the verdict is positive, it sets its state to global convergence and begins computing the next step or terminates the application. Here is the fourth critical procedure. If a node, involved in it, fails, the node recovery mechanism described in Section 3.3.4.2 is applied.

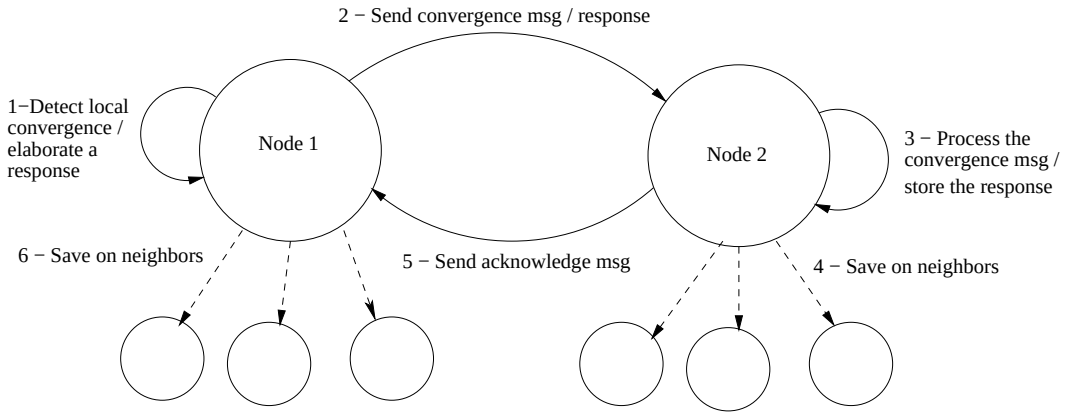

Figure 3.10: The instructions executed when sending a convergence or a response message.

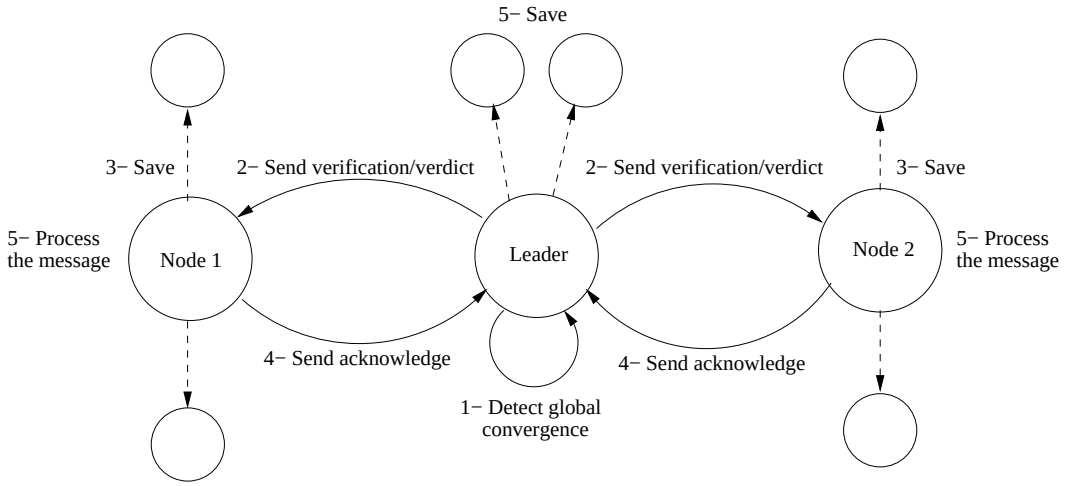

Figure 3.11: The instructions executed when broadcasting a verification or a verdict message.

3.3.5 Reduction functions

Many numerical parallel problems requires computing a reduction function at the end of a step using data from all the computing nodes. There are two methods to compute these functions: the first one is centralized. All the computing nodes are synchronized and they send their data to a central node which executes the reduction function using all these data. Then, the central node sends the result of the reduction function to all the computing nodes. Using the second method, all the computing nodes send directly or indirectly their data to each others, then each computing node evaluates the reduction function and continues its computing process. These two methods are

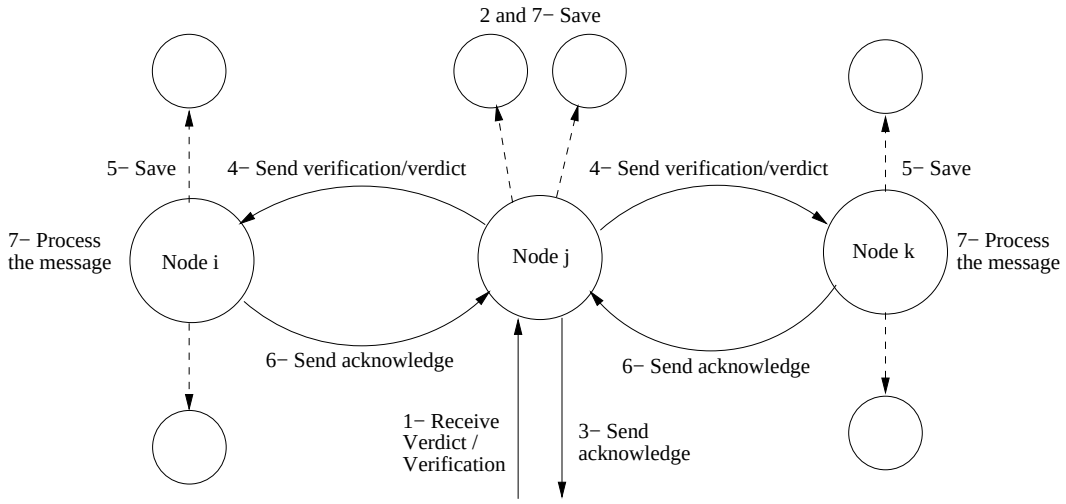

Figure 3.12: The instructions executed when receiving a verification or a verdict message.

Algorithm 3.3 Main()

```
 1: for each step do
 2:    Initialize the global convergence variables
 3:    globalConvergence ← false
 4:    state ← NORMAL
 5:    while globalConvergence = false do
 6:       Compute an iteration
 7:       if Residue < Threshold then
 8:          pseudoConvergence ← true
 9:       end if
10:       DetectGlobalConvergence(pseudoConvergence)
11:    end while
12: end for
```

very efficient in a stable environment like a cluster or a grid. However, these methods are not well adapted to volatile environments. In fact, in a volatile environment, we cannot have a central node because it could die at any moment and we cannot use the data messages because they could be lost if the receiver is dead or if there is a failure in the network. To remedy this problem, we have adopted a decentralized method to compute the reduction functions. We merged the reduction function to the decentralized global convergence detection algorithm. So, when the verification phase of the DCD algorithm begins and if the parallel iterative application being executed, requires a reduction function, each computing node sends the data (required for computing the reduction function) in the response message. Once a daemon receives a response message, it computes the reduction function using its data and the data contained in the response message. At the end, the Leader will receive the responses containing

the results of the reduction functions applied to the data of the nodes forming each subtree connected to the Leader. The Leader applies the reduction function using all these data, adds the result to the verdict message and broadcasts the verdict message to all the nodes. Since we use the response and the verdict messages which returns an acknowledge message when received, we are sure that the messages will not be lost. Furthermore, the results of the local reduction functions are saved when evaluated by the computing nodes.

3.4 Conclusion

In this chapter, we have presented the JACEP2P-V2 platform. It is a fault tolerant and distributed computing environment which is dedicated for executing parallel iterative algorithms based on the asynchronous iteration model over volatile architectures. The architecture of JACEP2P-V2 which is mainly composed of three types of nodes (the spawner, the daemon and the super-node) was presented in details. Moreover, all the functionalities and the mechanisms implemented in JACEP2P-V2 were exposed. In particular, we have detailed the fault detection and the restoring mechanisms which make the platform fault tolerant and the decentralized global convergence detection mechanism that detects efficiently the global convergence of parallel iterative asynchronous applications.

In the next chapter, we present the experiments that we have conducted over the Grid'5000 testbed in order to test the performances of JACEP2P-V2 and its scalability. We will also expose the different problems that we solved using asynchronous parallel iterative methods that were executed on JACEP2P-V2. Moreover, we will compare JACEP2P-V2 with its predecessor JACEP2P in order to prove the efficiency of the new mechanisms that we have implemented in the new version. All these experiments were executed over volatile heterogeneous computing nodes that were located on distant sites.

Algorithm 3.4 DetectGlobalConvergence(pseudoConvergence) (1/2)

```
1: if action = SendVerification / action = SendVerdict then {To broadcast the verification
   and verdict messages}
2:    Broadcast a verification message / Broadcast the verdict
3:    Wait for an acknowledge message from each neighbor
4:    if received an acknowledge message from each neighbor then
5:       action ← nothing
6:       Save Convergence Data on neighbors
7:    else
8:       Broadcast the message on the next iteration
9:    end if
10: end if
11: if state = NORMAL and action = nothing then
12:    if pseudoConvergence = false then
13:       reinitialize the pseudo-period
14:    else
15:       if PseudoPeriod = true then
16:          localConvergence ← true
17:          if NbNeighboursNotConv = 0 then
18:             BroadcastVerification()
19:             Node ← Leader
20:             state ← Verification
21:             Wait for an acknowledge message from each neighbor
22:             if received an acknowledge message from each neighbor then
23:                Save Convergence Data on neighbors
24:             end if
25:          else
26:             if NbNeighboursNotConv = 1 then
27:                Send convergence message to the neighbor that did not converge
28:                Wait for an acknowledge message from the neighbor
29:                if received an acknowledge message from the neighbor then
30:                   state ← WAITforVerification
31:                   Save Convergence Data on neighbors
32:                end if
33:             end if
34:          end if
35:       end if
36:    end if
37: else
38:    if state = WAITforVerification then
39:       if pseudoConvergence = false then
40:          localConvergence ← false
41:          Save Convergence Data on neighbors
42:       end if
43:    else
44:       see that part on page 72...
45:    end if
46: end if
```

Algorithm 3.4 DetectGlobalConvergence(pseudoConvergence) (2/2)

```
47: if (state = NORMAL and action = nothing) or state = WAITforVerification then
48:   see that part on page 71...
49:   if state = Verification then
50:     if node = leader then
51:       if pseudoConvergence = false or localConvergence = false or received a negative
          response then
52:         Broadcast a negative verdict
53:         Reinitialize the convergence variables
54:         Wait for an acknowledge message from each neighbor
55:         if received an acknowledge message from each neighbor then
56:           Save Convergence Data on neighbors
57:         end if
58:       else
59:         if PseudoPeriod = true and received positive responses from all neighbors then
60:           Broadcast a positive verdict
61:           state ← Finished
62:           Wait for an acknowledge message from each neighbor
63:           if received an acknowledge message from each neighbor then
64:             Save Convergence Data on neighbors
65:           end if
66:         end if
67:       end if
68:     else
69:       if node did not send a response yet then
70:         if pseudoConvergence = false or localConvergence = false or received a negative
            response then
71:           if action = nothing then
72:             Send a negative response to the neighbor that sent the verification message
73:             Wait for an acknowledge message from the neighbor
74:             if received an acknowledge message from the neighbor then
75:               Save Convergence Data on neighbors
76:             end if
77:           end if
78:         else
79:           if PseudoPeriod = true and received positive responses from all neighbors ex-
              cept one then
80:             Send a positive response to the neighbor that did not send a response
81:             Wait for an acknowledge message from the neighbor
82:             if received an acknowledge message from the neighbor then
83:               Save Convergence and computing Data on neighbors
84:             end if
85:           end if
86:         end if
87:       end if
88:     end if
89:   end if
90: else
91:   if state = FINISHED then
92:     globalConvergence ← true
93:   end if
94: end if
```

Algorithm 3.5 Reception methods

procedure: ReceiveConvergence()

1: **if** $reloading = false$ and action=nothing **then**
2: $NbNeighboursNotConv \leftarrow NbNeighboursNotConv - 1$
3: Save convergence data on its neighbors
4: Return an acknowledgment message to the sender
5: **end if**

procedure: ReceiveVerification()

1: **if** $reloading = false$ **then**
2: **if** $state = WAITforVerification$ **then**
3: $action \leftarrow SendVerification$
4: Initialize variables for verification phase
5: $state \leftarrow Verification$
6: Save convergence data on its neighbors
7: Return an acknowledge message to the sender
8: **end if**
9: **end if**

procedure: ReceiveResponse(Response)

1: **if** $reloading = false$ **then**
2: Store the response in the response vector
3: Save convergence data on its neighbors
4: Return an acknowledge message to the sender
5: **end if**

procedure: ReceiveVerdict(verdict)

1: **if** $reloading = false$ **then**
2: **if** $verdict = true$ **then**
3: $state \leftarrow FINISHED$
4: **else**
5: Reinitialize convergence variables
6: **end if**
7: $action \leftarrow SendVerdict$
8: Save convergence data on its neighbors
9: Return an acknowledge message to the sender
10: **end if**

Chapter 4

Solving Numerical Problems on Volatile Architectures

4.1 Introduction

In this chapter, we present our experimental work which is decomposed into two categories. The first one is dedicated for testing and evaluating the performances of the JACEP2P-V2 platform by executing large parallel iterative asynchronous applications over volatile heterogeneous architectures. Moreover, we compare the performances of JACEP2P-V2 to those of the previous version. The aim of these experiments was to prove the efficiency, the robustness and the scalability of JACEP2P-V2. Since this environment is dedicated to executing AIAC algorithms, the experiments, listed above, are composed of applications implemented according to this model. In the second part of the experimental work, we tackle our initial objective which is to solve linear and non linear systems over distributed volatile architectures. Therefore we evaluate some resolution methods that solve these problems. In particular, we concentrate our research on the Waveform Relaxation method [55, 67] which solves initial values problems and is compatible with the asynchronous iteration model. However, since this method has not been tested on distributed architectures before, we have first of all evaluated its performance in such environments by comparing it to the PVODE [24] solver. The method was implemented in C in order to have a fair comparison with the PVODE solver which is written in C. Since the results obtained in these experiments were encouraging, we have pursued our research on the Waveform Relaxation method. We have implemented it according to the asynchronous iteration model and we have ported the code to Java in order to execute it over JACEP2P-V2 because it is the only platform capable of executing iterative parallel asynchronous applications over volatile and distributed architectures. To evaluate the performance of the asynchronous Waveform Relaxation method over distributed volatile architectures, we compared it to the asynchronous Multisplitting-Newton method [17] which also solves initial value problems is compatible with the asynchronous iteration model.

