N° d'ordre 135
Année 2005

HABILITATION À DIRIGER DES RECHERCHES

Présentée à

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

Algorithmes itératifs asynchrones sur grappes distantes et équilibrage de charge sur topologies dynamiques

par
Raphaël COUTURIER

Soutenue le 25 Novembre 2005 devant la commission d'examen :

Rapporteurs	Franck Cappello	Directeur de Recherche INRIA Futurs, Orsay
	Michel Daydé	Professeur à l'Institut National Polytechnique de Toulouse
	Pierre Manneback	Professeur à la Faculté Polytechnique de Mons, Belgique
	Serge Petiton	Professeur à l'Université des Sciences et Technologies de Lille
Examinateurs	Jacques Bahi	Professeur à l'Université de Franche-Comté
	Laurent Philippe	Professeur à l'Université de Franche-Comté

Remerciements

Je tiens tout d'abord à remercier les membres du jury Jacques Bahi, Franck Cappello, Michel Daydé, Pierre Manneback, Serge Petiton et Laurent Philippe pour l'intérêt qu'ils ont porté à ces travaux. J'adresse un remerciement particulier à Franck Cappello, Michel Daydé, Pierre Manneback et Serge Petiton pour avoir accepté de rapporter ce document.

Merci à toutes les personnes de l'équipe AND. Jacques Bahi pour son énergie, ses conseils et pour m'avoir encouragé depuis mon arrivée au LIFC. Les doctorants que j'ai co-encadrés avec Jacques, Flavien Vernier et Philippe Vuillemin. Sylvain Contassot-Vivier pour avoir collaboré sur de nombreux travaux. Une partie des résultats présentés dans ce document n'aurait pas été possible sans vous. Je tiens aussi à remercier Jean-Luc Anthoine, Stéphane Domas, Arnaud Giersch, David Laiymani, Ahmed Mostefaoui, Michel Salomon et les doctorants de l'équipe Amine Abbas, Abdallah Makhoul, Kamel Mazouzi et Marc Sauget pour tous les moments passés ensemble.

Merci enfin à ma famille et mes proches qui m'ont soutenu et aidé depuis le début.

Table des matières

Introduction **1**

I Équilibrage de charge **5**

1 Équilibrage de charge sur réseaux statiques **7**
1.1 Introduction 7
1.2 Algorithmes d'équilibrage de premier ordre 8
1.2.1 Diffusion 8
1.2.2 Algorithme de dimension exchange généralisé : GDE 10
1.2.3 Diffusion relaxée 11
1.2.4 Détermination du paramètre de relaxation β optimal 12
1.3 Conclusion 13

2 Réseaux dynamiques **15**
2.1 Algorithmes de diffusion de premier ordre 16
2.1.1 Modélisation de l'algorithme 16
2.1.2 Conditions de convergence sur réseaux dynamiques 17
2.2 Generalized Adaptative Exchange 17
2.2.1 Modélisation de l'algorithme 18
2.2.2 Conditions de convergence 19
2.3 Algorithme de diffusion relaxée 19
2.3.1 Modélisation de l'algorithme 19
2.3.2 Paramètre de relaxation β optimal 20
2.3.3 Conditions et preuve de convergence 20
2.4 Conclusion 20

II Mise en œuvre efficace d'algorithmes itératifs asynchrones **23**

3 Méthodes itératives synchrones et asynchrones **25**
3.1 Introduction 25
3.2 Algorithmes itératifs 26

3.2.1 Algorithmes itératifs séquentiels 26
3.2.2 Algorithmes itératifs parallèles synchrones 27
3.2.3 Algorithmes itératifs parallèles asynchrones 29
3.3 Classification des algorithmes 31
3.4 Variantes des algorithmes IACA 33
3.5 Conclusion 34

4 Conception d'algorithmes IACA efficaces 37
4.1 Introduction 37
4.2 Inadéquation des environnements mono-threadés 38
4.3 Algorithme IACA dans un environnement multithreadé 39
4.4 Détection de convergence 40
4.4.1 Approche centralisée 41
4.4.2 Approche décentralisée 42
4.5 Conclusion 43

5 Environnements de programmation 45
5.1 MPI/Mad 46
5.1.1 MPI/Mad pour les IACA 47
5.2 PM2 47
5.2.1 PM2 pour les IACA 48
5.3 Corba OmniOrb 4.0 48
5.3.1 OmniOrb 4.0 pour les IACA 48
5.3.2 JACE 49
5.4 Avantages et inconvénients des environnements 50
5.5 Conclusion 52

6 Multidécomposition pour système linéaire 53
6.1 Introduction 53
6.2 Algorithme multisplitting-LU 54
6.3 Expérimentations 56
6.3.1 Expérimentations avec une grappe homogène locale 59
6.3.2 Expérimentations sur grappes hétérogènes (locale et distante) . . 59
6.3.3 Impact de la charge du réseau 61
6.3.4 Influence du recouvrement 61
6.4 Conclusion 63

7 Multidécomposition pour système non linéaire 65
7.1 Introduction 65
7.2 Algorithme de multisplitting-Newton 67
7.3 Expérimentations sur deux problèmes 71
7.3.1 Résolution d'un système de transport de composés polluants . . 72

7.3.2 Résolution d'un système d'équations d'ondes non linéaires . . . 76
7.4 Conclusion . . . 82

8 Équilibrage de charge pour les algorithmes IACA 85
8.1 Introduction . . . 85
8.2 Description de l'algorithme . . . 86
8.3 Choix de l'estimateur de la charge . . . 90
8.4 Illustration sur un exemple . . . 93
8.4.1 Présentation du problème . . . 93
8.4.2 Expérimentations . . . 94
8.5 Conclusion . . . 96

Conclusion - Perspectives 97

Bibliographie 101

Liste des publications 111

Table des figures

3.1 Algorithme ISCS . . . 32
3.2 Algorithme ISCA . . . 32
3.3 Algorithme IACA avec communications rigides . . . 33
3.4 Algorithme IACA avec communications flexibles . . . 34
3.5 Algorithme IACA avec communications semi-flexibles . . . 35

6.1 Décomposition de la matrice et des vecteurs . . . 55
6.2 Impacts du recouvrement avec grappe3 et la matrice générée 100000 . . 62

7.1 Décomposition de la Jacobienne du vecteur et de la fonction . . . 69
7.2 Influence du recouvrement . . . 74
7.3 Influence des communications perturbantes . . . 75
7.4 Temps d'exécution des versions synchrones et asynchrones sur une grille de taille 700*700 sans recouvrement. . . . 78
7.5 Influence du recouvrement sur une grille de taille 700*700 avec 16 processeurs homogènes. . . . 79
7.6 Influence du recouvrement sur une grille de taille 700*700 avec une grappe locale hétérogène de 40 processeurs. . . . 79
7.7 Influence du recouvrement sur une grille de taille 1000*1000 avec une grappe locale hétérogène de 40 processeurs. . . . 80
7.8 Influence du recouvrement sur une grille de taille 1300*1300 avec une grappe locale hétérogène de 40 processeurs. . . . 80
7.9 Influence du recouvrement sur une grille de taille 1000*1000 avec une grappe distante hétérogène de 16 processeurs. . . . 81
7.10 Influence des communications perturbantes sur une grille de taille 1000*1000 avec une grappe distante hétérogène de 16 processeurs. 82

8.1 Transfert de charge basé sur le résidu entre 2 processeurs . . . 92
8.2 Temps d'exécution (en secondes) avec les deux estimateurs de charge . . 94
8.3 Temps d'exécution sur une grappe homogène . . . 95

Introduction

La modélisation de systèmes complexes a toujours représenté un challenge pour la recherche. Les problèmes résultants, généralement difficiles à résoudre, nécessitent souvent l'utilisation simultanée de plusieurs ressources de calculs pour deux raisons. L'une d'elle est de diminuer les temps d'exécution. L'autre est de pouvoir traiter des problèmes de grande taille dont la résolution n'est pas possible avec la mémoire d'une seule machine. Dans certains cas, ces deux raisons sont invoquées.

Pour utiliser simultanément plusieurs ressources de calculs, deux possibilités sont alors envisageables. La première consiste à utiliser une machine parallèle constituée de processeurs homogènes et d'un réseau haute performance. La seconde solution est d'utiliser des machines standards reliées par un réseau (généralement moins rapide que celui d'une machine parallèle). Si les machines sont localisées sur le même site, on parle alors de grappe ou cluster. Dans ce cas, les machines peuvent être homogènes ou dans certains cas hétérogènes. Le réseau est généralement homogène. Afin de pouvoir traiter des problèmes de grandes tailles, il est nécessaire de posséder un nombre conséquent de machines. La meilleure solution est d'utiliser des machines situées sur des sites distants. On parle alors de grappes distantes ou de grilles de calcul. Ainsi le nombre de machines disponibles est en théorie illimitée. Dans ce contexte, les machines et les réseaux sont hétérogènes. C'est cet environnement de grilles de calcul ou de grappes distantes que j'ai étudié depuis mon arrivée à Belfort au sein du LIFC.

La distance entre les machines influe sur plusieurs paramètres qui ont des conséquences importantes. En effet, la latence et le débit des communications peuvent subir des variations importantes dans le contexte des grappes distantes. D'autre part, les algorithmes centralisés ne sont pas adaptés à ce contexte en raison du goulot d'étranglement qu'ils constituent. De plus, le passage à l'échelle se trouve limité par le nœud centralisateur. Finalement, les synchronisations fréquentes sont à éviter à cause de la distance et en raison du nombre potentiellement important de machines.

La conception d'algorithmes parallèles à gros grain permet de réduire sensiblement le nombre de synchronisations. Le principe est d'effectuer une longue partie de calcul sans communications bloquantes et, ensuite seulement, de réaliser des communications bloquantes et éventuellement des synchronisations. Une autre solution consiste à utiliser des algorithmes parallèles asynchrones, c'est-à-dire des algorithmes sans synchronisation. Au premier abord, il peut paraître surprenant qu'un algorithme parallèle

synchrone puisse être transformé en un algorithme parallèle asynchrone et produire le même résultat.

Les algorithmes numériques peuvent être décomposés en deux grandes classes : les algorithmes directs et les algorithmes itératifs. Pour ces deux classes, il existe des versions parallèles de la plupart des algorithmes. Les algorithmes directs donnent la solution d'un problème en effectuant une seule grosse étape de calcul. La solution est correcte modulo les erreurs d'arrondi. La méthode directe la plus utilisée pour résoudre un système linéaire est certainement LU. Les algorithmes itératifs, quant à eux, convergent vers la solution d'un problème par itérations successives. Le calcul de l'itération courante utilise le résultat de l'itération précédente et ainsi de suite. Le nombre d'itérations dépend de la précision choisie. Pour certains problèmes, on ne connaît pas d'algorithme direct pour les résoudre : c'est le cas, par exemple, de la recherche des racines d'un polynôme (de degré supérieur à 5), de la résolution de problème non linéaire, d'algorithme d'optimisation. La plupart des versions parallèles d'algorithmes itératifs utilisent des synchronisations à chaque itération afin de détecter la convergence. On parle alors d'algorithmes parallèles itératifs synchrones ou encore, par abus de langage, d'algorithmes synchrones.

Cependant, certains algorithmes itératifs parallèles supportent l'asynchronisme. Les algorithmes itératifs parallèles asynchrones ont pour particularité de supprimer toutes les synchronisations entre les nœuds de calcul. Par abus de langage ces algorithmes sont appelés algorithmes asynchrones. Leur convergence est assurée après une étude qu'il est impératif de réaliser. Les conséquences de l'asynchronisme sont nombreuses :

- Les processeurs calculent leurs itérations à leur propre vitesse (sans se soucier de la vitesse de leurs voisins). Donc il n'y a plus de temps d'attente dû aux synchronisations ou aux communications bloquantes.
- Les algorithmes asynchrones supportent les pertes de messages (partielles ou totales).
- Le nombre d'itérations qu'un processeur doit effectuer pour atteindre la convergence est généralement plus important qu'en version synchrone. Par contre, vu qu'il n'y a plus de temps d'attente, le temps d'exécution peut être réduit.
- Les algorithmes asynchrones sont bien adaptés à l'hétérogénéité des machines et des réseaux puisque la différence de vitesse des processeurs et des communications est relativement bien supportée.
- Au niveau implantation, certaines parties d'un algorithme asynchrone doivent être programmées différemment qu'en version synchrone. Les deux parties principales concernent la gestion des communications et la détection de convergence.

Pour les raisons évoquées précédemment, il paraît clair que la classe des algorithmes asynchrones est bien adaptée au contexte du calcul sur grille. Cependant, pour obtenir de bonnes performances, il faut concevoir ces algorithmes soigneusement. Notamment, il parait nécessaire de veiller à développer des algorithmes à gros grains et

décentralisés. La seconde partie de ce document s'attache à présenter nos résultats sur ce sujet.

Dans le contexte du calcul distribué l'équilibrage de charge a pour fonction de répartir au mieux la quantité de travail dont chaque processeur est responsable, afin d'accélérer les traitements. Il existe de nombreux algorithmes pour effectuer cette tâche. Les algorithmes centralisés peuvent répartir la charge sur les processeurs assez facilement. Mais ils ont l'inconvénient de centraliser toute l'information sur un nœud. Afin d'éliminer cette contrainte, des algorithmes décentralisés ont été proposés. Dans ce cas, chaque processeur échange de la charge uniquement avec ses voisins, sans connaissance globale de l'ensemble des nœuds. Ces algorithmes sont itératifs et ils convergent vers une distribution uniforme de la charge. Initialement, ces algorithmes ont été développés pour des réseaux statiques, c'est-à-dire pour un cadre dans lequel la topologie de communication des processeurs n'est pas modifiée au cours du temps. Avec l'avènement des réseaux longues distances, il semble important de considérer que la topologie du réseau n'est plus statique. En effet, des liens peuvent être surchargés ou même coupés temporairement, en raison d'une panne par exemple : on parle alors de réseaux dynamiques. L'intérêt des algorithmes d'équilibrage de charge pour topologies dynamiques est donc de réduire les temps d'exécution, même en présence de coupures de liens, tout en garantissant que l'algorithme converge vers l'équilibre. La première partie de ce document présente les nouveaux algorithmes que nous avons développés pour équilibrer la charge sur des réseaux dynamiques.

Plan du document

Ce document est découpé en deux parties. La première aborde l'équilibrage de charge sur réseaux dynamiques en deux chapitres. La seconde présente les algorithmes itératifs asynchrones pour grappes distantes, sous la forme de six chapitres.

Le premier chapitre présente les algorithmes d'équilibrage de charge distribués sur lesquels sont basés nos travaux. Il s'agit de l'algorithme de *diffusion* et de l'algorithme *dimension exchange*. Nous avons défini un nouvel algorithme pour accélérer l'algorithme de diffusion par une méthode de relaxation. Nous l'appelons *diffusion relaxée*.

Le second chapitre expose les algorithmes d'équilibrage de charge que nous avons adaptés aux réseaux dynamiques, à partir des algorithmes présentés dans le chapitre précédent. Ces algorithmes supportent des modifications de la topologie de communication au cours du temps. Une hypothèse réaliste sur la topologie du graphe nous permet d'établir la convergence des algorithmes.

Le troisième chapitre débute la seconde partie, il introduit les algorithmes itératifs parallèles synchrones et asynchrones. Pour chacun d'eux, nous donnons le modèle d'algorithme. Une classification des algorithmes itératifs permet de saisir les particularités de chacun et de leur donner des noms plus précis. Les algorithmes asynchrones

sont appelés *IACA* pour algorithmes à Itérations Asynchrones avec Communications Asynchrones.

Dans le quatrième chapitre, nous expliquons comment développer efficacement des algorithmes IACA. Deux points sont essentiels : disposer d'une bibliothèque de communication *multithreadée* afin de dissocier le calcul des communications et mettre en œuvre un algorithme efficace de détection de convergence, si possible décentralisé.

Le cinquième chapitre illustre quatre environnements de programmation que nous avons utilisés pour développer des algorithmes IACA. Il illustre que ces algorithmes ne sont pas trop liés aux environnements de programmation et qu'il est relativement aisé de les développer. Nous précisons les avantages et les inconvénients de chacun de ces environnements.

Le sixième chapitre introduit une méthode originale qui permet de résoudre un système linéaire avec un algorithme parallèle itératif en utilisant une méthode de résolution directe sur chaque processeur. Cet algorithme repose sur la méthode de multidécomposition (ou multisplitting) et il fonctionne en mode synchrone et asynchrone. À travers différentes expérimentations, nous montrons le comportement de notre algorithme sur plusieurs grappes de processeurs.

Le septième chapitre propose une méthode pour paralléliser la résolution d'un système non-linéaire. Il repose également sur l'utilisation des méthodes de multidécomposition. La partie non-linéaire est résolue avec la méthode de Newton, l'algorithme résultant est donc appelé multisplitting-Newton. Nous le mettons en pratique sur deux applications.

Le huitième chapitre rapproche les deux parties de ce document en montrant comment coupler un algorithme d'équilibrage de charge à un algorithme itératif asynchrone. Nous définissons un estimateur original de la charge qui permet d'accélérer l'exécution d'un algorithme IACA. Une expérimentation illustre l'avantage de ce couplage.

Finalement, nous dressons une conclusion de nos travaux et nous présentons des perspectives liées aux résultats présentés dans ce manuscrit.

Première partie

Équilibrage de charge

Chapitre 1

Équilibrage de charge sur réseaux statiques

1.1 Introduction

Dès qu'on utilise plusieurs ressources de calcul, il est nécessaire de répartir la charge entre ces ressources. Une problématique importante du calcul distribué est de répartir la charge des processeurs de manière à ce que le calcul s'exécute le plus rapidement possible. Il existe de nombreuses méthodes que l'on peut regrouper en deux grandes catégories : les algorithmes centralisés d'équilibrage de charge, et à l'inverse, les algorithmes décentralisés.

Les algorithmes centralisés reposent sur des techniques plus faciles à mettre en œuvre que les algorithmes décentralisés. En effet, la centralisation permet d'avoir une information globale sur l'ensemble des processeurs et simplifie, dans la pratique, la répartition de charge. Cependant le choix de la charge à déplacer est un autre problème à part entière, puisqu'il dépend du calcul en cours et donc des communications qu'il engendre. Comme beaucoup d'algorithmes centralisés, cette catégorie d'algorithmes d'équilibrage de charge possède le défaut d'être tributaire du nœud centralisateur qui devient souvent le goulot d'étranglement.

Les algorithmes décentralisés d'équilibrage de charge ont l'avantage de ne pas regrouper l'information sur un seul nœud. De ce fait, il est préférable de les utiliser pour les systèmes à grande échelle. Leur principe de fonctionnement peut sembler simple : un nœud connaît uniquement les caractéristiques de ses voisins proches. Le système global s'équilibre petit à petit en procédant par plusieurs itérations. Ce type d'équilibrage a été introduit par Cybenko [50].

À notre connaissance, jusqu'à maintenant, les algorithmes d'équilibrage de charge ont été étudiés uniquement pour des topologies de réseaux statiques : ces topologies n'évoluent pas avec le temps.

Dans la suite de ce chapitre, nous présentons les deux algorithmes de premier ordre

rencontrés le plus fréquemment dans la littérature : la diffusion et GDE. Ces algorithmes de premier ordre utilisent seulement la charge de l'itération précédente afin d'équilibrer la charge courante. Nous présentons ensuite une variante que nous proposons ; cette variante appelée diffusion relaxée consiste à utiliser un paramètre de relaxation afin d'accélérer la convergence.

1.2 Algorithmes d'équilibrage de premier ordre

Ces algorithmes permettent d'équilibrer la charge courante à partir de la charge précédente. Après un certain nombre d'itérations l'algorithme converge vers une répartition uniforme de la charge. Chaque nœud "diffuse" une partie de sa charge à ses voisins. Dans [50], Cybenko définit cet algorithme dans le contexte des réseaux statiques et homogènes. Son algorithme est néanmoins adapté au contexte de charge dynamique, c'est-à-dire évoluant au cours du temps.

1.2.1 Diffusion

Afin de comprendre le fonctionnement de l'algorithme de diffusion il faut définir le réseau que l'on souhaite utiliser. Celui-ci doit être connecté, son graphe associé ne doit pas être bipartite et les liens de communications sont supposés bidirectionnels. Un graphe bipartite est un graphe dans lequel l'ensemble des sommets peut être décomposé en deux sous-ensembles disjoints, tels que deux sommets du même sous-ensembles ne soient pas adjacents.

Classiquement, la topologie d'un réseau est représentée par un graphe $G = (V, E)$ avec V l'ensemble des nœuds et E l'ensemble des arcs du réseau. Notons que E est un sous-ensemble de $V \times V$. Chaque processeur du réseau est représenté par un nœud dans le graphe, et tout lien de communication entre deux processeurs i et j est représenté par l'arc (i, j) appartenant à E. Par définition, chaque nœud est numéroté entre 1 et n, ainsi nous avons $|V| = n$ et nous posons que $|E| = m$.

Dans la suite des explications, nous nous restreignons au contexte dans lequel la charge est statique, afin de simplifier les formules. Le passage au contexte dynamique est relativement aisé dans ce cas. Pour plus de détail, il faut consulter par exemple [50]. De même nous faisons le choix de présenter uniquement les algorithmes pour machines homogènes car le passage d'une situation homogène à une situation hétérogène complique les formules mais ne pose pas de difficulté majeure. Pour plus de détail, consulter [58].

Afin d'exprimer l'algorithme de diffusion, nous notons par $w_i^{(t)}$ la charge du processeur i à l'instant t. Nous considérons dans un premier temps, que cette charge $w_i^{(t)}$ est infiniment divisible. Avec ces définitions, l'algorithme de diffusion s'écrit de la manière

suivante :

$$w_i^{(t+1)} = w_i^{(t)} + \sum_j \alpha_{ij}(w_j^{(t)} - w_i^{(t)}), \tag{1.1}$$

où α_{ij} est une constante positive.

Nous pouvons remarquer que la charge de travail échangé $l_{ij}^{(t)}$ entre deux processeurs i et j voisins est donnée par :

$$l_{ij}^{(t)} = \alpha_{ij}(w_j^{(t)} - w_i^{(t)}).$$

La charge échangée est donc une fraction α_{ij} de la différence de charge entre i et j. On notera que pour cet algorithme $\alpha_{ij} = \alpha_{ji}$. De ces remarques, découlent différentes contraintes sur les constantes α_{ij} : α_{ij} est compris entre 0 inclus et 1 inclus ; α_{ij} est égal à 0 si et seulement si l'arc (i, j) n'existe pas dans E. $\sum_j \alpha_{ij}$ est compris entre 0 exclu et 1 inclus ; en d'autres termes, un nœud i est toujours connecté à au moins un nœud j et il ne peut pas donner plus de charge qu'il n'en possède.

En regroupant les termes $w_i^{(t)}$, nous obtenons l'équation suivante :

$$w_i^{(t+1)} = (1 - \sum_j \alpha_{ij})w_i^{(t)} + \sum_j \alpha_{ij} w_j^{(t)}, \tag{1.2}$$

ce qui permet de mettre en évidence le fait que l'équation 1.1 est linéaire. La mise à jour de tout le système peut alors s'écrire sous la forme d'une équation vectorielle :

$$W^{(t+1)} = MW^{(t)}, \tag{1.3}$$

où $W^{(t)}$ est le vecteur de dimension n contenant la charge de tous les processeurs à l'instant t, et M est une matrice que nous nommons matrice de diffusion définie par m_{ij} telle que :

$$m_{ij} = \begin{cases} \alpha_{ij} & \text{si } i \neq j, \\ 1 - \sum_j \alpha_{ij} & \text{si } i = j. \end{cases}$$

Notons que $\sum_i m_{ij} = 1$, de plus $m_{ij} = m_{ji}$, la matrice M est donc symétrique et doublement stochastique. Il est prouvé dans [50] que cet algorithme converge vers une répartition uniforme W^* de la charge telle que :

$$w_i^* = \frac{\sum_{j=1}^n w_j^{(0)}}{n}.$$

Dans la littérature, plusieurs méthodes ont été définies pour construire la matrice de diffusion. Celle-ci influe sur la vitesse de convergence de l'algorithme, c'est-à-dire sur le nombre d'itérations afin que l'algorithme équilibre la charge du système. Une approche globale [113] permet de construire cette matrice. Elle a l'inconvénient de nécessiter une connaissance globale du réseau. Une approche locale [42] permet de déterminer les α_{ij} uniquement en fonction du degré du nœud i et des degrés de ses voisins j.