The results showed the efficiency of the asynchronous Waveform relaxation method in distributed volatile environments.

The rest of this chapter is organized as follows: in the next section, we describe in details the different resolution methods used to solve linear and nonlinear equation systems. In particular, we present the PVODE solver, the Multisplitting-Newton method (for nonlinear systems), the Multisplitting method [13] for linear systems (coupled with the Conjugate Gradient method) and the Waveform Relaxation method. In the third section, we present the three problems that we tried to solve in our experiments using various resolution methods. These problems are: the advection-diffusion problem, the reaction-diffusion problem and the NAS Parallel Benchmark (with the Conjugate Gradient test). The experiments and their results are presented in the fourth section. As explained above, they are decomposed into two parts. In the first one, we evaluate the performances of JACEP2P-V2 over volatile environments. In the second part, we compare at first PVODE to the Waveform Relaxation method in a stable high latency environment because PVODE is not fault tolerant. Then we compare the Multisplitting-Newton method to the Waveform Relaxation method while executing both methods over volatile high latency environments using JACEP2P-V2.

4.2 The resolution methods

As mentioned before, we are interested in solving differential equations which arise from the simulation of physical and natural phenomena. In particular we focus on large differential equations, in a distributed cluster environment. There are two types of differential equations, the *ordinary differential equations* and the *partial differential equations*. An ordinary differential equation (ODE) is a relation that contains functions of only one independent variable, and one or more of its derivatives with respect to that variable.

Let y be an unknown function of x.

$$y : \mathbb{R} \rightarrow \mathbb{R}$$

An ODE of order n involving y has the form:

$$F(x, y, y', y'', \ldots, y^{(n)}) = 0$$

where $y^{'} = dy/dx$ is the first derivative with respect to x, and $y^{(n)} = d^n y/dx^n$ is the nth derivative with respect to x.

An ODE dependent of time is called a non stationary ODE or an initial value problem (IVP). An initial value problem of order n has the following general form:

$$F(t, y, y', y'', \ldots, y^{(n)}) = 0 \quad y(t_0) = y_0$$

where y is a function of t, t_0:initial time and y_0:initial values.

A partial differential equation (PDE) is an equation involving functions and their partial derivatives. For example, the wave equation is a PDE and it has the following form:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} = \frac{1}{v^2} \frac{\partial^2 \psi}{\partial t^2}$$

To solve a PDE, we can use the method of lines ("MOL") [64] which approximates the PDE with a large ODE by discretizing the partial derivatives. In the next paragraphs we present some techniques for solving ODE systems:

4.2.1 PVODE

PVODE is a parallel solver for large systems of ordinary differential equations in a distributed low latency environment. It is an extension of the sequential package known as CVODE [28] which has been widely distributed and used. The parallelization of CVODE to PVODE was accomplished through the modification of the vector kernels, allowing the solver to operate on vectors that have been distributed across processors. The message passing calls between nodes are made through MPI [58]. Although PVODE contains many methods for the resolution of both stiff and non-stiff IVPs, the standard approach to solve ODE systems is based on three steps:

- Converting the differential equations describing the system into a sequence of non-linear algebraic equations, using the Adams-Moulton Formula [25] to integrate non-stiff problems and the Backward Difference Formula (BDF) [52] for stiff ones.
- Transforming the non-linear algebraic equations into a sequence of linear problems using a modified Newton method for systems converted by the BDF method or using a functional method for the ones generated by the Adams-Moulton integration.
- Solving the system of linear equations with Gaussian Elimination like methods or iterative ones.

To parallelize this method, the system of linear equations is solved with a parallel solver where each node have to communicate with its neighbors when computing boundaries' values. Thus, as shown in [51], this method is fine grained. It synchronizes at each internal step. Therefore, if the implementation of this method is executed over an architecture that suffers from high latency communications, the performance of this method is severely reduced due to the extremely penalizing and frequent synchronizations, and communications.

4.2.2 The Waveform Relaxation method with Euler

First of all we describe the Euler method which can be used with the Waveform Relaxation method to solve nonlinear systems. Then, we present the coupling of the Waveform Relaxation method with the Euler method.

4.2.2.1 the Euler method

The Euler method is a first order numerical sequential procedure for solving ordinary differential equations (ODEs) with a given initial value. Consider the ordinary differential equation:

$$\frac{du(t)}{dt} = f(u(t), t) \tag{4.1}$$

in which $u(t)$ is the mesh points vector and f is a nonlinear function. We can solve this differential equation by approximating the left hand of the equation, using the explicit Euler method which produces the equation:

$$u(t + DT) = DT * f(u(t), t) + u(t) \tag{4.2}$$

where DT is the discrete fixed time step and $u(t + DT)$ is the vector u at time $t + DT$.

This method can be parallelized according to synchronous models. However, the resulting fine grained parallel methods are not adapted for heterogeneous distributed and volatile architectures. An alternative is to couple the Waveform Relaxation method with Euler to develop a coarse grained parallel iterative algorithm that solves ODEs and that is well suited to such architectures.

4.2.2.2 The Waveform Relaxation method coupled with Euler

In the early 1980's, the Waveform Relaxation method (WR) was introduced by E. Lelarasmee as an efficient parallel iterative method for solving large sparsely coupled differential equations systems that are generated by the simulation of integrated circuits. Since then, this method has been extended and applied to various other application areas. With the WR approach, the system of equations is decomposed spatially into sets of equations. Each set is solved iteratively by using values from previous iterations: each computing unit integrates its equations on the whole time interval without communicating with its neighbors. At the end of an iteration, each task exchanges with its neighbors its boundaries' values which are used in the evaluation of the next iteration. This procedure is executed iteratively until the solution vector converges to a stable solution.

The convergence of Waveform Relaxation methods is generally slow and often the parallel execution gain is not sufficient to compensate for the slowness of the convergence. However, there are different methods to accelerate this convergence. Among

them, the most usable schemes are the overlapping [53] and the windowing [69] concepts.

Overlapping: with this technique every node computes a small number of components already allocated to its neighbors, so that each node solves additionally to its own components a percentage of the components computed by its neighbors. This redundancy helps minimizing the error on the extremity points that depend on the unavailable results of the neighbor's frontier components. Figure 4.1 shows how the division of the components vector and the data exchanges between the nodes are changed if the overlapping concept is applied: each node has a small percentage of its neighbors components added to its initial components (which is illustrated by doted lines) and each node transfers the components of index equal to $l + 2 * overlap$ for left boundary and $r - 2 * overlap$ for right boundary, with $overlap$ equal to the number of components added to the initial local components vector, r the index of the right boundary and l the index of the left boundary. The benefits of this concept have been illustrated in [22].

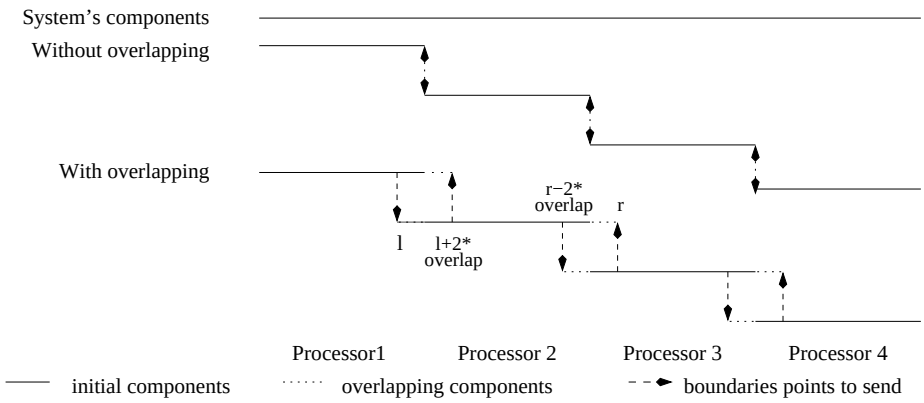

Figure 4.1: The decomposition of the system with/without overlapping. The doted lines simulate the overlapping and the arrows simulate the data exchanges.

Windowing: usually, using the Waveform Relaxation method, we integrate the ODE on the whole time interval at each iteration. This procedure slows down the convergence rate of the method because in contrast with PVODE, every node integrates its equations on a long time interval without communicating with its neighbors, using only the given initial values. A natural solution to this problem is to divide the time interval into windows and iterate on each window until convergence. After each iteration on a window, every node sends its new boundary values to its neighbors. Figure 4.2 displays an ODE system divided between four nodes and the time interval is divided at least into two windows.

Although both concepts are efficient in accelerating the convergence rate of the WR method, it is very difficult to initially choose the amount of overlapping or the size

of a window that gives the optimal results, i.e. a faster convergence. Note that some works have attempted to create an adaptive windowing, but we do not focus on this approach in this document.

Many methods can be coupled with the WR method to integrate the system on each time step. At first, we used the sequential solver found in the CVODE package but we had some convergence problems when we tried to solve large ODEs using a lot of computing nodes. These problems resulted from the adaptative discretization scheme adopted in CVODE and the heterogeneity of the subproblems being solved on each node. Indeed, since the boundaries' values change at each window, CVODE is unable to well approximate the solution vector. Therefore, we used the explicit Euler method to solve this problem. This method proved to be efficient in terms of precision and performance in solving large differential equations when coupled with WR.

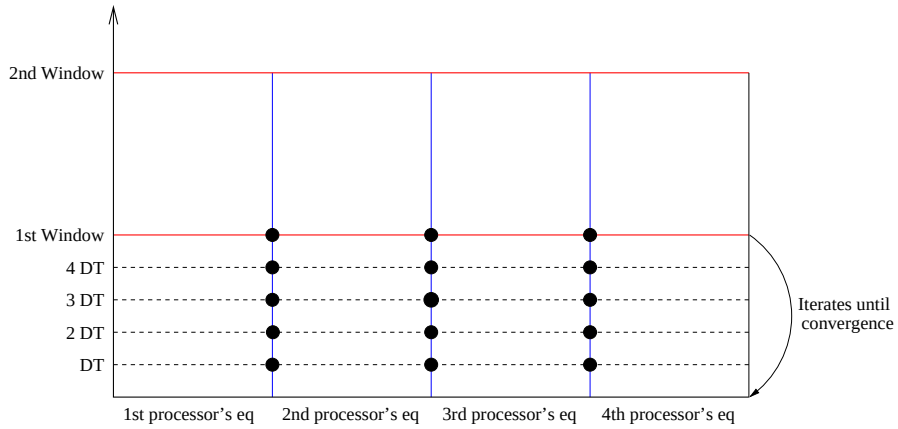

Figure 4.2: The windowing concept in the WR method: the system is equally split into four subsystems and the time interval is divided into several windows where each window contains multiple DTs

Algorithm 4.1 illustrates the main steps of the asynchronous Waveform Relaxation method coupled with Euler. Once the system has been initialized, the time interval is decomposed into windows. For each window, an iterative procedure is applied: the window is decomposed into fixed small time steps DT. Each computing units, integrates the system on each DT using the Euler method. Then the new boundaries values are stored in a buffer. After integrating the system on the whole window, the boundaries values, stored in the buffer at each DT, are exchanged between neighbors. Then, each computing unit integrates the system again on the same time window while using the received values. This procedure is repeated until the system converges to the solution. Once the convergence is reached, in conformity with the chosen threshold, it begins integrating the system on the next time window.