Pour trouver des exemples de fonctionnement de l'algorithme de diffusion, le lecteur intéressé est invité à consulter la thèse de Flavien Vernier [109].

1.2.2 Algorithme de dimension exchange généralisé : GDE

Les algorithmes de type "Dimension Exchange" sont dérivés de l'algorithme de diffusion. Ils sont basés sur des communications uni-ports, c'est-à-dire qu'un nœud i ne peut s'équilibrer qu'avec un seul de ses voisins j à un instant t donné. L'algorithme "Dimention Exchange" (DE), spécifique aux hypercubes, a été proposé par G. Cybenko dans [50]. Cet algorithme étant limité aux hypercubes, il est plus fréquent de considérer sa forme généralisée : GDE (Generalized Dimension Exchange).

Dans un premier temps, Hosseini et al. dans [79], proposent d'utiliser un graphe coloré afin de représenter les différentes dimensions sur un graphe quelconque. Dans un second temps, Xu et Lau dans [111] utilisent un paramètre λ à l'image de α_{ij} pour la diffusion. Ainsi, pour un graphe $G = (V, E)$ chaque arc (i, j) est associé à une couleur de sorte que pour un nœud donné, tous ses arcs aient une couleur différente. Le réseau est donc représenté par le graphe coloré $G_k = (V, E_k)$ avec E_k l'ensemble des triplets $(i, j; c)$ où i et j sont les sommets de l'arc (i, j) et c la couleur associée à l'arc. Notons que les couleurs sont numérotées de 0 à $k - 1$, où k est le nombre de couleurs du graphe. Si $d(i)$ représente le degré du nœud i et $d(G)$ le degré du graphe ($d(G) = \max_i d(i)$), il est prouvé dans [63] que le nombre minimum de couleurs k utiles à G_k est

$$d(G) \leq k \leq (d(G) + 1)$$

.

Le graphe ainsi défini, l'algorithme GDE pour un nœud i donné, s'exécute suivant l'équation (1.4).

$$\begin{aligned} w_i^{(t+1)} &= w_i^{(t)} + \lambda \left(w_j^{(t)} - w_i^{(t)} \right) && \text{si } \exists j | (i, j; c) \in E_k \wedge c = t \bmod k, \\ &= w_i^{(t)} && \text{sinon,} \end{aligned} \tag{1.4}$$

avec λ le paramètre de transfert. Notons que dans le cas de DE, $\lambda = \frac{1}{2}$. La mise à jour de la charge de tout le système s'effectue comme pour DE :

$$W^{(t+1)} = M_c W^{(t)}, \tag{1.5}$$

où M_c est la matrice de diffusion pour la couleur (dimension) c. La valeur de c est choisie en fonction de t, de manière à faire le cycle des k couleurs. Les couleurs étant numérotées de 0 à $k - 1$ ($0 \leq c \leq (k - 1)$), c est défini par

$$c = t \bmod k$$

Le paramètre λ peut être déterminé de deux manières différentes. L'approche la plus simple consiste à lui donner pour valeur $1/2$. Cette valeur permet d'équilibrer exactement la charge entre les deux processeurs connectés à l'itération courante. Dans [112], les auteurs définissent comment calculer λ de manière optimale, en fonction de la seconde plus grande valeur propre de la matrice M.

1.2.3 Diffusion relaxée

Le principe de ce nouvel algorithme est d'intégrer un paramètre de relaxation β positif dans l'algorithme de diffusion (1.1), afin d'accélérer la convergence de ce dernier. La diffusion de type relaxée s'écrit :

$$w_i^{(t+1)} = w_i^{(t)} + \beta \sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) \tag{1.6}$$

Par manipulation de l'équation 1.6, on obtient :

$$\begin{aligned} w_i^{(t+1)} &= (1-\beta)\, w_i^{(t)} + \beta w_i^{(t)} + \beta \textstyle\sum_j \alpha_{ij} w_j^{(t)} - \beta \sum_j \alpha_{ij} w_i^{(t)} \\ &= (1-\beta)\, w_i^{(t)} + \beta \left(1 - \textstyle\sum_j \alpha_{ij}\right) w_i^{(t)} + \beta \sum_j \alpha_{ij} w_j^{(t)} \\ &= (1-\beta)\, w_i^{(t)} + \beta \left[\left(1 - \textstyle\sum_j \alpha_{ij}\right) w_i^{(t)} + \sum_j \alpha_{ij} w_j^{(t)} \right], \end{aligned} \tag{1.7}$$

qui fait ressortir, en partie, l'expression de la diffusion. L'équation 1.7 peut donc se réécrire sous la forme vectorielle :

$$W^{(t+1)} = (1-\beta)\, W^{(t)} + \beta M W^{(t)}. \tag{1.8}$$

La convergence d'un tel algorithme est un résultat connu. Toutefois il est évident que β doit être correctement choisi afin d'accélérer la convergence et de respecter les contraintes sur la charge. La première condition est satisfaite si β est supérieur à 1. La seconde se limite à une seule contrainte à satisfaire : la positivité de la charge ; une charge $w_i^{(t+1)}$ négative n'a pas de sens. β doit donc être choisi, pour un i donné, tel que :

$$w_i^{(t)} + \beta \sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) \geq 0.$$

Il en découle que :

$$\beta \sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) \geq -w_i^{(t)}$$

Trois cas sont donc possibles :

- $\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) > 0$
  Dans ce cas :

  $$\beta \textstyle\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) \geq -w_i^{(t)}$$
  $$\beta \geq \frac{-w_i^{(t)}}{\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right)}$$

  Ceci implique que β doit être supérieur à une valeur négative, or β est défini positif.
- $\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) = 0$
  Dans ce cas, la positivité de $w_i^{(t+1)}$ ne dépend pas de β, puisque sous ces conditions : $w_i^{(t+1)} = w_i^{(t)}$.

– $\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) < 0$
Dans ce cas :

$$\begin{aligned}
\beta \textstyle\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) &\geq -w_i^{(t)} \\
\beta &\leq \frac{w_i^{(t)}}{\sum_j \alpha_{ij} \left(w_i^{(t)} - w_j^{(t)} \right)} \\
\beta &\leq \frac{w_i^{(t)}}{\sum_j \alpha_{ij} \left(w_i^{(t)} - \min_j \left(w_j^{(t)} \right) \right)} && \text{par minoration de } w_j^{(t)} \\
\beta &\leq \frac{w_i^{(t)}}{(1-M_{ii}) \left(w_i^{(t)} - \min_j \left(w_j^{(t)} \right) \right)} \\
\beta &\leq \frac{w_i^{(t)}}{(1-M_{ii}) \left(w_i^{(t)} - w_{min}^{(t)} \right)} && \text{avec } w_{min}^{(t)} = \min_j \left(w_j^{(t)} \right)
\end{aligned}$$

Cette borne supérieure est donc la seule que nous considérons, les deux cas précédents n'intervenant pas dans le choix de β. De plus, nous remarquons que cette borne dépend du temps. Jusqu'à présent, nous nous sommes restreints à un nœud i donné, pour la détermination de β. Pour le réseau complet, il en découle naturellement que :

$$\beta \leq \min_i \frac{w_i^{(t)}}{(1-M_{ii}) \left(w_i^{(t)} - w_{min}^{(t)} \right)} \quad \forall i | \textstyle\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) < 0. \tag{1.9}$$

Posons :

$$R^{(t)} = \min_i \frac{w_i^{(t)}}{(1-M_{ii}) \left(w_i^{(t)} - w_{min}^{(t)} \right)} \quad \forall i | \textstyle\sum_j \alpha_{ij} \left(w_j^{(t)} - w_i^{(t)} \right) < 0.$$

La fonction $R^{(t)}$ étant croissante et monotone, une borne déterminée à l'instant t est toujours valable à tout instant $t + n$ ($n \in N^+$). Une valeur de β déterminée à l'instant 0 est donc correcte à tout autre instant.

Remarque 1 *Ceci n'est vrai que dans le cas de charge statique. Dans un contexte de charge dynamique, $R^{(t)}$ n'est pas monotone, β devient donc fonction du temps et doit être recalculé à chaque itération.*

Notons pour finir, que si $\beta = 1$, la diffusion relaxée est équivalente à la diffusion de premier ordre.

1.2.4 Détermination du paramètre de relaxation β optimal

Le paramètre de relaxation β définit l'accélération de l'algorithme de diffusion. Il doit être choisi de manière optimale. Il est défini positif, mais il est connu que β doit être supérieur à 1 pour accélérer la diffusion.

La détermination de β optimal (β_{opt}) pour un algorithme de relaxation, et sous nos conditions, est définie dans [37]. Si M est une matrice stochastique, le paramètre optimal de relaxation est :

$$\beta_{opt} = \frac{2}{2 - (s + l)},$$

où s et l sont respectivement la plus petite et la seconde plus grande valeur propre de M.

Cette valeur optimale de β ne respecte pas forcément les conditions définies précédemment ($1 \leq \beta \leq R^{(t)}$). La borne supérieure de β dépendant du temps, notons donc $\beta_{opt}^{(t)}$ la valeur optimale de β pour chaque instant t. $\beta_{opt}^{(t)}$ est ainsi défini par

$$\beta_{opt}^{(t)} = \min\left(R^{(t)}, \frac{2}{2-(s+l)}\right) \tag{1.10}$$

Notons que pour finir, avec $\beta_{opt}^{(t)} = \beta_{opt}$, la vitesse de convergence de la diffusion relaxée est donnée par $-\ln \frac{l-s}{2-(s+l)}$ [37].

1.3 Conclusion

Ce chapitre présente les algorithmes du premier ordre utilisés pour répartir la charge d'un système distribué. L'algorithme de diffusion permet à tous les processeurs d'échanger une partie de leur charge avec leurs voisins. La variante appelée GDE repose sur une hypothèse différente, puisqu'à chaque itération, un nœud échange une partie de sa charge avec au plus un de ses voisins. Partant de ces algorithmes, nous avons proposé la diffusion relaxée qui permet, à l'aide d'un paramètre de relaxation, d'accélérer la convergence de l'algorithme.

Dans le chapitre suivant nous étudions comment adapter ces algorithmes d'équilibrage de charge sur réseaux dynamiques.

Chapitre 2

Réseaux dynamiques

Les algorithmes du chapitre précédent ont été conçus pour des réseaux dont la topologie est statique. Avec l'avènement des architectures de calcul distantes, certains liens de communication peuvent être surchargés, ou pire encore, être coupés. Dans ce cas, la topologie du réseau considéré est dynamique ; on parle alors de réseaux dynamiques.

Pour notre étude, nous nous limitons au dynamisme d'un point de vue liens de communication, un nœud du réseau ne peut ni spontanément apparaître, ni définitivement disparaître au cours de l'exécution de l'algorithme[1]. Sous ces conditions, les algorithmes d'équilibrage de charge, vus précédemment, ne peuvent s'exécuter correctement.

Nous proposons dans cette section leur adaptation aux réseaux dynamiques, nous avons publié ces adaptations dans [23, 24, 25].

La première étude de l'application de ces algorithmes aux réseaux dynamiques est proposée dans [30], mais cette étude contraint le réseau à être infiniment souvent connecté. Une étude récente est présentée dans [60], cette dernière confirme les résultats que nous présentons. Pour simplifier les explications, nous nous restreignons à une charge statique du système et à des réseaux homogènes. Comme dans le chapitre précédent, il est facile de généraliser les résultats en se référant par exemple à [58].

Ce chapitre présente, dans un premier temps, l'adaptation de l'algorithme de diffusion aux réseaux dynamiques. Cette modélisation nous permet d'énoncer les conditions qui assurent la convergence. En suivant le même raisonnement, nous proposons les adaptations aux topologies dynamiques de l'algorithme GAE et de l'algorithme de diffusion relaxée.

[1] Il est à noter que l'aspect dynamique des machines (apparition et disparition) est un travail sur lequel nous travaillons actuellement.

2.1 Algorithmes de diffusion de premier ordre

L'adaptation de l'algorithme de diffusion sur les réseaux dynamiques a été présentée dans [23].

2.1.1 Modélisation de l'algorithme

Avant de présenter l'algorithme de diffusion de premier ordre pour réseaux dynamiques, nous devons adapter le graphe représentatif de ce dernier afin qu'il prenne en compte le dynamisme du réseau. De manière générale, un réseau est modélisé par un graphe $G = (V, E)$ où V est l'ensemble des nœuds, et E est l'ensemble des arcs du réseau (cf. section 1.2.1). Le dynamisme est introduit par l'ensemble $E_B^{(t)}$ qui contient les arcs (i, j) inutilisables à l'instant t. $E_B^{(t)}$ est un sous-ensemble de E. Un réseau dynamique est donc représenté par le graphe $G^{(t)} = (V, E, E_B^{(t)})$. Notons que si $E_B^{(t)}$ est vide, $G^{(t)}$ est équivalent au graphe G.

Le réseau ainsi modélisé, nous pouvons donner l'adaptation de l'algorithme de diffusion présenté section 1.2.1 pour réseaux dynamiques. L'échange de charge entre deux processeurs i et j voisins est donné par l'équation 2.1.

$$l_{ij}^{(t)} = \begin{cases} \alpha_{ij}(w_j^{(t)} - w_i^{(t)}) & \text{si } (i,j) \in E \wedge (i,j) \notin E_B^{(t)}, \\ 0 & \text{sinon.} \end{cases} \tag{2.1}$$

En d'autres termes, deux processeurs i et j s'équilibrent, si et seulement si, ils sont voisins et le lien qui les relie est opérationnel. L'évolution de la charge d'un nœud i entre deux instants est donnée par la charge du processeur i plus la somme des charges échangées avec ses voisins (cf. équation 2.2).

$$w_i^{(t+1)} = w_i^{(t)} + \sum_j l_{ij}^{(t)}. \tag{2.2}$$

Aux conditions près, l'équation d'échange de charge entre deux processeurs est identique à celle de la diffusion sur réseaux statiques. La différence est liée au fait que l'algorithme ne s'applique pas sur un lien cassé ; ceci implique que la mise à jour du système dépend de l'ensemble $E_B^{(t)}$, et donc du temps. La matrice de diffusion est donc fonction du temps. L'équation vectorielle, modélisant la mise à jour de tout le système entre deux instants t et $t+1$ s'écrit, dans le cas de réseaux dynamiques, selon l'équation 2.3.

$$W^{(t+1)} = M^{(t)} W^{(t)}, \tag{2.3}$$

où $M^{(t)}$ est la matrice de diffusion à l'instant t, définie par :

$$m_{ij}^{(t)} = \begin{cases} \alpha_{ij} & \text{si } (i,j) \in E \wedge (i,j) \notin E_B^{(t)} \wedge i \neq j, \\ 1 - \sum_k \alpha_{ik} & \forall k | (i,k) \in E \wedge (i,k) \notin E_B^{(t)} \wedge i = j \\ 0 & \text{sinon.} \end{cases}$$

Notons que pour $m_{ij}^{(t)}$ ainsi défini, la matrice de diffusion $M^{(t)}$ est doublement stochastique comme M, mais n'est pas forcément connectée.

La détermination des paramètres de diffusion α_{ij} s'effectue selon les mêmes méthodes que dans le cas de réseaux statiques. Par contre, la détermination de paramètres optimaux ne peut pas être réalisée avec exactitude, du fait du dynamisme du réseau. Le paramètre α_{ij} optimal est donc toujours calculé à partir du réseau sans coupure.

2.1.2 Conditions de convergence sur réseaux dynamiques

Les caractéristiques de la matrice de diffusion induites par la détermination des paramètres de diffusion (doublement stochastique, connectée et non-bipartite), suffisent pour prouver, dans le cas de réseaux statiques, la convergence de ces algorithmes. Dans le cas présent, la perte de liens de communication peut rendre la matrice de diffusion non-connectée. Ces algorithmes convergent donc sous certaines conditions. Avant de préciser ces conditions, nous devons définir la notion de graphe de communication superposé.

Définition 1 *À chaque instant, le graphe de communication dédié à l'équilibrage de charge est le graphe contenant seulement les arcs utilisés pour les transferts de charge à cet instant. Le graphe de communication superposé dédié à l'équilibrage de charge noté $G_{t,t+n}$, est le graphe qui contient uniquement les arcs utilisés pour les transferts de charge entre les instants t et $t+n$. Le graphe de communication superposé est une représentation imagée de la superposition des graphes de communication entre deux instants t et $t+n$.*

Avec cette définition, nous pouvons donner les conditions de convergence.

Théorème 1 *L'algorithme de diffusion de premier ordre sur réseaux dynamiques converge vers une répartition uniforme de la charge si et seulement si pour tout instant t correspond un instant $t+L$ tel que le graphe de communication superposé $G_{t,t+L}$ soit connecté.*

La preuve de ce théorème est assez longue, c'est pourquoi nous invitons le lecteur intéressé à l'étudier dans la thèse de Flavien Vernier [109] ou dans [25]. Notons, d'une part, que ce théorème n'implique pas qu'un arc doive être absolument opérationnel à un instant t quelconque, ni que tous les arcs doivent être utilisables simultanément. D'autre part, il est à signaler que ces conditions sont proches de la réalité.

2.2 Generalized Adaptative Exchange

Étudions à présent l'adaptation aux réseaux dynamiques de l'algorithme GDE qui équilibre les processeurs deux à deux. Nous appelons cette adaptation GAE (Generalized Adaptative Exchange). La particularité de GAE par rapport à GDE est d'avoir

un degré de liberté quant au choix du voisin. L'algorithme GAE est présenté dans [25]. Avec l'algorithme GDE, les processeurs s'équilibrent par paires, suivant un ordre défini par la coloration du graphe. Cet ordre imposé peut être un frein à l'équilibrage lorsque GDE opère sur un réseau dynamique. Considérons à un instant donné qu'un nœud i doive s'équilibrer, selon la coloration, avec son voisin j ; cependant, l'arc (i, j) n'est pas opérationnel. Sous ces conditions et selon le mode opératoire de GDE, i et j ne s'équilibrent pas. Toutefois, il est possible que i ait un voisin k qui ne s'équilibre avec personne et pour lequel le lien (i, k) soit opérationnel. Il semble donc plus judicieux de ne pas respecter l'ordre imposé et d'équilibrer i avec k. Nous proposons donc un nouvel algorithme GAE, dérivant de GDE, pour lequel le choix des paires de voisins n'est pas imposé dans l'algorithme. Ce choix doit cependant être réalisé selon une stratégie (aléatoire, arbitraire (GDE) ou plus sophistiquée) respectant les contraintes des algorithmes de type "Dimension Exchange".

2.2.1 Modélisation de l'algorithme

D'un point de vue modélisation, nous considérons que GAE est l'algorithme de diffusion dans lequel chaque nœud possède au plus un arc non cassé. La stratégie de choix définit pour un nœud i, à un instant t, un voisin j tel que, d'une part, l'arc (i, j) soit opérationnel et que, d'autre part, j ne s'équilibre pas déjà avec un autre nœud. Tout autre arc (i, k) différent de (i, j) est considéré comme cassé : $(i, k) \in E_B^{(t)}$. La charge échangée entre deux processeurs se calcule donc selon l'équation 2.4.

$$l_{ij}^{(t)} = \begin{cases} \lambda(w_j^{(t)} - w_i^{(t)}) & \text{si } (i, j) \notin E_B^{(t)}, \\ 0 & \text{sinon.} \end{cases} \tag{2.4}$$

La mise à jour de la charge d'un nœud i donné se calcule selon l'équation 2.5.

$$w_i^{(t+1)} = w_i^{(t)} + l_{ij}^{(t)}. \tag{2.5}$$

Comme pour tous les algorithmes précédents, la mise à jour du système entre deux instants s'écrit sous la forme vectorielle donnée par l'équation 2.6.

$$W^{(t+1)} = M^{(t)} W^{(t)}, \tag{2.6}$$

où $M^{(t)}$ est la matrice de diffusion définie par :

$$m_{ij}^{(t)} = \begin{cases} \lambda & \text{si } (i, j) \in E \wedge (i, j) \notin E_B^{(t)} \wedge i \neq j, \\ 1 - \lambda & \exists k | (i, k) \in E \wedge (i, k) \notin E_B^{(t)} \wedge i = j \\ 1 & \nexists k | (i, k) \in E \wedge (i, k) \notin E_B^{(t)} \wedge i = j \\ 0 & \text{sinon.} \end{cases} \tag{2.7}$$

En d'autres termes, $m_{ij}^{(t)} = \lambda$ et $m_{ii}^{(t)} = 1 - \lambda$ si i communique avec un de ses voisins j à l'instant t, pour tout autre nœud k, $m_{ik}^{(t)} = 0$. Si i ne s'équilibre avec aucun de ses voisins à l'instant t, $m_{ii}^{(t)} = 1$ et $m_{ij}^{(t)} = 0$ pour tout nœud j.

2.2.2 Conditions de convergence

Nous savons que les algorithmes de type “dimension exchange” sont dérivés de la diffusion de premier ordre.

Corollaire 1 *GAE converge sous les conditions définies dans le théorème 1.*

La preuve est aisée puisque nous nous trouvons dans un cas particulier du théorème 1 où la matrice de diffusion M est définie par l'équation 2.7.

2.3 Algorithme de diffusion relaxée

Pour finir avec la mise en place des algorithmes de premier ordre sur réseaux dynamiques, intéressons nous à la diffusion relaxée.

2.3.1 Modélisation de l'algorithme

La modélisation de l'algorithme de diffusion relaxée pour des réseaux dynamiques, s'effectue en introduisant le paramètre de relaxation $\beta^{(t)}$ dans l'algorithme de diffusion dédié à ces réseaux, de la même manière que pour les réseaux statiques (cf. section 1.2.3). Ainsi, la charge échangée entre deux processeurs est donnée par l'équation 2.8.

$$l_{ij}^{(t)} = \begin{cases} \beta^{(t)}\alpha_{ij}(w_j^{(t)} - w_i^{(t)}) & \text{si } (i,j) \in E \wedge (i,j) \notin E_B^{(t)}, \\ 0 & \text{sinon.} \end{cases} \tag{2.8}$$

Comme pour le cas de la diffusion sur réseaux dynamiques, il n'y a échange de charge que si i et j sont voisins et que le lien qui les relie est opérationnel. La mise à jour d'un nœud donné i s'effectue de la même manière que pour la diffusion, le paramètre de relaxation étant inclus dans l'équation de la charge échangée. La charge de i à l'instant $t+1$ est calculée suivant l'équation 2.9 ; de manière littérale cette charge correspond à celle de i à l'instant t, plus à la somme des charges échangées.

$$w_i^{(t+1)} = w_i^{(t)} + \sum_j l_{ij}^{(t)}. \tag{2.9}$$

Les modifications engendrées sur l'équation vectorielle sont les mêmes qu'entre la diffusion sur réseaux statiques et sur réseaux dynamiques : la matrice M devient fonction du temps. L'évolution du système avec la diffusion relaxée est donc régie par l'équation 2.10.

$$W^{(t+1)} = \left(1 - \beta^{(t)}\right) W^{(t)} + \beta^{(t)} M^{(t)} W^{(t)}, \tag{2.10}$$

avec $M^{(t)}$ définie en section 2.1.1.

2.3.2 Paramètre de relaxation β optimal

La détermination du paramètre β optimal est, dans le cas de réseaux statiques, définie en section 1.2.4 par l'équation 1.10 que nous rappelons ici :

$$\beta_{opt}^{(t)} = \min\left(R^{(t)}, \frac{2}{2-(s+l)}\right)$$

Dans cette expression, nous avons $R^{(t)}$ qui est uniquement fonction de la répartition de charge à l'instant t, et $\frac{2}{2-(s+l)}$ qui dépend de la matrice de diffusion. Dans le contexte de réseaux dynamiques, la matrice diffusion peut être différente à chaque instant. Notons donc, $s^{(t)}$ et $l^{(t)}$ respectivement la plus petite et la seconde plus grande valeur propre de $M^{(t)}$. La valeur de β_{opt} peut ainsi être déterminée à tout instant par l'équation 2.11.

$$\beta_{opt}^{(t)} = \min\left(R^{(t)}, \frac{2}{2-(s^{(t)}+l^{(t)})}\right) \tag{2.11}$$

Bien que cette expression donne la valeur optimale de β, elle nécessite une connaissance de la matrice de diffusion à chaque instant, ce qui est contraire au caractère distribué de l'algorithme.