Algorithm 4.1 The asynchronous Waveform Relaxation-Euler algorithm

```
1: Split the system's components between the computing nodes
2: Add the overlapped components to the local components
3: uLoc = array containing the local components
4: Set initial values
5: for Each window in the considered time interval do
6:    Copy uLoc into olduLoc
7:    repeat
8:       for Each step of the current window do
9:          Compute the values of the local components using the received boundaries values
            and uLoc{in our algorithm using the explicit Euler method}
10:         Store the new boundaries values in a buffer
11:         Compute the local error
12:      end for
13:      Send asynchronously the stored boundaries values to neighbors
14:      Non blocking reception for boundaries values from neighbors
15:      Global convergence detection
16:      if Not converged then
17:         Copy olduLoc into uLoc
18:      end if
19:   until Global convergence
20: end for
```

4.2.3 The Multisplitting-Newton method

In this section, we first of all describe the Newton method which could be used in conjunction with the Multisplitting method to solve nonlinear systems. Then, we present the Multisplitting-Newton method which is a parallel iterative method, compatible with the asynchronous iteration model.

4.2.3.1 The Newton method

To solve equation (4.1), we can also use an implicit time integration method that transforms the system into:

$$\frac{u(t+DT)-u(t)}{DT} = f(u(t+DT), t+DT), \tag{4.3}$$

where DT is a fixed time-step.

The main differences between the explicit and the implicit methods is that the explicit methods calculate the state of a system at a later time from the state of the system at the current time (like in Equation 4.2), while implicit methods find a solution by solving an equation involving both the current state of the system and the later one (like in Equation 4.3). It is clear that implicit methods require more computation than explicit methods (solving the equation), and they can be much harder to implement.

However, implicit methods are often used because many problems arising in real life are stiff, for which the use of an explicit method requires impractically small time steps to keep the error in the result bounded. For such problems, to achieve a given accuracy, it takes much less computational time to use an implicit method with larger time steps. That said, whether one should use an explicit or implicit method depends upon the problem to be solved.

The solution to the nonlinear system 4.3 is computed using the standard Newton method. This approach leads to an iterative scheme, given an initial approximation u^0:

$$J * d^{k+1} = -F(u^k) \tag{4.4}$$

where J is the Jacobian matrix of $F(u^k)$ (the Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function with $J_{ij} = \frac{\partial F_i(u)}{\partial u_j}$), $d^{k+1} = du(t + DT)^{k+1}$ and $F(u^k) = F(u(t + DT)^k, u(t), t) = u(t) + DT * f(u(t + DT)^k, t + DT) - u(t + DT)$. For more information, the reader can refer to [14].

Solving the equation (4.4) is equivalent to finding the solution of a linear system at each iteration. One can notice that the Jacobian matrix is sparse. In practice, the quasi-Newton method is preferred. It consists in computing the Jacobian matrix only at the first iteration of a given time step in order to reduce the execution time, since this part is often very time consuming. However, the quasi-Newton may require a slightly higher number of iterations than the Newton method to converge.
From a parallel point of view, two approaches are possible. The first one consists in using a parallel sparse linear solver. In this case, a synchronization is required at each Newton iteration and unless using an asynchronous sparse linear solver, synchronizations are required between each iterations of the solving process. The alternative is called the Multisplitting-Newton method. It is described below.

4.2.3.2 The Multisplitting-Newton method

The asynchronous Multisplitting-Newton method (MN) has some similarities with block decomposition techniques. The principle is to split the initial domain into several sub-domains in order to assign one of them to each computing unit involved in the parallel computation. In our case, the Multisplitting-Newton algorithm allows us to solve equation (4.4) in parallel.

The Multisplitting-Newton's decomposition is illustrated in figure 4.3. In fact, the Jacobian matrix is split into blocks and du and F are decomposed in a compatible manner and are respectively called $dLoc$ and $FLoc$ (because each part is a local one assigned to a computing unit). Dependencies on the left and the right, illustrated by parts with 0 in the figure, can be ignored since those parts of the Jacobian are taken into account in the right hand side. This solution has the advantage of ignoring large parts of the Jacobian matrix which simplifies its implementation.

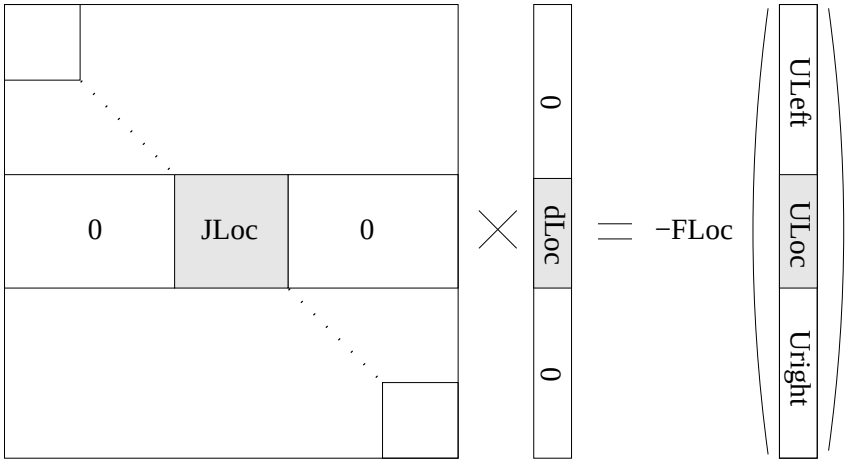

Figure 4.3: the decomposition of the Jacobian matrix, vector solution and function in the Multisplitting-Newton method.

Algorithm 4.2 The asynchronous Multisplitting-Newton algorithm

```
1: oldu = Array containing vector u at the previous iteration
2: uLoc = Array containing local components
3: -FLoc = Local part of the function used to approximate the ODE
4: JLoc = Local part of the Jacobian matrix
5: dLoc = Local part of du, the solution of the linear system obtained with Newton
6: Initialization of variables, especially oldu and uLoc
7: for each step of the considered time interval do
8:    repeat
9:       Computation of boundary conditions if processor is concerned
10:      Computation of the Jacobian matrix JLoc at the first iteration with -FLoc, uLoc and
         oldu
11:      Computation of FLoc with uLoc and oldu
12:      dLoc=LinearSolver(JLoc,-FLoc)
13:      uLoc=uLoc+dLoc
14:      Send asynchronously the boundaries values to neighbors
15:      Non blocking reception for boundaries values from neighbors
16:      Global convergence detection
17:   until Global convergence
18:   Copy uLoc into oldu
19: end for
```

Algorithm 4.2 summarizes the main ideas of the Multisplitting-Newton algorithm. After the initialization part, the main loop is executed until the considered simulation time is reached. For each time step, the algorithm iterates on the Newton process. At each iteration the computing process applies, if necessary, the boundary conditions. This algorithm uses the quasi-Newton method, therefore it only computes the Jacobian matrix at the first iteration. Afterward, *FLoc* is computed using both arrays *uLoc* and

oldu. As explained previously, the exchanged vector between neighbors is not *du* but *u*. That is why, in the algorithm, the vector *u* at the previous iteration (*oldu*) is not a local vector, since it contains values computed by some neighbors. The following step allows the algorithm to solve the local linear system composed of the Jacobian matrix and *FLoc*. The solution is used to set up the value of the vector *uLoc*.

Once the convergence is reached, in conformity with the chosen threshold, values of *uLoc* are copied into *oldu* at the right location. For each time step the quasi-Newton method is applied, resulting in the computation of the solution of the ordinary differential equation.

4.2.4 The Multisplitting method for linear systems

In this section we describe the association of Multisplitting method with the the Conjugate Gradient method to solve linear systems. The resulting parallel iterative method uses the Multisplitting concept to decompose the linear system and then each subsystem is solved using the sequential Conjugate Gradient method. This method is also compatible with the asynchronous iteration model.

4.2.4.1 The sequential Conjugate Gradient

The Conjugate Gradient (CG) belongs to both minimization and projection methods. It solves systems of linear equations ($Ax = b$ where the $n \times n$ matrix A is symmetric positive definite, the unknown vector x and right hand vector b are of dimensions n).

At each iteration, $x^{(k+1)}$ is updated in the following way:

$$x^{(k+1)} = x^k + \alpha^{(k+1)} p^{(k+1)} \tag{4.5}$$

where $\alpha^{(k+1)}$ is a multiple of the search direction vector $p^{(k+1)}$ and $\alpha^{(k+1)} = (r^{(k)}, r^{(k)})/(Ap^{(k+1)}, p^{(k+1)})$ ((a, b) denotes the scalar product of vector a and b). The residual vectors are updated in the same way:

$$r^{(k+1)} = r^k + \alpha^{(k+1)} q^{(k+1)} \tag{4.6}$$

where $q^{(k+1)} = Ap^{(k+1)}$. Now the search directions are updated as follows: $p^{(k+1)} = r^{(k+1)} + \beta^{(k)} p^{(k)}$ where $\beta^{(k)} = (r^{(k)}, r^{(k)})/(r^{(k-1)}, r^{(k-1)})$. This process iterates until a given threshold is reached. Clearly, at each iteration, two inner products and one matrix-vector product are required. It is shown in [68] that this last operation represents about 90% of the total computation time.

4.2.4.2 The Multisplitting-Conjugate Gradient method

In the multisplitting-Conjugate Gradient parallel method, the A matrix is split into horizontal rectangle parts (see figure 4.4). Each of these parts is then affected to a processor. In this way, a processor is in charge of computing its *XSub* part by solving the

following subsystem: $ASub * XSub = BSub - DepLeft * XLeft - DepRight * XRight$. $XSub$ must then be sent to other processors which depend on it.

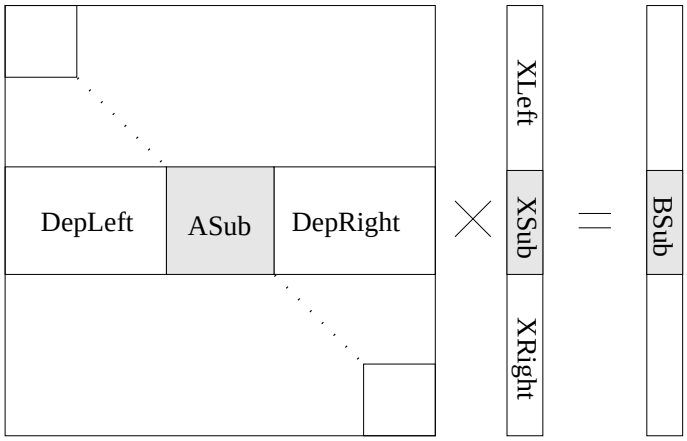

Figure 4.4: The decomposition of the system using the Multisplitting method for linear systems

The method can be decomposed into four phases:

1. **Data decomposition.** In this phase, data are allocated to each processor assuming the decomposition exposed in figure 4.4. Then, each processor iterates until convergence on the following phases.

2. **Computation.** Firstly, each processor computes $BLoc = BSub - DepLeft * XLeft - DepRight * XRight$. Then, it solves $ASub * XSub = BLoc$ by using a sequential version of the Conjugate Gradient method.

3. **Data exchange.** Each processor sends its $XSub$ part to its neighbors. Here, the neighborhood is closely related to the density of the A matrix. Clearly, a dense matrix implies an *All-to-all* communication scheme while a matrix with a small bandwidth reduces the density of the communication scheme.

4. **Convergence detection.** Each processor computes its local convergence and depending on the type of the global convergence detection method that is applied the global convergence can be detected in a centralized or decentralized manner.

It is possible to modify the data decomposition in order to obtain non disjoint rectangular matrices, to apply the overlapping concept, in order to improve the convergence speed.

4.3 The problems studied in the experiments

In this section we present the problems solved in the experiments using JACEP2P-V2.

4.3.1 The Advection-Diffusion problem

The Advection-Diffusion equation represents mathematically the transport processes of pollutants, salinity, and so on, combined with their bio-chemical interactions. It follows an initial boundary value problem for a nonlinear system of PDEs, in which nonlinearity only comes from the bio-chemical interspecies reactions.

4.3.1.1 Mathematical description

A system of 3D advection-diffusion-reaction equations has the following form:

$$\frac{\partial c}{\partial t} + A(c,a) = D(c,d) + R(c,t) \tag{4.7}$$

where c denotes the vector of unknown species concentrations, of length m, and the two vectors

$$A(c,a) = [\mathbf{J}(c)] \times a^T, \tag{4.8}$$
$$D(c,d) = [\mathbf{J}(c)] \times d \times \nabla^T, \tag{4.9}$$

respectively define the advection and diffusion processes ($\mathbf{J}(c)$ denotes the Jacobian of c with respect to (x,y,z)). The local fluid velocities u, v and w of the field $a = (u,v,w)$ and the diffusion coefficients matrix d are supposed to be known in advance. A simulation of pollution evolution in shallow seas is obtained if a is provided by a hydro-dynamical model. The chemical species dynamic transport is defined by both advection and diffusion processes, whereas the term R includes interspecies chemical reactions and emissions or absorption from sources.

Following a common approach, $R(c,t)$ can be expressed using production and loss terms, denoted respectively by P and L:

$$R(c,t) = P(c,t) - L(c,t). \tag{4.10}$$

Both terms can be further refined:

$$P(c,t) = P_I(c,t) + P_S(t),$$
$$L(c,t) = (L_I(c) + L_S(t)) \times c.$$

While, the terms P and L indexed by I denote the contributions (emission and absorption) from chemical interspecies reaction, the contributions from sources are indexed by S.