2.3.3 Conditions et preuve de convergence

L'algorithme de diffusion relaxée est dérivé de la diffusion de premier ordre et peut être vu comme un cas particulier de la diffusion où la matrice de diffusion est définie par

$$(1-\beta^{(t)})Id + \beta^{(t)}M^{(t)},$$

avec Id la matrice identité aux dimensions de $M^{(t)}$.

Proposition 1 *Avec β choisi selon l'équation 2.11, et sous les conditions définies dans le théorème 1, l'algorithme de diffusion relaxée converge.*

Pour prouver cette proposition il faut appliquer les résultats présentés dans [37].

2.4 Conclusion

Ce chapitre présente les adaptations des algorithmes de premier ordre (détaillés au chapitre 1) aux réseaux dynamiques. Le dynamisme du réseau est pris en compte par des coupures de lien dans la topologie du réseau. Pour l'instant, nous n'avons pas encore étudié le contexte plus général dans lequel les nœuds sont dynamiques, c'est-à-dire lorsque des processeurs apparaissent ou disparaissent définitivement.

L'adaptation des algorithmes de premier ordre aux réseaux dynamiques consiste à modifier la matrice de diffusion afin de prendre en compte les liens inutilisables.

L'algorithme GAE que nous proposons comme adaptation de GDE n'utilise plus la coloration du graphe. Ainsi, il est capable de s'adapter au dynamisme du réseau. Nous avons également proposé une version dynamique de l'algorithme de diffusion relaxée.

Cette première partie présente des algorithmes décentralisés d'équilibrage de charge. Ceux-ci ont pour objectif de diminuer les temps d'exécution des algorithmes numériques en réduisant les temps de synchronisations. Une autre approche pour atteindre le même objectif consiste à utiliser des algorithmes itératifs asynchrones. La seconde partie de ce document est consacrée à l'étude de ces algorithmes en vue d'obtenir des applications performantes, notamment dans un contexte de grappes de calcul distantes.

Deuxième partie

Mise en œuvre efficace d'algorithmes itératifs asynchrones

Chapitre 3

Méthodes itératives synchrones et asynchrones

3.1 Introduction

La résolution de problèmes complexes et de grande taille requiert l'utilisation simultanée de plusieurs ressources de calculs. Les grappes de machines, composées d'ordinateurs standards reliés par un réseau local rapide, permettent de répondre à ce besoin de ressources dans certains cas. Cependant, afin d'obtenir une capacité de calcul en théorie illimitée, la meilleure solution consiste à associer des grappes de calcul. Ainsi, nous obtenons des grappes distantes, également appelé grilles de calcul (ou grid computing en anglais).

Les grilles de calcul sont hétérogènes au niveau des machines et au niveau des réseaux. De plus, les débits et les latences de communication étant variables, il est nécessaire d'élaborer des algorithmes prenant ces caractéristiques en compte.

Pour résoudre un problème numérique on distingue les méthodes directes des méthodes itératives. Les premières donnent la solution exacte du problème modulo les erreurs d'arrondi. Les secondes convergent vers la solution du problème par approximation successive de la solution. Les algorithmes itératifs sont utilisés pour résoudre de nombreuses classes de problèmes [11], citons par exemple :

- les systèmes linéaires avec les méthodes de Jacobi, de Gauss, les méthodes de gradient et de relaxation, GMRES, la plupart de ces algorithmes se trouvant dans [103],
- les systèmes non linéaires avec les équations différentielles ordinaires (ODE) et les équations aux dérivées partielles (EDP) dont la résolution utilise la méthode de Newton,
- certaines méthodes de recherche de valeurs propres [110], les méthodes de recherche de racines de polynômes [40], les algorithmes d'optimisation [81] dont notamment les algorithmes génétiques [96], les calculs de radiosité en image-

rie [72] et bien d'autres.

Les méthodes itératives sont généralement assez faciles à programmer et à paralléliser. La parallélisation la plus simple consiste à distribuer le travail effectué dans une itération aux processeurs. A chaque nouvelle itération, les processeurs s'échangent les données dont ils ont besoin et déterminent si le critère de convergence est atteint. Ces algorithmes parallèles sont synchrones et ont le même comportement que les algorithmes séquentiels dont ils sont dérivés. Néanmoins, les synchronisations peuvent se révéler très coûteuses dans un contexte hétérogène et distant. Une solution pour réduire les temps de synchronisations consiste à recouvrir une partie des communications par du calcul.

Les algorithmes parallèles itératifs asynchrones sont bien adaptés au contexte de grappes distantes puisqu'ils ne nécessitent aucune synchronisation. Ces algorithmes supportent l'hétérogénéité des processeurs et des réseaux. Les itérations sont différentes du mode synchrone et il faut étudier minutieusement la convergence des algorithmes asynchrones.

D'un point de vue pratique, peu de personnes utilisent les algorithmes asynchrones. Ceci s'explique probablement par différentes raisons.

- Les algorithmes asynchrones sont beaucoup moins connus que les algorithmes synchrones.
- Les algorithmes asynchrones paraissent plus difficiles à programmer que les algorithmes synchrones.
- Les environnements de programmation parallèle usuels ne sont pas adaptés au développement des algorithmes asynchrones.

Dans la suite de ce document, nous verrons pourquoi les algorithmes asynchrones offrent une alternative intéressante au synchronisme et comment les programmer efficacement pour obtenir des algorithmes très performants.

La suite de ce chapitre est organisée de la manière suivante. Nous présentons dans un premier temps les méthodes itératives, puis nous proposons une classification des algorithmes itératifs parallèles. Finalement nous détaillons les différentes variantes des algorithmes asynchrones.

3.2 Algorithmes itératifs

Avant d'aborder le principe des algorithmes itératifs asynchrones, il est indispensable d'aborder le principe des algorithmes itératifs séquentiels.

3.2.1 Algorithmes itératifs séquentiels

On s'intéresse à un problème dy type :

$$x = f(x) \tag{3.1}$$

Le modèle d'algorithme itératif présente la structure suivante sous forme d'équation ou d'algorithme.

$$x^{k+1} = f(x^k), \qquad k = 0, 1, \ldots \tag{3.2}$$

avec x^0 donné.

Algorithme 3.1 Modèle d'algorithme itératif séquentiel

x^0 donné
for k=0,1,... **do**
  $x^{k+1} = f(x^k)$
end for

Les x^k sont des vecteurs de dimension n, et f est une fonction de $\mathbb{R}^n$ dans $\mathbb{R}^n$. Si la suite des itérés x^k engendrée par l'algorithme 3.1 converge et si f est continue, alors la propriété $x^* = f(x^*)$ est vérifiée et x^* est solution de (3.1).

Pour mettre en œuvre un algorithme itératif séquentiel, on définit un seuil d'arrêt afin de terminer l'algorithme dès lors que celui-ci est passé sous le seuil. La plupart des algorithmes utilisent un critère défini à partir d'une norme sur les 2 dernières itérations. Par exemple, on peut choisir d'appliquer la norme :

$$\max_{i \in 1..n} |x_i^{k+1} - x_i^k| < \epsilon \tag{3.3}$$

où ϵ représente le critère d'arrêt et x_i^k représente la composante i du vecteur x à l'itération k. En séquentiel, il existe d'autres normes qui sont applicables pour calculer le résidu entre 2 itérations. Néanmoins, pour la suite, nous utiliserons cette norme appelé norme du max.

3.2.2 Algorithmes itératifs parallèles synchrones

La parallélisation la plus simple des algorithmes itératifs consiste à découper le vecteur x^k en m blocs de composantes et à partitionner la fonction f en m partie de manière compatible. Ainsi on obtient :

$$x^k = (x_1^k, x_1^k, \ldots, x_n^k) \equiv X^k = (X_1^k, X_2^k, \ldots, X_m^k)$$

L'équation 3.2 peut alors s'écrire sous la forme suivante :

$$(X_1^{k+1}, X_2^{k+1}, ..., X_m^{k+1}) = (F_1(X^k), F_2(X^k), ..., F_m(X^k)), \qquad k = 0, 1, \ldots \tag{3.4}$$

Voici le modèle d'algorithme itératif synchrone correspondant :

À partir de cette formulation, un algorithme itératif peut être parallélisé en associant un bloc de composantes à chaque processeur. À chaque itération, le processeur

Algorithme 3.2 Modèle d'algorithme itératif synchrone

$(X_1^0, ..., X_m^0)$ est fixé
for k=0,1,... **do**
  for i=1,...,m **do**
    $X_i^{k+1} = F_i(X_1^k, ..., X_m^k)$
  end for
end for

chargé de calculer le bloc de composantes i connaît les valeurs de toutes les composantes impliquées dans le calcul de F_i, il calcule les nouvelles composantes de X_i^{k+1} et les envoie à tous les processeurs qui en ont besoin pour calculer l'itération suivante.

Il est alors possible de proposer un algorithme générique qui peut s'adapter simplement en fonction du problème que l'on souhaite résoudre. Selon les modes de communications employés, on peut différencier deux variantes d'algorithmes.

La forme la plus simple consiste à utiliser des communications synchrones. Nous qualifions l'algorithme obtenu d'ISCS pour Itérations Synchrones avec Communications Synchrones. Avec cette variante, les communications sont effectuées à la fin d'une itération (donc après la phase de calcul). L'algorithme 3.3 schématise le principe d'ISCS. Le tableau *AnciennesDonnees* (qui représente $X_1^k, ..., X_m^k$) peut, suivant le calcul considéré (représenté par F_i) contenir ou non la totalité des données, alors que le tableau *NouvellesDonneesLoc* (qui représente X_i^{k+1}) ne contient que les données locales d'un processeur.

Algorithme 3.3 Algorithme itératif parallèle ISCS

Initialisation de la bibliothèque de communication
AnciennesDonnees = *Tableau des valeurs de l'itération précédente*
NouvellesDonneesLoc = *Tableau des nouvelles valeurs locales*
Initialisation des données
repeat
  Calcul de NouvellesDonnesLoc en fonction de AnciennesDonnees
  Envois non bloquants des données de NouvellesDonnesLoc aux processeurs qui en ont besoin
  Réceptions bloquantes des données des processeurs voisins dans AnciennesDonnees
  Recopie de NouvellesDonnesLoc dans AnciennesDonnees
  Détection de convergence
until Convergence globale atteinte

La seconde variante consiste à utiliser des communications asynchrones. L'avantage de cette variante est de pouvoir recouvrir une partie des communications par du calcul. Ainsi, l'exécution de l'algorithme est, à priori, plus rapide. Nous appelons cette

variante ISCA pour Itérations Synchrones avec Communications Asynchrones. Au niveau algorithmique, les communications sont légèrement différentes afin de traiter le recouvrement. Selon le calcul considéré, il faut envoyer les données aux processeurs qui en ont besoin le plus rapidement possible afin de poursuivre le calcul, et pour qu'une partie des communications soit effectuée en même temps que la fin du calcul de l'itération.

Algorithme 3.4 Algorithme itératif parallèle ISCA

Initialisation de la bibliothèque de communication
AnciennesDonnees = *Tableau des valeurs de l'itération précédente*
NouvellesDonnesLoc = *Tableau des nouvelles valeurs locales*
Initialisation des données
repeat
  Calcul de NouvellesDonnesLoc en fonction de AnciennesDonnees avec envois non bloquants des données de NouvellesDonnesLoc aux processeurs qui en ont besoin *(tous les envois à l'exception du dernier sont susceptibles d'être recouverts par du calcul)*
  Réceptions bloquantes des données des processeurs voisins dans AnciennesDonnees
  Recopie de NouvellesDonnesLoc dans AnciennesDonnees
  Détection de convergence
until Convergence globale atteinte

Ces deux variantes engendrent les mêmes suites d'itérés que les algorithmes séquentiels leur correspondant. L'intérêt des algorithmes ISCS et ISCA est qu'ils convergent dès lors que la version séquentielle converge. La détection de convergence au niveau pratique ne pose pas de difficulté. En effet, il suffit que chaque processeur applique le critère d'arrêt sur ces données locales. Ensuite les processeurs combinent leurs résultats afin d'obtenir la convergence globale. C'est cette phase de combinaison qui réalise une synchronisation.