As we consider a test problem in three spatial dimensions, reduced to two chemical species, the system described by Equation (4.7) becomes a system of two PDEs:

$$\begin{pmatrix} \frac{\partial c^1}{\partial t} \\ \frac{\partial c^2}{\partial t} \end{pmatrix} + \begin{pmatrix} \nabla c^1 \times a \\ \nabla c^2 \times a \end{pmatrix} = \begin{pmatrix} \nabla \cdot \left((\nabla c^1) \times d \right) \\ \nabla \cdot \left((\nabla c^2) \times d \right) \end{pmatrix} + \begin{pmatrix} R^1(c,t) \\ R^2(c,t) \end{pmatrix}, \tag{4.11}$$

where ∇c is the space derivative of vector c and matrix d satisfies

$$d = \begin{pmatrix} \epsilon(x) & 0 & 0 \\ 0 & \epsilon(y) & 0 \\ 0 & 0 & \epsilon(z) \end{pmatrix}, \tag{4.12}$$

and the vector function $R(c,t)$ is defined according to:

- production terms P_I and P_S:

$$P_I(c,t) = \begin{pmatrix} q_4(t)\, c^2 \\ q_1 c^1 c^3 \end{pmatrix}, \quad P_S(t) = \begin{pmatrix} 2q_3(t)\, c^3 \\ 0 \end{pmatrix}, \tag{4.13}$$

- loss terms L_I and L_S:

$$L_I(c) = \begin{pmatrix} 0 & q_2 c^1 \\ q_2 c^2 & 0 \end{pmatrix}, \quad L_S(t) = \begin{pmatrix} q_1 c^3 & 0 \\ 0 & q_4(t) \end{pmatrix}. \tag{4.14}$$

Clearly, the coupling of the two PDEs is induced by the reaction term R.

As far as transport is concerned, in this work vertical advection (dimension z) is neglected, and the velocity field vector a is supposed to have a constant value. Hence, a is given by:

$$a = (u, v, w) = (-V, -V, 0), \tag{4.15}$$

with $V = 10^{-3}$. Regarding diffusion coefficients, horizontally they are positive constants, whereas the vertical ones vary. Thus we have a matrix d of the form:

$$d = \begin{pmatrix} K_h & 0 & 0 \\ 0 & K_h & 0 \\ 0 & 0 & K_v(z) \end{pmatrix}, \tag{4.16}$$

where $K_h = 4.0 \times 10^{-6}$ and $K_v(z) = 10^{-8} \times \exp\left(\frac{z}{5}\right)$.

For the reaction term (apart from the two unknown concentrations of the contaminants, i.e. c^1 and c^2), the different quantities in equations (4.11)-(4.14), are chosen as follows:

- $c^3 = 3.7 \times 10^{16}$, $q_1 = 1.63 \times 10^{-16}$ and $q_2 = 4.66 \times 10^{-16}$,
- $q_3(t), q_4(t)$ are chosen according to ($j = 3, 4$),

$$\begin{aligned} q_j(t) &= \exp\left[\tfrac{-a_j}{\sin(\omega t)}\right] \quad \text{if } \sin(\omega t) > 0, \\ q_j(t) &= 0 \qquad\qquad\qquad\quad \text{if } \sin(\omega t) \leq 0, \end{aligned} \tag{4.17}$$

  using the following parameters: $\omega = \pi/43200$, $a_3 = 22.62$ and $a_4 = 7.601$.

For more information concerning the mathematical description of the problem the reader can refer, for example to [66]. On the other hand, for more information concerning the implementation of the problem, we can cite [48].

4.3.2 The reaction-diffusion system

A reaction-diffusion system describes how the concentration of one or more substances, distributed in space, changes under the influence of two processes: local chemical reactions in which the substances are converted into each other, and diffusion which causes the substances to spread out in space. The reaction-diffusion system, also known as the Diffusion Equation [47], has the following form:

$$\frac{\partial u}{\partial t} = B + u^2 v - (A+1)u + \alpha \frac{\partial^2 u}{\partial x^2}$$
$$\frac{\partial v}{\partial t} = Au - u^2 v + \alpha \frac{\partial^2 v}{\partial x^2}$$

In our experiments, $A = 3$, $B = 1$ and $\alpha = 1/50$.
Here u and v denote chemical concentrations of reaction products, A and B are concentrations of input reagents which are taken to be constant and $\alpha = \frac{d}{L^2}$ where d is a diffusion coefficient and L a reactor length. To solve this problem, we apply the MOL technique and we obtain a large ODE.

4.3.3 The NAS parallel benchmark 3.0

The Numerical Aerodynamic Simulation (NAS) program, which is based at the NASA Ames research center, has developed a set of benchmarks, called the NAS parallel benchmarks [29], that are dedicated to the performance evaluation of highly parallel supercomputers. These benchmarks consist of five parallel kernels and three simulated application benchmarks. All details of these benchmarks are only specified algorithmically in order to make them independent of programming languages and distributed architectures constraints. The user implementing the benchmarks on a given system is expected to solve the various problems in the most appropriate way for the specific system. The choice of data structures, algorithms, processor allocation and memory usage are all (to the extent allowed by the specification) left open to the discretion of the implementer.

In our experiments, we solved the Conjugate Gradient benchmark using the Multisplitting method (coupled with a sequential Conjugate Gradient method) which is adapted to the asynchronous iteration model. It uses a conjugate gradient method to compute an approximation of the smallest eigenvalue of a large, sparse, symmetric positive definite matrix with a random pattern of nonzeros. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication.

4.4 Experimentations

In this section we present our experimental work which is divided into two parts. In the first subsection, we present some experiments that compare JACEP2P-V2 to JACEP2P and show the efficiency, scalability and robustness of JACEP2P-V2. In particular, we compare the two platforms while solving the 3D advection-diffusion problem over a local cluster and over distributed clusters and we test the scalability of JACEP2P-V2 by solving a large 3D advection-diffusion problem with a large number of computing units. Finally, we solve the CG NAS Benchmark over JACEP2P-V2 which shows its compatibility with different problems. In the second subsection, we evaluate some resolution methods that solve initial value problems. In particular, we concentrate our research on the Waveform Relaxation method which is compatible with the asynchronous iteration model. However, since this method has not been tested on distributed architectures before, we have first of all evaluated its performance in such environments by comparing it to the PVODE solver. The methods were implemented in C in order to have a fair comparison with the PVODE solver which is written in C. Since the results obtained in these experiments were encouraging, we have pursued our research on the Waveform Relaxation method. We have implemented it according to the asynchronous iteration model and we have ported the code to Java in order to execute it over JACEP2P-V2 because it is the only platform capable of executing iterative parallel asynchronous applications over volatile and distributed architectures. To evaluate the performance of the asynchronous Waveform Relaxation method over distributed volatile architectures, we compared it to the asynchronous Multisplitting-Newton method. The results showed the efficiency of the asynchronous Waveform relaxation method in distributed volatile environments. Most of the experiments were conducted over the Grid'5000 platform described in Section 1.3.

4.4.1 Comparison between JACEP2P and JACEP2P-V2

4.4.1.1 First experiment: local cluster

In this experiment, we compare JACEP2P (with the decentralized convergence detection algorithm) to JACEP2P-V2 while solving the advection-diffusion 3D problem using the Multisplitting-Newton method. This application solves a system containing 405,224,000 components and that simulates a 90 seconds time interval. 252 bi-processors computing units, located in Orsay, were used to run this application. The computing nodes were equipped with 2 AMD Opteron 246 2.0GHz or 250 2.4GHz processors. To prove that both platforms are fault tolerant, we used a shell script that randomly kills three computing nodes each n seconds and launches again the dead daemons after a short period of time.

The results for this set of experiments are presented in table 4.1. It shows the execution times taken by JACEP2P and JACEP2P-V2 to solve the problem with various

The frequency of the 3 crashes: n	∞	90	60	30
Execution time for JACEP2P	522s	873s	1003s	1611s
Total number of crashes for JACEP2P	0	30	51	159
Execution time for JACEP2P-V2	495s	565s	595s	744s
Total number of crashes for JACEP2P-V2	0	18	28	68

Table 4.1: Execution time taken to solve the 3D advection-diffusion problem using 252 machines located on a single site and while killing 3 random computing nodes every n seconds

frequencies of nodes crashes. It is clear that JACEP2P-V2 outperforms JACEP2P in each category. We also notice that JACEP2P-V2 is less affected than JACEP2P by the disconnection of computing nodes. Indeed, when the computing nodes disconnect frequently, JACEP2P suffers a lot because of the centralized nature of some of its components. On the other hand, with the JACEP2P-V2's decentralized dead nodes detection, the dead nodes are detected faster by their neighbors and thus they are quickly replaced by new ones to continue their tasks. Although during the recovery the daemon has to reinitialize the task which for some problems could be highly time consuming, thanks to the newly implemented mechanisms the influence of crashes on the performance of JACEP2P-V2 platform is drastically reduced.

4.4.1.2 Second experiment: distributed clusters

In this second set of experiments, we aimed at simulating a global computing architecture which has the following characteristics: large number of heterogeneous computing units, high latency communications and volatile nodes. So, we used the same number of computing nodes but this time we have chosen them from three distant sites in order to have heterogeneous computing nodes. Moreover, the latency between two nodes from distinct sites is superior to the one between two nodes located on the same site, thus the latency of the communications is also heterogeneous. The computing nodes were selected from the following sites:

- The site of Nancy where each computing unit is equipped with 2 dualcores 1.6 GHz Intel Xeon 5110.
- The site of Sophia Antipolis (Nice) where each computing unit is equipped with 2 processors AMD Opteron 246 2.0GHz
- The site of Orsay (Paris) which is described in the first experiment.

We executed the same application as in the first experiment using JACEP2P and JACEP2P-V2. We have also simulated the volatility of the computing nodes by using the same perturbator script. However in this experiment, the script killed one daemon

on each site each n seconds. The results for this set of experiments are presented in table 4.2. As in the previous experiment, JACEP2P-V2 outperforms JACEP2P, in particular when the environment is highly volatile. Moreover, the crashes' overhead is totally acceptable in JACEP2P-V2. These experiments prove that the modifications implemented in JACEP2P improve its performance on volatile architectures that suffer from high latency between computing nodes. Finally, if we compare the execution times between the first and the second experiment, we notice that the problem is solved a little bit faster on a single cluster than on distributed clusters. This is caused by the higher communications' latency and resources' heterogeneity in the distributed clusters' architectures.

The frequency of the 3 crashes: n	∞	90	60	50
Execution time for JACEP2P	565s	1438s	2008s	2050s
Total number of crashes JACEP2P	0	48	100	122
Execution time for JACEP2P-V2	581s	624s	632s	663s
Total number of crashes JACEP2P-V2	0	19	30	38

Table 4.2: Execution time taken to solve the 3D advection-diffusion problem using 252 machines located on 3 distant sites and while killing one computing node every n seconds at each site

4.4.1.3 Third experiment: the scalability test

To test the scalability of JACEP2P-V2, we tried to solve a large problem using a huge number of nodes. We executed an application that solves the advection diffusion 3D problem. The system contained 512,000,000 components and simulated a 90 seconds time interval with a relative convergence threshold equal to 10^{-11}. We used 392 machines (as a single user) to solve this problem. These computing units were located on the following three sites:

- Site 1 (Nancy), where we used two clusters, in the first one, each machine is equipped with 2 dualcore 1.6 GHz Intel Xeon 5110 and in the second one they are equipped with 2 processors AMD Opteron 246 2.0GHz.
- Site 2 (Sophia), where we used three clusters. In the first one each machine is equipped with 2 processors AMD Opteron 246 2.0GHz, the second one is composed of machines containing 2 dualcore processors AMD Opteron 275 2.2GHz and the third one is formed of machines containing 2 dualcore processors AMD Opteron 2218 2.6GHz.
- Site 3 (Orsay), the computing nodes are equipped with 2 AMD Opteron 246 2.0GHz processors or with 2 AMD Opteron 250 2.4GHz processors.

The total number of cores used in these experiments exceeds 1000 cores. This experiment was only conducted using JACEP2P-V2 because JACEP2P was unable to handle the huge number of computing nodes, due to its centralized mechanisms. The results for this set of experiments are presented in table 4.3.

The frequency of the crashes: n	∞	30	20	10
Execution time	591s	660s	682s	763s
Total number of crashes	0	20	31	67

Table 4.3: Execution time taken to solve the 3D advection-diffusion problem, composed of 512,000,000 components, using 392 machines located on 3 distant sites and while killing a random computing node every n seconds

As shown in table 4.3, JACEP2P-V2 handles well the huge number of computing nodes. Moreover, it resists to nodes crashes. Indeed, if a daemon crashes, the platform replaces the dead daemon with a new one that could be located in a different site or cluster. This new node continues the task from the last checkpoint. Furthermore, the spawners in JACEP2P-V2 share the load between them and their number can be increased in order to support more computing nodes. From an experimental point of view, JACEP2P-V2 is much more scalable than its previous versions and this due to its decentralized mechanisms (global convergence detection, failure detection and backup mechanisms). Finally, we notice that the frequent crashes do not reduce significantly the performance of JACEP2P-V2 (the fault management overhead is acceptable).