3.2.3 Algorithmes itératifs parallèles asynchrones

À partir du découpage du vecteur x^k obtenu précédemment, on peut obtenir un algorithme parallèle itératif asynchrone en désynchronisant les itérations. Les algorithmes asynchrones sont plus généraux que les algorithmes synchrones. Autrement dit, un algorithme synchrone n'est qu'un cas particulier d'un algorithme asynchrone (dans lequel les itérations sont synchronisées). Ceci étant dit, les différences surviennent au niveau du modèle et de l'implantation. En désynchronisant les itérations, les communications sont forcément asynchrones. Afin de rester cohérent avec les dénominations précédentes, nous renommons les algorithmes asynchrones en al-

gorithmes IACA. Cet acronyme a pour signification Itérations Asynchrones avec Communications Asynchrones. Cette dénomination possède l'avantage de lever l'ambiguïté sur le mot "asynchrone" qui concerne les itérations et les communications (et non seulement les communications comme on le rencontre dans de nombreux travaux).

Rappelons dans un premier temps le modèle d'un algorithme IACA ou algorithme parallèle itératif asynchrone. Partons de la décomposition par blocs de composantes X^k obtenue dans la section précédente. Dans le modèle asynchrone, la dernière version d'un bloc de composantes X_i n'est pas forcément disponible sur tous les processeurs, seule une version antérieure est disponible. Le nœud i responsable du calcul du bloc X_i possède, bien entendu, la dernière version. Notons $X_i^{s_i^j(k)}$ la dernière version disponible du bloc de composantes i sur le processeur j à l'itération k avec $0 \leq s_i^j(k) \leq k$. Soit $J(k)$ l'ensemble des nœuds mis à jour à l'itération k, un algorithme IACA est modélisé par :

Algorithme 3.5 Modèle d'algorithme IACA

for k=0,1,... **do**
  for i=1,...,m **do**
    if $i \in J(k)$ **then**
      $X_i^{k+1} = F_i(X_1^{s_1^i(k)}, ..., X_m^{s_m^i(k)})$
    else
      $X_i^{k+1} = X_i^{k+1}$
    end if
  end for
end for

À chaque itération, certains processeurs (ceux qui appartiennent à $J(k)$) calculent une nouvelle itération, les autres nœuds reprennent le résultat de l'itération précédente. Lorsqu'un processeur calcule une nouvelle itération, il prend les dernières données dont il dispose. Par conséquent, bien que ce modèle présente une allure synchrone, celui-ci est parfaitement adapté au cadre asynchrone, puisque tous les processeurs ne se mettent pas à jour simultanément. Il faut, par ailleurs, ajouter que dans le cadre le plus général les délais de mises à jour des données ne sont pas bornés, c'est-à-dire qu'il est tout à fait possible que ceux-ci soient très longs.

Pour assurer la convergence de tels modèles, deux conditions doivent être vérifiées [57, 38]. La première condition énonce que les composantes du système doivent être infiniment souvent mises à jour ($\forall i \in \{1, ..., m\}$ l'ensemble $\{k, i \in J(k)\}$ est infini). La seconde condition consiste à vérifier que les mises à jour des données non locales suivent l'évolution des itérations ($\lim_{k\to\infty} s_j^i(k) = \infty$, $\forall i$ et $j \in \{1, ..., m\}$) bien que les délais ne soient pas bornés.

À partir du modèle des algorithmes IACA, il est possible d'implanter ce type d'algorithme de plusieurs manières différentes. Nous détaillons l'implantation efficace des

algorithmes IACA dans le chapitre suivant.

Pour plus de renseignement sur les algorithmes asynchrones nous invitons le lecteur intéressé à consulter [47, 94, 34, 56, 57, 38, 65].

Remarque 2 *Le modèle d'algorithme IACA présenté dans ce document est différent de l'approche choisie par les méthodes hybrides permettant de calculer la solution d'un problème, comme l'a fait Nahid Emad pour le calcul des valeurs propres [61]. En effet, l'objectif des méthodes hybrides est de combiner, de manière asynchrone, plusieurs méthodes numériques afin de calculer efficacement la solution d'un problème. Notre approche consiste à répartir le vecteur solution entre les processeurs, alors que les méthodes hybrides combinent les résultats de plusieurs calculs, initialisés avec des valeurs différentes ; les calculs étant effectués sur l'ensemble du vecteur.*

3.3 Classification des algorithmes

Afin de comprendre les différences fondamentales entre les précédents algorithmes, nous illustrons graphiquement les algorithmes ISCS, ISCA et IACA dans les figures suivantes. Nous avons présenté cette classification dans [14] et [16].

La figure 3.1 représente la parallélisation la plus triviale, au niveau informatique, des algorithmes synchrones. Sur cette figure et sur les suivantes, nous avons représenté les itérations de deux processeurs. Les rectangles schématisent les itérations, et les parties en blanc représentent les temps morts entre deux itérations. Dans le mode ISCS, les échanges de données sont réalisés à la fin d'une itération. Comme les communications sont synchrones, le processeur ayant terminé son itération avant son voisin doit attendre que son voisin ait également terminé son itération afin d'initier la communication. Ce temps d'attente est représenté par le pointillé sur le dessin avant la flèche qui symbolise l'envoi. Sur ce schéma n'apparaissent pas les communications afin de détecter la convergence. En ce qui concerne les algorithmes synchrones, la détection de convergence joue généralement le rôle de barrières de synchronisation, puisqu'une nouvelle itération ne démarre qu'après avoir testé la convergence de l'itération précédente. Par conséquent, les temps d'attente entre deux itérations peuvent être importants en raison des communications et de la détection de convergence pour les algorithmes synchrones.

La figure 3.2 illustre un moyen de réduire les temps d'attente pour les algorithmes synchrones. En utilisant des communications asynchrones, une partie des communications a la possibilité d'être recouverte par du calcul. Pour ce faire, il faut réaliser plusieurs communications au sein d'une même itération. Dès qu'une partie des résultats d'une itération est calculée, il est possible d'envoyer ce résultat. Ainsi, si l'envoi est réalisé avec des communications asynchrones, il pourra être recouvert par du calcul. Néanmoins, une partie des résultats doit nécessairement être envoyée à la fin de l'itération. Sur la figure, deux envois de données apparaissent par itération. Si un mes-

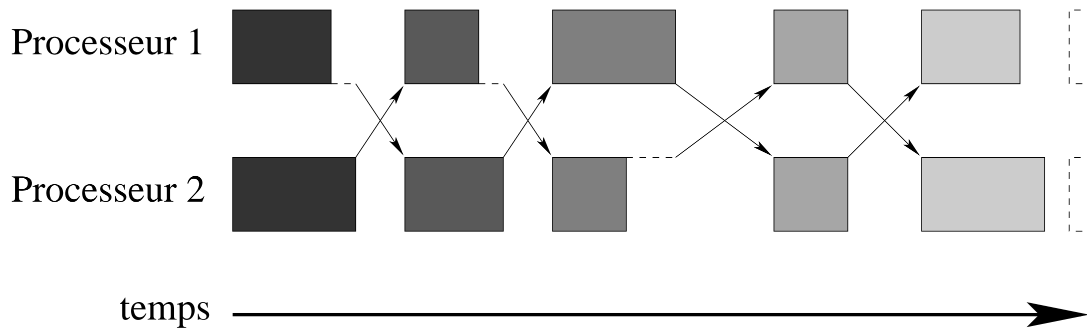

FIG. 3.1 – Algorithme ISCS

sage arrive alors qu'un processeur n'a pas achevé son itération, la réception n'a lieu qu'à la fin de l'itération. Ce temps d'attente est matérialisé par des pointillé apparaissant après certains envois de la première partie des échanges de données. L'utilisation des communications asynchrones permet de réduire les temps d'attente entre les itérations. Les temps d'exécution des algorithmes ISCA sont par conséquent, à priori, inférieurs aux temps d'exécution des algorithmes ISCS.

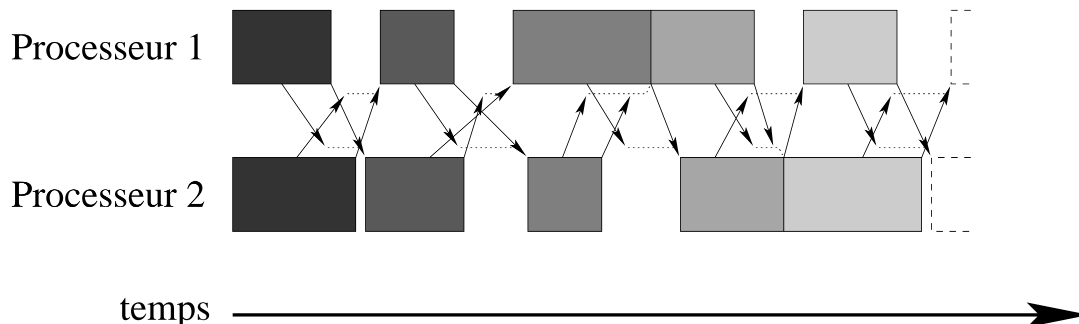

FIG. 3.2 – Algorithme ISCA

La figure 3.3 permet de remarquer la caractéristique fondamentale propre aux algorithmes IACA. Un nœud n'attend pas de recevoir les dépendances de ses voisins avant de commencer une nouvelle itération. Il en résulte que les temps morts entre les itérations ont disparu. Sur la figure nous avons représenté le cas le plus simple à implanter. Il s'agit du cas pour lequel il n'y a qu'un envoi pour chaque voisin à la fin d'une itération et les réceptions n'ont lieu qu'au début d'une nouvelle itération. Il est possible qu'un processeur reçoive plusieurs messages du même voisin correspondant à des itérations différentes. La partie en pointillé sur la figure représente le temps nécessaire avant qu'un message arrive et soit traité par un nœud. Cette situation est visible sur la figure 3.3 pour le processeur 2, au début de l'itération 7. En effet, le processeur 2 reçoit les messages du processeur 1 correspondants aux itérations 5, 6 et 7. Dans ce cas, seul le dernier message est intéressant puisque les autres sont antérieurs.

Si les messages peuvent arriver dans un autre ordre que celui de leur émission, il est nécessaire de connaître l'itération à laquelle ils ont été émis. La suppression de toutes les synchronisations est contraire à l'utilisation simultanée d'une étape de détection de convergence sur tous les processeurs. C'est pourquoi la détection de convergence est plus complexe à traiter pour les algorithmes IACA que pour les algorithmes synchrones. Les propriétés intéressantes des algorithmes IACA sont les suivantes :

- Les algorithmes IACA tolèrent les pertes de messages totales et / ou partielles et permettent de recouvrir les communications par du calcul.
- Ils sont adaptés aux environnements peu propices aux algorithmes synchrones. On peut citer par exemple les environnements de grappes distantes possédant des latences non négligeables et des débits variables, mais également supportant bien l'hétérogénéité des machines et des réseaux. Ceci en raison du fait qu'un processeur effectue ses itérations librement, sans synchronisation avec ses voisins.

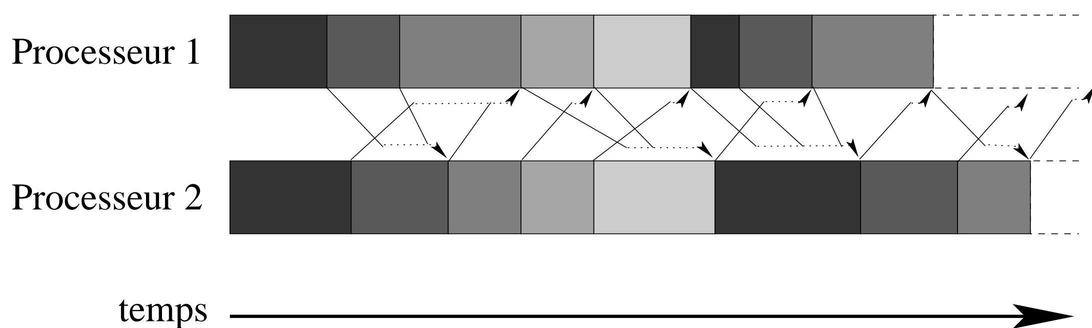

FIG. 3.3 – Algorithme IACA avec communications rigides

Il existe plusieurs variantes des algorithmes IACA suivant la manière de gérer les communications.