4.4.1.4 Fourth experiment: the NAS parallel benchmark CG

aThe NAS parallel benchmark CG consists in a number of steps, each one of them calling the iterative multisplitting method which also uses a sequential conjugate gradient algorithm. When the system converges internally, each computing node executes three reduction operations using data from all the computing nodes. Moreover, we have reduced the matrix bandwidth (illustrated in Figure 4.5) to reduce some of the computing dependencies. The sparse bands contain the nonzero values, but since they are sparse, they also contain components with zero values. For more details on the resolution of this problem, interested readers can refer to [13].

We have implemented this method to test the decentralized global convergence detection mechanism that computes at the same time the reduction functions. In the past when executing this application, the computing nodes had to synchronize three times at the end of each step to compute these reduction functions. With this new mechanism, the computing nodes do not have to synchronize at all. In this experiment, we solved a large problem containing 20,000,000 components and the application computed only six steps with a relative convergence threshold equal to 10^{-14}. We only

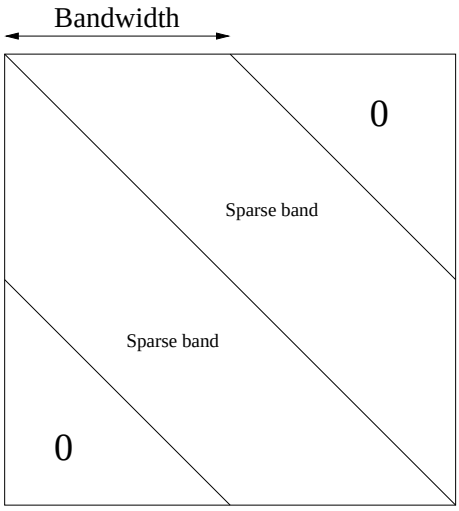

Figure 4.5: The bandwidth of a input matrix in the NAS CG experiment.

used 200 machines (as a single user) because this application consumes a lot of memory and requires machines equipped with more than 2GB of RAM which are not easy to reserve on Grid'5000 due to their small number compared to the the great number of users. These computing units were located on the following three sites:

- Site 1 (Rennes), where we used two clusters, all the machines are equipped with 2 dualcore processors Intel Xeon 5148 LV 2.33GHz.
- Site 2 (Sophia), where we used two clusters, their specifications are presented in the third experiment.
- Site 3 (Bordeaux), each machine is equipped with 2 dualcore processors AMD Opteron 2218 2.6GHz

The frequency of the crashes: n	∞	30	20	10
Execution time	304s	457s	515s	545s
Total number of crashes	0	16	25	52

Table 4.4: Execution time taken to solve the NAS parallel benchmark CG, composed of 20,000,000 components, using 200 machines located on 3 distant sites and while killing a random computing node every n seconds

Table 4.4 presents the execution time taken for solving the problem with different frequencies of crashes. JACEP2P-V2 has computed the reduction functions at the end of each step without any synchronization between computing nodes. We notice that our approach has resisted the high number of nodes' crashes. When a dead node is

replaced by a new one, it retrieves the last checkpoint of the dead node and has to regenerate the random matrix A to which we are computing the eigen value. Therefore, the recovery phase for this experiment is more time consuming than the recovery phase in the advection-diffusion experiment. Nevertheless, the performance of JACEP2P-V2 is not significantly reduced with the huge number of crashes, with more than 50 disconnections it needs less than twice the execution time with 0 disconnections.

4.4.2 Testing of the Waveform Relaxation method

Since our main objective is to solve linear and non linear systems over volatile distributed environments, while developing JACEP2P-V2 we were also working on iterative parallel methods compatible with the asynchronous iteration model. In particular, we concentrated our research on the Waveform Relaxation method. It is an iterative parallel method that solves initial value problems and could be implemented according to the asynchronous iteration model. Most of the work in this domain tested the waveform relaxation method on parallel machines with low latency between processors. So we had to test the Waveform Relaxation method over distributed architectures interconnected via high latency networks before adding failures to the environment. For this reason, we have compared the Waveform Relaxation method to the standard solver for solving non linear systems, PVODE. Since PVODE is implemented in C, the WR method was also implemented in C and the nodes exchanged messages via MPI. The good results obtained from this experiment motivated us to pursue our research in this domain. The next step was to implement the Waveform Relaxation method according to the asynchronous iteration model and testing it over distributed and volatile environments. However, the only platform capable of executing such algorithms over distributed volatile environments is JACEP2P-V2. Therefore, we had to port the code to the Java programming language and implement it according to the JACEP2P-v2's specification. The preliminary tests proved that the application worked fine over distributed and volatile architectures. So afterwards, we compared the WR method to the asynchronous iterative parallel method, Multisplitting-Newton, while solving a 2D advection-diffusion problem over distributed volatile environments. The objective of this comparison is to test the performance of the WR method in such environments and to rate its capacities compared to the Multisplitting-Newton method.

In the next subsections we present in details the experiments undertaken to compare the WR method with the other resolution methods, cited above.

4.4.2.1 Comparison between PVODE and the Waveform Relaxation method

The comparisons between PVODE and the Waveform Relaxation method can be divided into two parts. First of all, we coupled the WR method with the CVODE solver

to solve non linear systems. This method is compared with the PVODE solver while solving the 1D reaction-diffusion problem and the 2D advection-diffusion problem. Afterwards, the WR method is coupled with the Euler method to solve the large scale 2D advection-diffusion problem over the Grid'5000 testbed.

WRVODE: Waveform Relaxation with CVODE

The WR method can be used with different sequential solvers. First we used CVODE which is a very powerful and complex sequential solver. Also, it is an adaptative solver with dynamic stepping. We used the CVODE solver as a black box, so no modifications were done to its core. In order to test the performance of the Waveform Relaxation algorithm with CVODE on large problems, we have conducted two experiments:

1. **One dimensional reaction-diffusion equation**

   In this experiment, we solve the one dimensional reaction-diffusion problem presented in Section 4.3.2. It is well-known that the nature of components ordering can have important influence on the convergence of the Waveform Relaxation algorithms. These are the two natural ways to order the components of the system:

   $$u_1, u_2, ..., u_N, v_1, v_2, .., v_N \quad \text{and} \quad u_1, v_1, u_2, v_2, ..., u_N, v_N$$

   We chose the second ordering because it is shown in [22, 23] that the application that uses this ordering requires less storage space and less execution time than the one adopting the first ordering. In our approach, the system components are split equally between the nodes.

   We implemented two algorithms using the standard ANSI C language to solve this system: the first uses PVODE scheme and the second uses the WRVODE method. We used the predefined solvers available in the Sundials suite [49]. The nodes communicate with each other using the LAM MPI interface, with synchronized exchanges.
   We tested the two applications using a cluster composed of 20 machines, each one contains a 3.0 Ghz processor and up to 1GB RAM. The nodes in this cluster are interconnected via a high speed network with a bandwidth of 1Gbps. However, since we wanted to test this application in a high latency environment, we had to increase the latency of the network. In order to simulate this latency, we have implemented a shell script that delays each packet going through a port to another node a specified amount of time. Using our program a 9ms delay corresponds to 50ms round-trip time when pinging a machine located in the same country and connected to Internet via a DSL connection. Moreover, an 18ms delay corresponds to a 250ms round-trip time when pinging a machine located in

another continent and connected to Internet via a DSL connection. Many real-life tests were conducted to evaluate these delays. It is obvious that the resulting environment is not realistic. However, we realized this first experiment to get an idea of the behavior of the new algorithm because it is easier to experiment on a cluster, before tackling real grid systems.

latency(ms)	0	9	18
PVODE	13m18s	533m3s	931m46s
WRVODE	161m6s	180m12s	199m22s

Table 4.5: Execution time taken by PVODE and WRVODE to solve a 100,000 components 1D reaction-diffusion problem on the time interval [0.555s, 0.65s] with various latencies in communications over 10 homogeneous machines

The results of the experiments are displayed in table 4.5. It presents the time taken by each algorithm to solve a non-stationary differential equation system consisting of 100,000 components. The system is integrated on the time interval $[0.555s, 0.65s]$ which corresponds to a hard part of the problem, using ten machines. The results show that PVODE took just 13 minutes and 18 seconds to solve the system using a very low latency network and the WRVODE algorithm needed about 161 minutes to solve the same problem, which means that PVODE goes ten times faster. However, these results are completely turned around when some latency is added to the communications between the nodes. With only 9ms seconds of latency, PVODE needs around 553 minutes to solve the same system that it had solved in 13 minutes using a fast network. This huge increase in time consumption is due to the great amount of communications between the processors using the PVODE scheme. On the other hand, the WRVODE algorithm is not very affected by the delays, it only needs 180 minutes to solve the same problem. As we increase the latencies we notice that PVODE suffers more and more in terms of time consumption, but, the WRVODE algorithm is not much sensible to these delays. These results allow us to conclude that if the network latency is low enough, PVODE is very adequate to solve large problems. Otherwise the WRVODE algorithm is more robust to solve large problem in a high latency network.

2. **2D advection-diffusion equation** In this experiment, we solve a two dimensional instance of the advection-diffusion problem presented in Section 4.3.1.1. The ODE system is generated from the following 2-species diurnal kinetics advection-diffusion PDE system in a 2 dimensions space [50]:

$$\frac{\partial c(i)}{\partial t} = Kh * (\frac{\partial}{\partial x})^2 c(i) + V * \frac{\partial c(i)}{\partial x} + (\frac{\partial}{\partial y})(Kv(y) * \frac{\partial c(i)}{\partial y} + Ri(c1, c2, t))$$

for i=1,2, where:
$R1(c1, c2, t) = -q1 * c1 * c3 - q2 * c1 * c2 + 2 * q3(t) * c3 + q4(t) * c2$
$R2(c1, c2, t) = q1 * c1 * c3 - q2 * c1 * c2 - q4(t) * c2$
$Kv(y) = Kv0 * e^{(\frac{y}{5})}$
$Kh, V, Kv0, q1, q2$, and $c3$ are constants, and $q3(t)$ and $q4(t)$ vary diurnally. The problem is posed on the square $0 \leq x \leq 20$, $30 \leq y \leq 50$ (all in km), with homogeneous Neumann boundary conditions.

To solve this system, we have also implemented the PVODE and WRVODE algorithms and we have conducted the experiments over the Grid'5000 platform. It is important to notice that Grid'5000 is a fast grid where the latency between the sites is lower than the usual latency of a normal grid. Indeed, the round-trip time between two machines with a standard connection located in France is about $80ms$, on the other hand, the round-trip time between two machines in Grid'5000 on different sites, for example Sophia and Toulouse, is equal to $14.5ms$. To have a latency similar to the one in distributed clusters that communicates over ADSL connections, we applied the perturbator script on the computing nodes while testing the performance of the two algorithms. For these experiments, we used 32 heterogeneous machines of the Grid'5000 platform, distributed over four sites:

- Site of Nancy (cluster Grelon), 8 nodes, 2 CPUs 1.6GHz Intel Xeon 5110 per node with 2 cores per CPU.
- Site of Rennes (cluster Paravent), 8 nodes, 2 CPUs 2.0GHz AMD Opteron 246 per node.
- Site of Lyon (cluster Capricorne), 8 nodes, 2 CPUs 2.0GHz AMD Opteron 246 per node.
- Site of Sophia Antipolis (cluster Helios), 8 nodes, 2 CPUs 2.2GHz AMD Opteron 275 per node with 2 cores per CPU.

latency(ms)	0	9	18
PVODE	8m15s	55m27s	85m54s
WRVODE	9m37s	31m5s	46m37s

Table 4.6: Execution time taken by PVODE and WRVODE to solve the 2D advection-diffusion problem composed of 720,000 components over 32 heterogeneous computing units located over four distant sites.

Table 4.6 displays the execution time taken by the two algorithms to solve a system of 720,000 components on the time interval [0s, 7200s] with various latencies in communications. These experiments confirm the previous ones and show that

the WRVODE algorithm can be applied on different kinds of applications and is very well adapted to high latencies communications. In fact, if the nodes are connected via an ADSL connection, this algorithm outperforms PVODE in terms of execution time. Here, we underline that although we used heterogeneous nodes, the computing process is not affected because we are using synchronized communications between neighboring nodes. In this way we think that it would be very useful to apply the asynchronous iteration model on the WRVODE algorithm. Indeed, this will fasten its execution time because the nodes would not have to synchronize and wait for the dependencies data from its neighboring nodes at each iteration. On the other hand, we had some slow convergence problems with CVODE because it uses an adaptative stepping. Therefore, in the rest of this document, we used the explicit Euler method with a fixed stepping.

The Waveform Relaxation method with Euler

Although we obtained interesting results when coupling the WR method with the adaptative sequential solver CVODE, this algorithm had limited scalability, because we could not control CVODE without changing its core methods, specially the reinitializing method which we use after each iteration. In addition to that, the iterative algorithm had some slow convergence problems when solving large systems because of the dynamic stepping used in CVODE. For all these reasons, we decided to test the WR method with a simpler solver that implements the Euler or Runge-Kutta methods which use fixed steps. In the following paragraph we test the implementation of the WR method with Euler to solve the 2D advection-diffusion equation presented before.

2D advection-diffusion equation with Euler

For this experiment, we used 100 heterogeneous nodes distributed over 4 distant sites:

- Site of Nancy (cluster Grillon), 25 nodes, 2 CPUs 2.0GHz AMD Opteron 246 per node.
- Site of Rennes (cluster Paravent), 25 nodes, 2 CPUs 2.0GHz AMD Opteron 246 per node.
- Site of Toulouse (cluster Violette), 25 nodes, 2 CPUs 2.2GHz AMD Opteron 248 per node.
- Site of Sophia antipolis (cluster Helios), 25 nodes, 2 CPUs 2.2GHz AMD Opteron 275 per node with 2 cores per CPU.

Table 4.7 presents the execution time taken to solve the 2D advection diffusion system using PVODE or Euler with various number of components. These tests prove that Euler with the WR method outperforms the parallel solver PVODE: our algorithm

solves the problem twice faster than PVODE over distributed clusters. This experiment demonstrates that combining Euler with the WR method results in a parallel iterative algorithm very well suited for solving this problem on a distributed grid architecture. Furthermore, it proves that this algorithm is very scalable (we used up to 100 nodes) and that it converges even when using a large number of nodes. For this experiment, we did not use the perturbator script because our method outperformed PVODE without any perturbation of the Grid'5000 platform. So, we predict that on a normal grid, with ADSL connection speed, PVODE will suffer more and our algorithm will be almost unaffected. We also tested the effect of the overlap technique on the convergence

Number of components (millions)	8	12.5	18
PVODE	76m27s	100m55s	135m40s
Euler+WR	35m46s	49m42s	68m14s

Table 4.7: Execution time taken by PVODE and Euler with the Waveform Relaxation method to solve the 2D advection-diffusion problem with various number of components on the time interval [0,7200] using 100 computing units.

of the WR method combined with the Euler method. The results are displayed in figure 4.6, where we present two graphs: the first one shows the execution time and the number of iterations needed to solve the 2D advection-diffusion problem on the time interval [1000s, 1057.6s] (which is equivalent to a window composed of $800 * DT$ with $DT = 0.072s$), while varying the amount of overlapped points. We chose this time interval because we have noticed that the values of the components vary a lot when integrating on this time interval which make it a little bit difficult for the Waveform Relaxation method to converge, specially if no overlap is used. This graph shows the importance of the overlapping concept in accelerating the convergence of the iterative algorithm. It also shows how difficult it is to predict the value of an optimal overlap. In the second graph, we present the execution time needed to solve the same application on a larger time interval which contains both easy and difficult sub-time intervals. With this experiment, we show that a large amount of overlapped points would reduce the benefits of this concept on the overall execution time of the application. As a conclusion we can say that some overlapped points are necessary for increasing the convergence rate of this iterative method, but the optimal percentage of overlapped points is directly related to the problem being solved and on which time interval the application is integrating.

4.4.2.2 Comparison between PVODE, the Multisplitting-Newton and the Waveform Relaxation-Euler method

Since the experiments, comparing PVODE and the WR method, showed that the WR method have a great potential and outperforms PVODE in high latency environments.

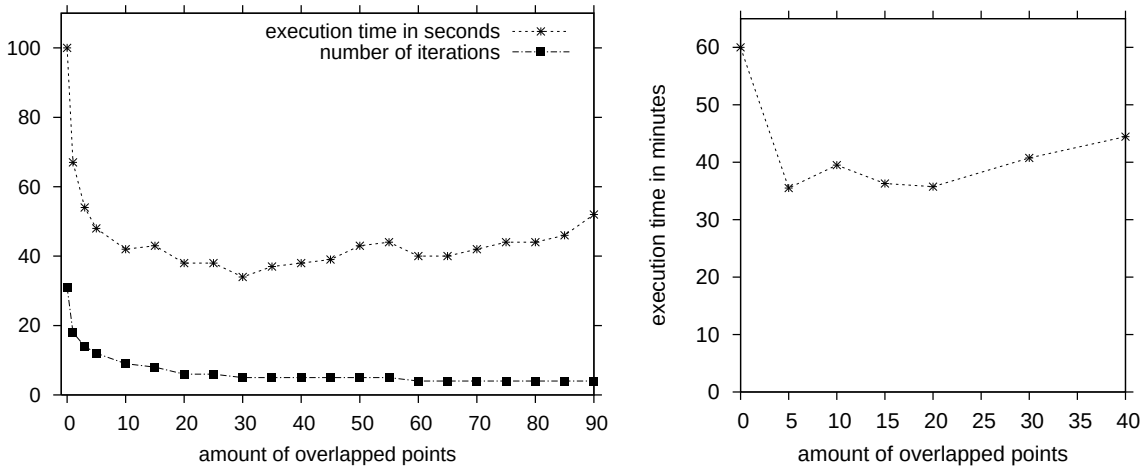

Figure 4.6: The effect of the overlap concept on the convergence of the WR method. The first graph shows the execution time and the number of iterations needed to solve the 2D advection-diffusion problem on the time interval [1000s, 1057.6s] while varying the amount of overlapped points and the second graph shows the execution time taken to solve the same problem on the time interval [1000s, 2000s].

We have continued our studies in this domain and we have ported the WR method to Java and implemented it according to the asynchronous iteration model. The resulting Java code can be executed on JACEP2P-V2 and we can now compare the asynchronous WR method with the Newton-Multisplitting method while solving large problems over distributed volatile environments. We have chose the transport model for this experiment. This problem models the transport of pollutants in shallow seas. The main objective of the simulation is to exhibit the long term evolution trends of the considered ecosystem after pollution. The results of the simulation are chemical species concentrations in time and space. Transport processes of pollutants, salinity, and so on, combined with their bio-chemical interactions can be mathematically formulated as a system of advection-diffusion-reaction equations. It follows an initial boundary value problem for a nonlinear system of PDEs.

Before comparing the performances of the two methods, we have compared their precision to the standard solver, PVODE.

Numerical precision

To test the pertinence of the solutions given by the two methods and since we do not know the exact solution of the problem, we considered the solution obtained by PVODE as the reference solution for this problem. Then, we have evaluated the relative error for each solution according to the solution given by PVODE. The relative error

(r) is computed as follows:

$$r = \frac{max_i(|v_i^{approx} - v_i|)}{max_i(v_i, v_i^{approx})} \quad i = 0, ..., n$$

where $V = (v_0, ..., v_n)$ is the solution vector for PVODE and V^{approx} is the solution vector computed using the WR-Euler method or the Multisplitting-Newton method.

Table 4.8 presents the relative errors obtained when the solutions given by the different methods are compared. We have executed the PVODE method using two precisions: For the first one (respectively second one) the required precision was equal to 10^{-4} (respectively 10^{-10}). For the WR-Euler method and the Multisplitting-Newton method, the respective required precisions were 10^{-11} and 10^{-12}. The first set of relative errors is obtained by comparing the solution vectors for a simulation over the time interval $[0, 200s]$. These experiments show that the solutions obtained by the WR-Euler method and the Multisplitting-Newton method are very close to the solution computed by the PVODE method. Indeed, the relative error between the three solutions is less than 0.1%. Moreover, if we compare the solution vectors of these methods with PVODE's solution that is computed with high precision, we notice that the relative error between the three solutions is less than 0.01%. Therefore, the solutions computed with the two coarse grained methods are more accurate than those computed with PVODE at normal precision. So, we can consider that they are relatively correct.

The second set of relative errors is obtained by comparing the solution vectors of the different methods for a simulation over the time interval $[0, 1000s]$. We have performed these experiments to discover how much the accuracy of the results is reduced when simulating over long time intervals. Using the relative errors presented in table 4.8, we notice that the results are 10 times less accurate than those obtained over a small time interval. This reduction of precision is due to the small errors (like rounding errors) that accumulate over time. These problems are very common in the numerical computing domain, even the PVODE method suffers from them.

Performance

Since we are interested in solving large differential equations in high latency, volatile and heterogeneous environments, we have implemented the Multisplitting-Newton method and the WR-Euler method according to the asynchronous iteration model which is more suitable to such architectures. These applications were executed over JACEP2P-V2 which made them fault tolerant and provided them with all the functionalities required to solve the problem according to the asynchronous iteration

Time interval	Method	MN	WR-Euler	PVODE	PVODE High precision
0 to 200s	WR-Euler	4.27×10^{-5}	0	1.01×10^{-4}	2.32×10^{-5}
	MN	0	4.27×10^{-5}	1.23×10^{-4}	3.89×10^{-5}
	PVODE	1.23×10^{-4}	1.01×10^{-4}	0	1.04×10^{-4}
0 to 1000s	WR-Euler	3.31×10^{-4}	0	2.8×10^{-3}	3.09×10^{-4}
	MN	0	3.31×10^{-4}	2.79×10^{-3}	1.79×10^{-4}
	PVODE	2.79×10^{-3}	2.8×10^{-3}	0	2.78×10^{-3}

Table 4.8: The relative errors obtained when comparing the solutions computed by the different methods: Multisplitting-Newton (MN), Waveform Relaxation coupled with Euler (WR-Euler) and PVODE.

model (like asynchronous messaging and multi-threading).

Two sets of experiments have been realized. In the first one, we only used one site with homogeneous computing units: we used the cluster Grelon located in Nancy. Each computing unit was equipped with two dualcores Intel Xeon 5110 1.6GHz and 2GB RAM. We used 100 nodes to solve a problem containing 16,000,000 components on the time interval $[0, 200s]$. The optimal overlap values were used in these experiments. To simulate a volatile environment, we used a shell script that randomly kills each 60 seconds a daemon that is executing a task. Then, a new daemon is launched on that computing unit. This new daemon is connected to the platform and is ready to execute a new task.

		Method	
Decomposition	Status	Multisplitting-Newton	WR-Euler
10×10	without crashes	21m30s	3m24s
	with crashes	21m56s	3m32s
100×1	without crashes	12m5s	3m4s
	with crashes	12m16s	3m13s
1×100	without crashes	17m9s	4m56s
	with crashes	17m30s	5m5s

Table 4.9: Execution time taken with JACEP2P-V2 to integrate the system on the simulated time interval [0,200s] using 100 computing units located on one site and while killing a random computing node each 60 seconds.

Table 4.9 presents the execution times taken for solving the problem described above using the Multisplitting-Newton method and the WR-Euler method. It also shows the different execution times taken while varying the problem's decomposition

scheme or the volatility of the computing units. The 10×10 decomposition means that the system has been vertically decomposed into 10 subsystems and each subsystem is horizontally decomposed into 10 smaller subsystems. In the same way, the 100×1 (respectively 1×100) decomposition means that the system is horizontally (respectively vertically) decomposed into 100 subsystems. The results show that the WR-Euler method outperforms the Multisplitting-Newton method and solves the problem in a small time period. Although, the Multisplitting-Newton method can integrate the system on a larger time steps than the WR-Euler method (we used a time step equal to 10 seconds for the Multisplitting-Newton method and equal to 0.1 second for the the WR-Euler method), the iterative Multisplitting-Newton method requires solving a linear system at each iteration which takes an important amount of time. In our implementation of this method, we used the GMRES method for solving the linear system on each computing unit. This method is implemented in the Matrix Toolkits for Java package (MTJ) [2] which can benefit from multicores machines because it is multi-threaded. Moreover, the Multisplitting-Newton method requires more iterations than the WR-Euler method to converge to the solution. On the other hand, the WR-Euler method computes directly the solution at each iteration using the Euler formula and the implementation of the windowing concept in this method has drastically accelerated its convergence and reduced the execution time it takes. All these reasons made the WR-Euler method a faster resolution method than the Multisplitting-Newton method for solving complex ODEs. However, we can notice that if we decompose the system in just one dimension, (1×100 or 100×1), the execution time taken by the Multisplitting method is considerably reduced. Indeed, when the system is decomposed in one dimension rather than two, each subsystem has two or less boundaries rather than four. Thus it is less dependent on data received from neighbors and the linear system is easier to solve. Therefore, each iteration is computed faster than in a two dimensional decomposition scheme and the system converges in fewer iterations. In the same way, we notice that if the system is decomposed only horizontally, it is solved faster than when decomposed vertically. This is related directly to the problem being solved. Indeed, in this problem the values of the components aligned horizontally vary more than those aligned vertically. So if the system is only decomposed vertically, each subsystem must executes a lot of iterations to converge to the solution because the values of the boundaries components are evolving at each time step. Moreover, we also notice that both methods resist to the computing units' crashes which demonstrates the benefits of the asynchronous iteration model and the efficiency of the JACEP2P-V2 platform for detecting the crashes and replacing the dead daemons. Finally, we point out that the effect of these crashes is negligible since the execution times are almost unaffected. Therefore, these methods are well adapted to volatile environments.