3.4 Variantes des algorithmes IACA

La figure 3.3 illustre le modèle le plus simple des IACA. Il correspond dans la littérature aux algorithmes asynchrones avec communications rigides.

Pour modéliser les architectures à mémoire partagée, le modèle avec communications flexibles a été proposé dans [36, 64]. Le principe est d'envoyer des valeurs intermédiaires afin de faire progresser plus rapidement le calcul. Les valeurs reçues sont immédiatement prises en compte. Dans le modèle ISCA, plusieurs envois sont effectués à chaque itération, mais le but est d'envoyer une partie des données finales afin de recouvrir une partie des communications par du calcul. Avec les communications flexibles, on peut également procéder ainsi, mais initialement l'approche est différente.

La figure 3.4 illustre le comportement de 2 processeurs exécutant un algorithme IACA avec communications flexibles. À chaque itération, un processeur envoie deux messages à son voisin. Le premier message d'une itération, illustré par un trait en pointillé, contient des résultats partiels de l'itération courante. Le second message intervient à la fin de l'itération et contient le résultat final. De plus, les messages sont pris en compte à leur arrivée, et non à l'itération suivante. Sur la figure, il apparaît clairement que le nombre de messages est plus important. Par contre, cette figure ne fait pas ressortir la vitesse de convergence. Avec des communications flexibles, la convergence est normalement plus rapide. Ce type d'algorithme étant prévu pour des architectures à mémoire partagée, comportant des débits de communications importants, n'est pas spécialement adapté au contexte de grappes distantes.

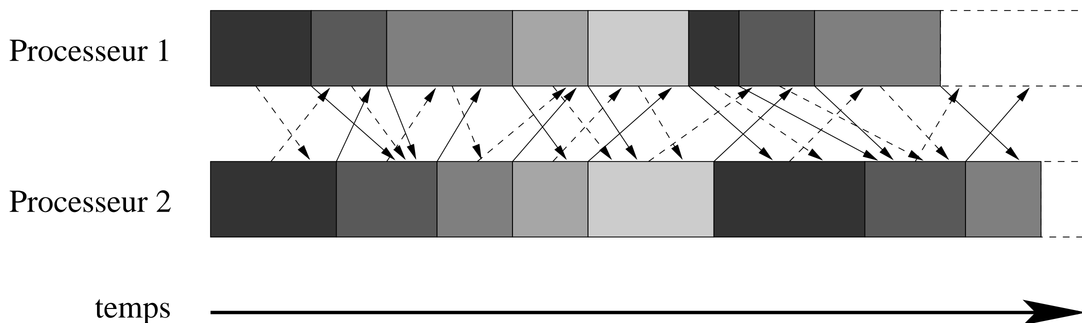

FIG. 3.4 – Algorithme IACA avec communications flexibles

Pour notre problématique, il est intéressant d'intégrer les messages dès leur arrivée (et non à l'itération suivante). Cela ne prend pas plus de temps et le nombre d'itérations ne peut qu'être réduit. Dans la suite de ce document, pour des raisons d'efficacité, nous étudions et développons des algorithmes IACA avec des communications que nous qualifions de semi-flexibles. La figure 3.5 illustre l'exécution d'un algorithme IACA avec communications semi-flexibles. Pour ne pas saturer inutilement les réseaux ayant un faible débit, nous choisissons de ne pas envoyer un message à un voisin qui n'aurait pas reçu un message précédent. C'est pourquoi certains messages ne seront tout simplement pas émis. Ils sont représentés par un trait en pointillé. Par exemple, le second message du processeur 1 n'est pas émis car au moment où il devait l'être, le premier message n'était pas arrivé. Comme les algorithmes IACA supportent la perte totale de messages, avec ce principe, la bande passante n'est pas surchargée inutilement.

3.5 Conclusion

Ce chapitre présente les différentes versions des algorithmes itératifs parallèles en fonction du mode de synchronisation des itérations et des communications. Nous

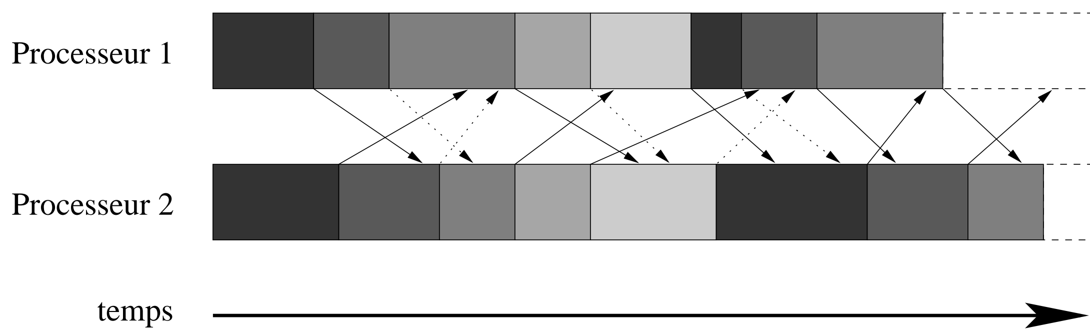

FIG. 3.5 – Algorithme IACA avec communications semi-flexibles

détaillons les algorithmes IACA puisque que nous les développons par la suite. Finalement, nous expliquons les variantes des algorithmes IACA en fonction de la manière dont les communications sont mises en œuvre.

Le chapitre suivant détaille les éléments à prendre en compte pour concevoir des algorithmes IACA performants.

Chapitre 4

Conception d'algorithmes IACA efficaces

4.1 Introduction

Dans le chapitre précédent nous avons présenté les algorithmes IACA. Leur mise en œuvre peut paraître délicate. Suivant les outils utilisés, l'implantation de tels algorithmes ne pose pas de difficulté importante. Avant de détailler cela par la pratique sur des exemples concrets, ce chapitre explique comment concevoir des algorithmes IACA performants.

L'élément essentiel pour répondre à tous les cas de figures est d'utiliser un environnement multithreadé, c'est-à-dire composé de plusieurs processus légers qui s'exécutent simultanément sur un processeur. En effet, ce type d'environnement permet de dissocier le calcul des communications, ceci en utilisant des threads différents.

On peut, bien entendu, s'en passer, mais cela complique nettement la tâche. Les difficultés pour concevoir des algorithmes IACA peuvent être résumées par deux points.

- Actuellement, peu d'environnements permettent de concevoir des algorithmes IACA assez facilement, car ces environnement doivent être multithreadés et trop peu d'environnements possèdent cette caractéristique.
- La principale difficulté pour développer efficacement des algorithmes IACA tient au fait qu'il faut maîtriser les concepts de programmation multithreadée.

Dans un premier temps, nous justifions pourquoi les environnements monothreadés ne sont pas adaptés aux développements d'algorithmes IACA. Ensuite, nous expliquons les raisons qui justifient l'utilisation d'une bibliothèque de communication multithreadée. Finalement nous indiquons deux méthodes permettant de détecter la convergence.

4.2 Inadéquation des environnements mono-threadés

La très grande majorité des outils standards de programmation parallèle basés sur le paradigme d'envois de messages, comme PVM [69] ou MPI [75] dans la plupart de leur implantation, ne permet pas de développer une application qui utilise plusieurs processus légers. On trouve au moins trois raisons à cela. Une première raison est que peu d'applications parallèles nécessitent (du moins pour l'instant) d'utiliser plusieurs threads. Une seconde raison provient du fait que la gestion des threads, dans une application, peut être source de blocage, surtout avec l'utilisation des mécanismes d'exclusion mutuelle, donc la programmation peut sembler plus délicate au début. Enfin il est difficile de créer une bibliothèque de communication par envoi de messages qui soit multithreadée. C'est pourquoi il en existe peu actuellement. En fait, il est probablement plus facile de concevoir une bibliothèque de communication multithreadée dès le début, plutôt que de rendre une bibliothèque existante multithreadée.

La conception d'algorithmes IACA avec communications rigides (cf. chapitre précédent), dans un environnement de programmation monothreadé n'est pas triviale. Avec les autres modes de communication (semi-flexible, et flexible) c'est même impossible puisqu'il faut traiter les messages à leur réception, et non à l'itération suivante. En effet, les bibliothèques par envoi de messages nécessitent l'utilisation d'une fonction de réception pour recevoir un message. Cette fonction de réception est placée à certains endroits dans le code, mais en aucun cas, elle ne peut être présente à tous les endroits (sinon il n'y a plus de calcul). Concernant le cas des communications rigides, la difficulté provient du fait qu'au début d'une nouvelle itération un processeur doit prendre le dernier message provenant d'un voisin. Trois cas de figures sont alors envisageables.

1. Aucun nouveau message n'est parvenu de la part d'un voisin particulier depuis la dernière itération. Dans ce cas, il n'y a pas de mises à jour de la part de ce voisin.
2. Un seul message est parvenu, donc il faut le lire et le prendre en compte afin de démarrer l'itération courante.
3. Plusieurs messages sont parvenus de la part d'un même voisin. Dans ce cas, seul le dernier est intéressant (si l'on suppose qu'ils sont réceptionnés dans leur ordre d'émission). Les bibliothèques standards ne permettent pas de savoir combien de messages sont arrivés et ne permettent pas d'obtenir directement la dernière version de ces messages, donc il faut les lire un à un jusqu'à ce qu'il n'y en ait plus.

Bien évidemment, le troisième cas généralise les deux premiers. Mais avec les bibliothèques standards, c'est au programmeur qu'il incombe d'effectuer cette tâche. De plus, en terme de performance, certains messages sont envoyés inutilement sur le réseau, et de ce fait, consomment des ressources processeurs et réseaux.

4.3 Algorithme IACA dans un environnement multi-threadé

Comme nous l'avons évoqué dans la section précédente, la programmation d'applications parallèles multithreadées peut paraître plus complexe, surtout au début. Concernant les algorithmes IACA, nous pensons réellement que l'utilisation de plusieurs threads représente un atout majeur qui simplifie même le développement des algorithmes. La séparation du code effectuant les itérations et des threads réalisant les communications offre l'avantage de structurer plus clairement un programme. De plus, dans la majorité des cas, les inconvénients liés aux blocages d'applications multithreadées n'existent pas. En effet, le modèle de programmation des algorithmes IACA ne nécessite pas de synchronisation particulière entre les threads de communication et le thread de calcul. Par conséquent, les sources de blocage dues à l'utilisation d'exclusion mutuelle ne se rencontrent pas dans la majorité des applications.

Pour comprendre le fonctionnement d'un algorithme IACA selon le mode de communication qui nous intéresse, à savoir les communications semi-flexibles, nous allons le détailler. L'algorithme 4.1 réalise la plupart du travail. Il commence par initialiser l'algorithme et il crée les threads pour les communications entrantes et sortantes. Suivant l'application considérée, et surtout les dépendances qu'elle engendre en fonction de la décomposition, il est préférable de créer autant de threads que de voisins pour chaque nœud (si ce nombre de voisins est petit) ou au contraire, il est indispensable de regrouper la gestion de plusieurs voisins dans un même thread. Dans l'algorithme 4.1 n'apparaissent pas les communications puisqu'elles sont déléguées à des threads.

Algorithme 4.1 Algorithme IACA avec communications semi-flexibles

Initialisation de la bibliothèque de communication
AnciennesDonnees = *Tableau des valeurs de l'itération précédente*
NouvellesDonneesLoc = *Tableau des nouvelles valeurs locales*
Création des threads pour les communications entrantes (appel à EnvoisDonnees)
Création des threads pour les communications sortantes (appel à ReceptionsDonnees)
Initialisation des données
repeat
  Calcul de NouvellesDonnesLoc en fonction de AnciennesDonnees
  Activation des threads chargés des communications sortantes
  Recopie de NouvellesDonnesLoc dans AnciennesDonnees
  Détection de convergence
until Convergence globale atteinte

Dans la fonction 4.2 exécutée par le ou les threads chargés des communications sortantes, nous faisons apparaître un mécanisme assurant qu'un nouvel envoi ne peut se

produire que si le précédent est déjà terminé. En fait, suivant si l'on utilise des communications bloquantes ou non, et suivant la bibliothèque utilisée, il faudra utiliser un mutex et un message d'acquittement pour vérifier si le précédent message est arrivé. En général, il est pratique d'utiliser des envois bloquants. Un message bloquant effectué par un thread ne bloquera que le thread et permet de ne pas surcharger le réseau comme indiqué dans la section 3.4.