In the second set of tests, we used distributed computing units located on two dis-

tant sites in order to have a higher latency in the communications between nodes, especially between nodes from distinct sites. Half of the computing units were located in Nancy's site. The architecture of the computing units on this site was described in the previous set of experiments. The second half of the computing units were located in Rennes' site. We used some computing units from the Paraquad cluster. Each one was equipped with two dualcores Intel Xeon 5148 LV 2.33GHz and 4GB RAM. This heterogeneous architecture represents a real distributed cluster environment. We used 100 computing units distributed over these two sites to solve the same problem described above over the same time interval. Moreover, the same shell script was used to simulate the volatility of the computing units.

		Method	
Decomposition	Status	Multisplitting-Newton	WR-Euler
10×10	without crashes	23m2s	3m24s
	with crashes	23m29s	3m40s
100×1	without crashes	11m33s	3m1s
	with crashes	11m51s	3m32s
1×100	without crashes	16m53s	4m44s
	with crashes	17m25s	4m57s

Table 4.10: Execution time taken with JACEP2P-V2 to integrate the system on the simulated time interval [0,200s] using 100 computing units located on two distant sites and while killing a random computing node each 60 seconds.

Table 4.10 presents the execution times taken for solving the problem described above using the Multisplitting-Newton method and the WR-Euler method. It also shows the different execution times taken while varying the problem's decomposition scheme or the volatility of the computing units. The results of this set of experiments show that the WR-Euler method outperforms again the Multisplitting-Newton. In fact, the results are very similar to those of the first experiment. Since both methods are coarse gained and implemented in the asynchronous iteration model, they are almost immune to the high latency of the communications and to the heterogeneity of the computing units. Indeed, the computing units do not have to synchronize at each iteration and they do not have to wait for the reception of data messages from their neighbors to compute the next iteration. So, there is no idle times between iterations and fast computing units do not have to wait for slower ones. Therefore, both methods are well adapted for high latency and heterogeneous environments. It is important to point out that the connection between the two sites has a large bandwidth and if it was smaller, we predict that the performance of the WR-Euler method would be drastically reduced because the data messages exchanged between neighbors when using the WR-Euler method are a lot bigger than those in the Multisplitting-Newton method. For example, if a subsystem has 100 boundaries components with its right-hand side

neighbor and is using the Multisplitting-Newton method to solve its local task, each message sent by this node to its right-hand side neighbor is about $100 \times 8 = 800Bytes$ (each component is a double that requires $8Bytes$ of storage space). For our experiments, this method required around 60 iterations to integrate on one time step, so the total size of the messages sent to that neighbor is around $60 \times 800 = 48KBytes$. On the other hand, if using the WR-Euler method with windows composed of 100 discrete time steps, each message is about $100 \times 800 = 80kBytes$. For our experiments, this method required around 15 iterations to integrate on one window, so the total size of the messages sent to the right-hand side neighbor is around $15 \times 80000 = 1.2MBytes$ which is 25 times bigger than the size of the messages exchanged in the Multisplitting-Newton method.

4.5 Conclusion

In this chapter, we have described some iterative parallel methods for solving linear and nonlinear equation systems. In particular, we have presented the PVODE solver, the Multisplitting-Newton method, the Waveform Relaxation method and the Multisplitting method for linear systems. These methods were used to solve large problems like the advection-diffusion problem in 2D and 3D. All these iterative methods are compatible with the asynchronous iteration model.
Many experiments were conducted over the Grid'5000 platform using the JACEP2P-V2 platform to solve large problems using these different methods. These experiments proved the efficiency, the robustness and the scalability of the JACEP2P-V2 platform. Moreover, the comparison between the JACEP2P and JACEP2P-V2 platforms showed the efficiency of the improvements that we have introduced into JACEP2P-V2. JACEP2P-V2 outperformed JACEP2P in every category. Thanks to these improvements, JACEP2P-V2 is totally fault tolerant, totally decentralized and highly scalable. In particular, it has withstood the frequent crashes and was able to simultaneously harness the computing power of around 400 computing units (more than 1000 cores).
We have also compared the different resolution methods. In particular, we have compared the execution time taken by the PVODE solver and the the Waveform Relaxation method to solve large problems over distributed clusters. These experiments showed that the WR method is more adapted than the PVODE solver to high latency environments because it reduces the penalizing synchronizations between the distant computing units. We have also compared the performance of the Multisplitting-Newton method and the WR-Euler method while executing a large parallel application over volatile environments and according to the asynchronous iteration model. This comparison was conducted using JACEP2P-V2 and it showed that the WR-Euler method outperforms the Multisplitting-Newton method in such conditions but gives similar precisions with the appropriate parameters.

Conclusion and perspectives

Conclusion

The work presented in this document concerns numerical parallel computing over volatile heterogeneous distributed architectures like distributed clusters and volunteer computing architectures. Our main objective was to efficiently solve large complex linear and nonlinear equation systems over the distributed architecture described above. To solve such problems, we were in particular interested by iterative resolution methods that can be parallelized according to the asynchronous iteration model. As shown in the third chapter, this model has many advantages over synchronous models in volatile heterogeneous distributed environments like:

- Eliminating idle time periods between successive iterations.
- Eliminating synchronizations between the computing nodes
- Tolerating the loss of data messages.

To be able to execute parallel applications based on this concept, a dedicated platform that fulfill all the functionalities of this model must be implemented. Therefore, our work was divided into two main parts:

1. Developing a distributed platform dedicated to designing and executing parallel iterative applications based on the asynchronous iteration model over volatile heterogeneous distributed architectures.
2. Optimizing, testing and comparing various numerical parallel resolution methods that are implemented according to the asynchronous iteration model and executed over volatile heterogeneous architectures.

In this way we have presented the JACEP2P-V2 platform which is an evolution of the JACEP2P platform. This platform is dedicated to executing parallel iterative applications based on the asynchronous iteration model. Moreover, JACEP2P-V2 is completely fault tolerant:

- It uses the uncoordinated distributed checkpointing mechanism to save the daemon's data over other daemons.
- It duplicates the spawner by transforming some daemons into spawners which share the load between them.
- It uses a decentralized fault detection mechanism: every group of entities form a circular network where each node sends heartbeat messages to the next node in the circular network. If a node does not receive heartbeat messages from its neighbor for a given time period, it detects that the next node is dead.
- It replaces dead daemons and spawners without interrupting the computing process of other nodes. A new daemon, replacing a dead one, continues the task from the last checkpoint the dead node had made.

All the mechanisms that are implemented in JACEP2P-V2 are totally decentralized, for example:

- Decentralized global convergence detection mechanism.
- Distributed uncoordinated checkpointing mechanism.
- Decentralized fault detection mechanism.

This distribution of tasks makes JACEP2P-V2 highly scalable.

To confirm this approach, we have conducted over the Grid'5000 testbed, a set of experiments that test the performance of JACEP2P-V2 while solving parallel iterative applications based on the asynchronous iteration model over heterogeneous volatile architectures. The experiments proved the efficiency and robustness of JACEP2P-V2. Indeed, This new version has resisted to the frequent crashes and outperformed JACEP2P on local and distant clusters. Furthermore, This platform was highly scalable: it was able to solve large problems using more than 1000 cores distributed over three different sites. On the other hand, we have combined the Waveform Relaxation method with the Euler sequential method and compared its performance to the widely used PVODE algorithm while solving the advection diffusion problem over high latency architectures. The results showed that the WR-Euler method is more adapted to high latency environments than the standard PVODE algorithm. Finally, we have compared the Multisplitting-Newton method to the WR-Euler method, both implemented according to the asynchronous iteration model, while solving the advection-diffusion 2D problem over a heterogeneous volatile environment using JACEP2P-V2. The results show that the WR-Euler method outperforms the Multisplitting-Newton method in terms of time of execution. Although the research work presented in this document is complete, it offers many open tracks to continue researching in this domain.

Perspectives

In the near future, we will pursue our research in two fields: JACEP2P-V2 and parallel iterative methods.

The JACEP2P-V2 platform

This platform is now completely operational and ready to be published as an open source project. Interested researchers can soon use this platform to execute parallel iterative applications based on the asynchronous iteration model. However, JACEP2P-V2 will not be the last version of this platform. Indeed, until now, we did not consider the hardware specifications of the heterogeneous computing nodes that are being used to execute parallel applications. We did not either took into account the characteristics (bandwidth and latency) of the networks relating the different computing units participating to the computation. Therefore, in the near future, we would like to implement a scheduler algorithm into the JACEP2P-V2 platform. It has to assign every task to an appropriate computing unit while taking into consideration the estimated computing power and memory space required to execute that task and the capacities of the selected computing unit. Thus, a problem can be divided into non equal subproblems and fast computing units will execute the big subproblems while the slow computing units will get the small ones. This distribution added to the asynchronous iteration model will allow the platform to tackle more efficiently the heterogeneity problem. Moreover, the scheduler must take into account the geo-localization of all the computing units. This property will allow it to assign heavily interdependent tasks to computing units located in the same area and thus reducing network latencies and communication times. Furthermore, a scheduler can be added to the daemon in order to assign properly the different threads onto the different computing cores and thus increasing the performances of daemons.

Besides conceiving an adapted scheduler, it will be interesting to test the JACEP2P-V2 platform in a real volunteer computing environment. Until now, all the experiments were conducted over the Grid'5000 testbed which is composed of distributed clusters dedicated for research experiments and the volatility of the nodes was simulated by randomly killing a daemon every n seconds. We would like to evaluate the performances of JACEP2P-V2 in a environment composed of heterogeneous volatile public computing units interconnected via high latency networks, like DSL Internet connections.

Finally, it would be interesting to compare the performance of JACEP2P-V2 to other existent platforms. However, to our knowledge there is no other platform able to ex-

ecute parallel iterative applications based on the asynchronous iteration model over volatile distributed architectures. It would not be objective to compare JACEP2P-V2 with other existent platforms that are not dedicated to executing the same tasks. However, we are planning on modifying some platforms in order to make them compatible with the asynchronous iteration model and then we can compare them to JACEP2P-V2.

Parallel iterative methods based on the asynchronous iteration model

In this document, we have presented three methods based on the asynchronous iteration model: the Waveform Relaxation-Euler method, the Multisplitting-Newton method and the Multisplitting method for linear systems. As emphasized before, these methods are well adapted to volatile heterogeneous high latency environments. Moreover, a comparison between WR-Euler and PVODE showed that WR-Euler outperforms the latter in a high latency environment. However, we were not able to compare these two methods in a volatile environment because PVODE uses a standard implementation of MPI which is not fault tolerant. But nowadays, fault tolerant implementations of MPI, like MPICH-V, exist and it would be interesting to compare synchronous and asynchronous resolution methods over volatile environments. The advantages of parallel iterative methods based on the asynchronous model over synchronous parallel iterative methods will be clearly demonstrated in such environments.

It would be also interesting to solve different types of problems using parallel iterative methods based on the asynchronous iteration model in order to prove that this model is compatible with a wide range of numerical problems and not limited to the ones presented in this document.

Finally, since there are no real numerical comparisons between the various parallel iterative resolution methods based on the asynchronous iteration model, it would be interesting to compare the performances and precisions of these methods while solving large problems over volatile heterogeneous environments with high latency networks as we did in the experiment comparing the WR-Euler method to the Multisplitting-Newton method.

Bibliography

[1] CORBA website. http://www.corba.org.

[2] Matrix toolkits for java. http://www.ressim.berlios.de/.

[3] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, 1996.

[4] N. Abdennadher and R. Boesch. Towards a peer-to-peer platform for high performance computing. In *Advances in Grid and Pervasive Computing, Second International Conference, GPC 2007, Paris, France, May 2-4, 2007, Proceedings*, volume 4459 of *Lecture Notes in Computer Science*, pages 412–423. Springer, 2007.

[5] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229–236, British Columbia, Canada, 1995. IEEE Computer Society press.

[6] D. P. Anderson. BOINC: A system for public-resource computing and storage. In Rajkumar Buyya, editor, *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*, pages 4–10. IEEE Computer Society, 2004.

[7] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM (CACM)*, 45(11):56–61, November 2002.

[8] G. Antonoiu and P. K. Srimani. A self-stabilizing leader election algorithm for tree graphs. *Journal of Parallel and Distributed Computing*, 34(2):227–232, 1996.

[9] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[10] J. Bahi, S. Contassot-Vivier, and R. Couturier. Asynchronism for iterative algorithms in global computing environment. In *16th Int. Symposium on High Performance Computing Systems and Applications*, pages 90–97, Moncton, Canada, 2002. IEEE computer society press.

[11] J. Bahi, S. Contassot-Vivier, and R. Couturier. *Parallel Iterative Algorithms: from sequential to grid computing*, volume 1 of *Numerical Analysis & Scientific Computation*. Chapman & Hall/CRC, 2007.

[12] J. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Trans. Parallel Distrib. Syst*, 16(1):4–13, 2005.

[13] J. Bahi, R. Couturier, and D. Laiymani. Comparison of the conjugate gradient of NAS benchmark and of the multisplitting algorithm with the Jace environment. In *IPDPS'08, ACM/IEEE*, Miami, Florida, USA, 2008.

[14] J. Bahi, R. Couturier, K. Mazouzi, and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Elsevier Applied Mathematical Modelling*, 30(7):616–628, 2006.

[15] J. Bahi, R. Couturier, and P. Vuillemin. JaceP2P: an environment for asynchronous computations on peer-to-peer networks. In *Cluster 2006, IEEE Int. Conf. on Cluster Computing*, pages 1–10. IEEE Computer Society Press, 2006.

[16] J. Bahi, S. Domas, and K. Mazouzi. Jace : a java environment for distributed asynchronous iterative computations. In *12th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04*, pages 350–357, Coruna, Spain, February 2004. IEEE computer society press.

[17] J. Bahi, J.-C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Springer Numerical Algorithms*, 15:315–345, 1997.

[18] M. Baker and B. Carpenter. MPJ: A proposed java message passing API and environment for high performance computing. In *Parallel and Distributed Processing Workshop, IPDPS'2000, Cancun, Mexico, May 1-5, 2000, Proceedings*, volume 1800 of *Lecture Notes in Computer Science*, pages 552–559. Springer, 2000.

[19] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs NJ, 1989.

[20] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v: a multiprotocol fault tolerant mpi. *In International Journal of High Performance Computing and Applications*, 20(3):319–333, 2006.

[21] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[22] K. Burrage, C. Dyke, and B. Pohl. On the performance of parallel waveform relaxations for differential systems. *Elsevier Science*, 1995.

[23] K. Burrage and B. Pohl. Implementing an ode code on distributed memory computers. *Computers Math. Applic.*, 28:235–252, 1994.

[24] G. Byrne, D. George, and A. C. Hindmars. Pvode, an ode solver for parallel computers. *Int. J. High Perf. Comput. Apps.*, 13(4):354–365, 1999.

[25] G. Byrne and A. Hindmarsh. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. on Math. Soft.*, 1:71–96, 1975.

[26] G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(12):1213–1225, December 1998.

[27] S. J. Chapin, D. Katramatos, J. F. Karpovich, and A. S. Grimshaw. The legion resource management system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, 13th IPPS/10th SPDP'99 Workshop (5th JSSPP'99)*, volume 1659 of *Lecture Notes in Computer Science (LNCS)*, pages 162–178, San Juan, Puerto Rico, USA, April 1999. Springer-Verlag (Berlin).

[28] S. D. Cohen and A. C. Hindmarsh. Cvode, A stiff/nonstiff ode solver in C, 1996.

[29] T. Lasinski D. Bailey, J. Barton and H. Simon. The NAS parallel benchmarks. Technical Report Report RNR-91-002 revision 2, NASA Ames Research Center, 1991.

[30] L. Dagum and R. Menon. OPENMP : An industry standard API for shared memory programming. *IEEE Computational and Evolutionary Programming*, 5:46–55, 1997.

[31] A. Denis, C. Pérez, and T. Priol. PadicoTM: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19(4):575–585, May 2003.

[32] M. El-Ruby, J. Kenevan, R. Carison, and K. Khalil. Leader election in distributed computing systems. In Naveed A. Sherwani, Elise de Doncker, and John A. Kapenga, editors, *Proceedings of Computing in the 90's*, volume 507 of *LNCS*, pages 350–356, Berlin, Germany, October 1991. Springer.

[33] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Computers*, 41(5):526–531, 1992.

[34] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed process in manetho. In *the twenty-second International Symposium on Fault-Tolerant Computing*, pages 18–27. IEEE Computer Society, Boston, USA, 1992.

[35] G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb: A generic global computing system. In *International Symposium on Cluster Computing and the Grid*, pages 582–587. IEEE Computer Society, 2001.

[36] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *DOA*, pages 132–141, 1999.

[37] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[38] A. Frommer and D. B. Szyld. Asynchronous iterations with flexible communication for linear systems. *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 10:421–429, 1998.

[39] A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of computational and applied mathematics*, 23:201–216, 2000.

[40] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, USA, 1994.

[41] S. Genaud and C. Rattanapoka. Fault management in P2P-MPI. In Christophe Cérin and Kuan-Ching Li, editors, *Advances in Grid and Pervasive Computing, Second International Conference, GPC 2007, Paris, France, May 2-4, 2007, Proceedings*, volume 4459 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2007.

[42] S. Genaud and C. Rattanapoka. Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In *5th High Performance Grid Computing International Workshop, IPDPS conference proceedings*. IEEE, April 2008.

[43] L. Gong. Jxta: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[44] Grid'5000. http://www.grid5000.fr.

[45] A. S. Grimshaw, A. Natrajan, M. A. Humphrey, M. J. Lewis, A. Nguyen-Tuong, J. F. Karpovich, M. M. Morgan, and A. J. Ferrari. *Grid Computing : Making the Global Infra-structure a Reality*, chapter 10, From Legion to Avaki : The Persistence of Vision, pages 265–298. John Wiley, March 2003.

[46] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[47] E. Hairer, S. P. Norsett, and G. Wanner. Solving ordinary differential equations, nonstiff problems. *Springer Verlag, New York*, 1, 1987.

[48] A. C. Hindmarsh. User documentation for PVODE, an ODE solver for parallel computers, 2007.

[49] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005.

[50] A. C. Hindmarsh and R. Serban. Example programs for cvode v2.5.0. 2006.

[51] R. Couturier J. Bahi, J. Charr and D. Laiymani. A parallel algorithm to solve large stiff ode systems on grid systems. *HETEROPAR'07, Cluster*, pages 534–541, 2007.

[52] K. Jackson and R. Sacks-Davis. An alternative implementation of variable step-size multistep formulas for stiff odes. *ACM Trans. on Math. Soft.*, 6:295–318, 1980.

[53] R. Jeltsch and B. Pohl. Waveform relaxation with overlapping splittings. *Siam J. Sci. Comput.*, 16, 1995.

[54] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology, 2009.

[55] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli. The waveform relaxation method for time domain analyzes of large scale integrated circuits. *IEEE Trans. CAD*, 1:131–145, 1982.

[56] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computer Systems*, pages 104–111, Washington, D.C., June 1988. IEEE Press.

[57] A. Marowka. Extending openMP for task parallelism. *Parallel Processing Letters*, 13(3), 2003.

[58] MPI Forum. MPI: A message passing interface. In *Proceedings of Supercomputing '93*, pages 878–883, Portland, OR, November 1993. IEEE CS Press.

[59] OAR. http://oar.imag.fr/about/overview.html.

[60] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *USENIX Winter*, pages 213–224, 1995.

[61] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection. In *Service,Ž01d Proc. Conf. Middleware*, pages 55–70, 1998.

[62] Roadrunner. http://www.lanl.gov/roadrunner/.

[63] S. A. Savari and D. P. Bertsekas. Finite termination of asynchronous iterative algorithms. *Parallel Computing*, 22(1):39–56, 1996.

[64] W. E. Schiesser. *The Numerical Method of Lines*. Academic Press, 1991.

[65] M. El Tarazi. Some convergence results for asynchronous algorithms. *Numerische Mathematik*, 39:325–340, 1982.

[66] J. G. Verwer, J. G. Blom, and W. Hundsdorfer. An implicit-explicit approach for atmospheric transport-chemistry problems. *Applied Numerical Mathematics: Transactions of IMACS*, 20(1–2):191–209, February 1996.

[67] J. White, F. Odeh, A. Ruehli, and A. S. Vincentelli. Waveform relaxation: Theory and practice. *Trans. of Soc. for Computer Simulation*, 2:95–133, 1985.

[68] Y.Zhang, V. Tipparaju, J. Nieplocha, and S. Hariri. Parallelization of the nas conjugate gradient benchmark using the global arrays shared memory programming model. In *Proc. of 19th International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2005.

[69] H. Zhang. A note on windowing for the waveform relaxation. Technical Report NASA CR-194907 ICASE Report No. 94-28, Institute for Computer Applications in Science and Engineering,NASA Langley Research Center Hampton, April 94.

List of publications

International journals

[1] R. Couturier J. Bahi, J. Charr and D. Laiymani. A parallel algorithm to solve large stiff ODE systems on grid systems. *The International Journal of High Performance Computing Applications*. In press.

[2] R. Couturier J. Charr and D. Laiymani. A decentralized and fault tolerant convergence detection algorithm for asynchronous iterative algorithms. *The Journal of Supercomputing*. In press.

International Conferences

[3] Jacques M. Bahi, Jean-Claude Charr, Raphaël Couturier, and David Laiymani. A parallel algorithm to solve large stiff ODE systems on grid systems. In *CLUSTER*, pages 534–541, Austin Tx, USA, 2007. IEEE.

[4] R. Couturier J. Charr and D. Laiymani. JACEP2P-V2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures. In *Advances in Grid and Pervasive Computing, 4th International Conference, GPC 2009, Geneva, Switzerland, May 4-8, 2009. Proceedings*, volume 5529 of *Lecture Notes in Computer Science*, pages 446–458. Springer, 2009.

[5] R. Couturier J. Charr and D. Laiymani. Parallel numerical asynchronous iterative algorithms: Large scale experimentations. In *the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, MAY 2009. IEEE.

National conferences

[6] R. Couturier J. Charr and D. Laiymani. Un algorithme décentralisé et asynchrone pour la détection de la convergence dans un environnement volatil. In *Rencontres Francophones du Parallélisme, (RenPar)*, Fribourg, Switzerland, Feb 2008.

Master thesis

[7] Jean claude Charr. Visualisation et supervision d'ordinateurs et de systèmes informatiques en environnement virtuel, 2006.

Résumé

Avec l'émergence de nouvelles architectures distribuées, comme les grappes de calcul distantes et les architectures de calcul volontaire, il apparaît important de définir des algorithmes et des intergiciels bien adaptés à ces architectures. En effet, l'utilisation de ces plate-formes introduit plusieurs nouvelles contraintes par rapport à un contexte de grappes locales homogènes : hétérogénéité des machines, hétérogénéité des réseaux, volatilité des noeuds de calcul, etc. Dans ce contexte, plusieurs travaux montrent que pour les algorithmes itératifs il peut être préférable d'utiliser les algorithmes IACA (Itérations Asynchrones avec Communications Asynchrones) pour lesquels les communications sont recouvertes par du calcul et la perte des messages de données est tolérée.
Les travaux présentés dans cette thèse concernent la conception et la mise en oeuvre d'une plate-forme dédiée à l'exécution d'algorithmes IACA sur des architectures distribuées, hétérogènes et volatiles. Cette plate-forme, JACEP2P-V2, est tolérante aux pannes et décentralisée. Elle offre un mécanisme de communications asynchrones et un mécanisme de détection de la convergence globale adapté aux caractéristiques des algorithmes IACA.
De plus, nous reportons des expérimentations sur grappes hétérogènes volatiles et distantes afin de tester l'efficacité et la robustesse de notre plate-forme. Les résultats obtenus, avec plus de 1000 coeurs de calculs, sont très encourageants et montrent que JACEP2P-V2 est extensible et performante. Nous terminons ce document par la présentation d'une étude comparative de plusieurs méthodes de résolutions de systèmes non linéaires (comme la multi-décomposition et la relaxation d'ondes) implémentées avec JACEP2P-V2.

Mots clefs: algorithmes parallèles itératifs asynchrones, plateforme décentralisée, tolérance aux pannes, systèmes linéaires et non linéaires.

Abstract

With the emergence of new distributed architectures, such as distributed clusters and volunteer computing architectures, it seems important to design algorithms and middlewares that are well adapted to these architectures. Indeed, when using these architectures, developers are faced with many new constraints that they do not encounter when using local clusters, like the heterogeneity of the machines and the networks that interconnect them, the volatility of the computing nodes, etc. In this context, many research works show that for iterative methods, it is preferred to use the AIAC (Asynchronous iterations with Asynchronous Communications) model where the communications are overlapped by the computations and the loss of data messages is tolerated.
The research work, presented in this document, concerns the design and the implementation of a platform dedicated to executing AIAC algorithms over distributed heterogeneous volatile architectures. This platform, JACEP2P-V2, is fault tolerant and decentralized. It offers an asynchronous communication mechanism and a global convergence detection mechanism well adapted to the characteristics of AIAC algorithms.
Moreover, we present many experiments that we have conducted over volatile distributed heterogeneous architectures using JACEP2P-V2 in order to test the efficiency and robustness of our platform. The experiments' results are very encouraging and prove that JACEP2P-V2 is scalable and powerful. We end this document with a comparative study of many methods that solve nonlinear systems (such as Multisplitting and Waveform relaxation) and are implemented according to JACEP2P-V2's API.

Key words: parallel asynchronous iterative algorithms, decentralized platform, fault tolerance, linear and nonlinear systems.