D'autre part, de nombreux algorithmes entrainent qu'un processeur communique avec plusieurs voisins. Dans ce cas, si l'on choisit d'utiliser un seul thread pour envoyer les messages à ces différents voisins, selon les applications, il est possible que ces messages soient différents.

Algorithme 4.2 fonction EnvoisDonnees()

for les voisins concernés i **do**
  if envoi précédent au voisin i est terminé **then**
    envoi d'une partie de NouvellesDonneesLoc au processeur i
  end if
end for

La fonction 4.3 est exécutée en boucle par le ou les threads chargés des communications sortantes. Il est impératif d'utiliser une fonction de réception bloquante qui laissera le thread "endormi" tant qu'aucun message n'est arrivé et qui le "réveillera" à l'arrivée d'un nouveau message. Suivant la bibliothèque de communication utilisée, les mécanismes offerts et le besoin du programmeur, il peut paraître intéressant d'envoyer un message d'acquittement pour confirmer qu'un message a été reçu. En fonction de l'émetteur il faut placer les données dans le tableau *AnciennesDonnees* à l'endroit correspondant.

Algorithme 4.3 fonction ReceptionsDonnees()

Réception des données d'un processeur voisin dans AnciennesDonnees

Dans toutes les applications que nous avons réalisées, nous avons suivi cet algorithme. Quelques variantes peuvent survenir afin de prendre en compte une spécificité d'un algorithme numérique. Pour davantage de détails, nous invitons le lecteur à consulter [13, 14, 15, 16, 17, 18, 26, 27, 28].

4.4 Détection de convergence

Dans la section précédente, nous avons abordé les modifications à entreprendre pour transformer un algorithme synchrone en un algorithme asynchrone IACA. Cependant un élément crucial n'a pas encore été détaillé. Il s'agit de la détection de convergence. Pour les algorithmes synchrones, la détection est effectuée en calculant

le résidu local avec la norme du max (voir équation 3.3) et ensuite on détermine le maximum des résidus locaux. Cette opération synchronise l'ensemble des processeurs. Concernant les algorithmes asynchrones, on ne peux pas procéder de la sorte.

Dans la littérature, on trouve différentes approches. Dans [38], les auteurs modifient le processus itératif en stoppant, dans certains conditions, les nœuds passés sous le seuil d'arrêt. Il en résulte que le processus peut ne pas converger dans ces conditions. On retrouve ce type d'inconvénient dans [104] où les auteurs utilisent également des communications FIFO. Dans [35], une étude permet de comparer les approches existantes. Les méthodes ont le désavantage de, soit modifier le processus itératif, soit d'utiliser un algorithme centralisé [46].

Dans un premier temps, nous avons utilisé une approche centralisée. Celle-ci présente l'avantage d'être assez simple à mettre en œuvre.

4.4.1 Approche centralisée

Un processeur centralise le résidu local de chaque nœud. De cette manière, le processeur centralisateur peut décider à quel moment l'ensemble du calcul est passé sous le critère d'arrêt. Pour appliquer cette approche, il faut cependant prendre garde au fait que le résidu du calcul associé à chaque processeur ne décroît pas forcément de manière monotone. Ceci s'explique par le fait que quand un processeur commence une nouvelle itération sans avoir reçu de message de mise à jour, son résidu peut devenir très faible, puis redevenir important par la suite, à la réception d'un message d'un voisin. Cela signifie qu'un nœud dont le résidu local s'approche du seuil d'arrêt peut osciller autour de ce seuil pendant plusieurs itérations. Afin de ne pas détecter une fausse convergence globale, un processeur doit compter le nombre de fois consécutives pour lequel il se trouve sous le seuil de convergence. Quand ce nombre devient égal à un nombre fixé par l'utilisateur en fonction du contexte (algorithme numérique et condition d'exécution), le nœud envoie un message au processeur qui centralise les convergences locales. Si, par la suite, le processeur passe à nouveau au dessus du seuil, il envoie immédiatement un message au nœud centralisateur. Le principe est donc d'informer le processeur centralisateur uniquement des changements d'états. Pour plus de détail, nous invitons le lecteur à consulter [13].

Comme évoqué dans tout ce document, les algorithmes centralisés ne sont clairement pas adaptés au contexte de grille de calcul. La centralisation pour la détection de convergence ne supporte pas le passage à l'échelle. Elle engendre des envois de messages entre des sites éloignés, et elle n'est pas applicable dès qu'on utilise un pare-feu (à moins d'utiliser un mécanisme de relayage). C'est pourquoi nous avons conçu un algorithme décentralisé pour la détection de convergence.

4.4.2 Approche décentralisée

Les contraintes que nous nous sommes fixées afin de concevoir un algorithme de détection de convergence décentralisé sont les suivantes :

- Il faut qu'on puisse appliquer notre mécanisme de détection de convergence sur tous les algorithmes IACA existants, c'est-à-dire sans modifier le processus de calcul.
- Un nœud ne communiquera qu'avec ses voisins proches afin que les configurations avec pare-feu soient utilisables.
- Dans la mesure du possible, l'algorithme ne doit pas envoyer trop de messages.

L'algorithme que nous avons développé reprend le principe de l'algorithme de l'élection de maître [105] ayant pour but de trouver ou d'élire un processeur maître dans une topologie sous forme d'arbre, c'est-à-dire sans cycle. Cet algorithme d'élection de leader est utilisé dans de nombreuses situations : terminaison d'algorithme distribué [1], synchronisation d'horloge dans une grille [114], algorithme distribué [5, 55], etc. Le principe de fonctionnement de l'algorithme est le suivant. Chaque nœud connaît uniquement ses voisins proches. Un nœud envoie un message à l'unique voisin qui ne l'a pas contacté. Quand un nœud reçoit un message, il identifie le voisin qui lui envoie le message, et à son tour, il teste s'il a un seul voisin qui ne l'a pas encore contacté. Quand un nœud n'a plus de voisin c'est lui le maître. Afin d'éviter les situations incohérentes, un système d'acquittement est mis en place.

Pour notre problématique de détection de convergence distribuée pour les algorithmes IACA, les conditions sont les suivantes :

- Un algorithme IACA s'exécute sur un ensemble de processeurs, et à partir d'un certain temps, le système passe sous le seuil de convergence.
- Un processeur peut détecter une convergence locale très tôt, en raison de l'absence de réception de message de la part de ses voisins, et par la suite, il peut pendant longtemps se trouver au dessus du seuil.

On considère que si l'ensemble des processeurs est passé sous le seuil avec, comme hypothèse, que les processeurs reçoivent régulièrement des mises à jour, alors on détecte la convergence. On notera que l'ensemble du système peut passer par la suite au dessus du seuil puisqu'un processus itératif ne décroît pas forcément de manière monotone ; mais ce problème affecte également les algorithmes séquentiels. Pour éviter ce désagrément, la solution consiste à choisir un critère d'arrêt plus petit.

La différence fondamentale entre le protocole d'élection de leader et notre système de détection de convergence réside dans le fait qu'un processeur, dans notre cas, peut détecter une convergence locale et, par la suite, revenir sur sa décision.

Comme pour la détection de convergence centralisée, un processeur envoie un message pour signaler qu'il a convergé localement, uniquement s'il est resté sous le seuil de convergence un nombre consécutif de fois ; ce nombre étant choisi par l'utilisateur.

Notre algorithme de détection de convergence peut être résumé par les points suivants :
- À l'initialisation, un nœud connaît l'ensemble de ses voisins, le critère d'arrêt et le nombre de fois qu'il doit rester sous ce critère avant d'informer un de ses voisins.
- Quand un processeur passe sous le critère d'arrêt, il compte le nombre de fois consécutif où il reste sous ce seuil.
- Après être resté un nombre de fois consécutif sous un seuil, un processeur considère qu'il a convergé localement.
- Lorsqu'un processeur a convergé localement et n'a qu'un seul voisin qui ne l'a pas informé de sa convergence, ce processeur informe ce voisin.
- À la réception d'un message de convergence locale, un nœud décrémente de un le nombre de ses voisins ne l'ayant pas informé de leurs convergences.
- Si un nœud passe au dessus du seuil, il remet à zéro le nombre de fois consécutif pour lequel il était sous le seuil. Si, de plus, il avait convergé précédemment et informé son unique voisin n'ayant pas convergé, il informe ce voisin de son changement d'état.
- Finalement lorsqu'un nœud a convergé et n'a pas de voisin à contacter (car ils ont déjà tous convergé), il a détecté la convergence globale.

Dans [19], nous présentons en détail cet algorithme avec la preuve qu'il va détecter la convergence.

4.5 Conclusion

Ce chapitre énonce les propriétés nécessaires afin de programmer efficacement un algorithme IACA. Par rapport à un algorithme synchrone, deux points essentiels sont à prendre en compte afin d'implémenter un algorithme IACA. Il s'agit, premièrement, de la gestion des communications qui ne doit pas perturber le calcul. Pour ce faire, l'utilisation d'une bibliothèque de communications multithreadées est indispensable. Le second point concerne la convergence. Dans ce chapitre nous proposons deux approches. La première utilise un processus qui centralise l'information sur un nœud. Comme nous l'avons déjà évoqué, la notion de centralisation est à éviter dans un contexte de grilles distantes car elle constitue un goulot d'étranglement. Pour corriger ce défaut, nous proposons une seconde approche décentralisée qui permet notamment de s'affranchir des pare-feu et qui utilise uniquement des communications entre des voisins proches.

Après avoir étudié la mise en œuvre des algoritmhes IACA, nous présentons, dans le chapitre suivant, les environnements de programmation que nous avons utilisés

pour développer ces algorithmes.

Chapitre 5

Environnements de programmation

Dans cette section, nous présentons les différents environnements de programmation parallèle que nous avons utilisés pour les expérimentations que nous avons menées. Comme évoqué dans le chapitre 4, la programmation des algorithmes IACA nécessite un environnement multithreadé. Concernant les communications, il est possible d'utiliser des émissions bloquantes et non bloquantes et il souvent pratique que les réceptions soient bloquantes.

Parmi ces conditions, la plus contraignante est certainement d'avoir un environnement multithreadé. En effet, de nombreuses bibliothèques de communications ne permettent pas actuellement de gérer les communications depuis des threads différents.

Dans la suite de ce chapitre, nous présentons les quatre environnements de programmation que nous avons utilisés. Pour chacun d'eux, nous expliquons succinctement sa provenance et quelques caractéristiques s'y rapportant.

- MPI/Mad [10, 93] est une version de MPI basé sur MPICH dans laquelle les communications sont effectuées par la bibliothèque de communication Madeleine [9]. La gestion des threads est assuré par Marcel. Pour le programmeur, cette version de MPI permet de créer un programme pouvant s'exécuter en mode synchrone et asynchrone.
- PM2 [99] a été élaboré avec deux bibliothèques appelées Madeleine et Marcel, c'est-à-dire avec les mêmes bibliothèques que MPI/Mad. En fait, PM2 a été développé initialement avant MPI/Mad. Nous avons beaucoup utilisé cet environnement, surtout au début.
- Corba [100] est un environnement initialement prévu pour développer des programmes distribués selon un mode de communication basé sur des objets, et non des messages. Néanmoins, la version libre OmniOrb 4.0 nous est apparue comme répondant parfaitement aux critères retenus pour développer nos algorithmes.
- Jace [29, 32] se distingue des trois précédents environnements par trois points essentiels. Il est écrit en Java alors que nous avons utilisés les trois autres avec du C ou C++ (Corba est également utilisable en Java). Jace est un environnement

conçu pour les algorithmes itératifs asynchrones. Finalement, dernier point et non des moindres, celui-ci a été conçu par Kamel Mazouzi, un doctorant de notre équipe.

Pour chacun d'entre eux, nous essayons par la suite d'en faire ressortir les points forts et les inconvénients.

5.1 MPI/Mad

L'environnement MPI [75] (Message Passing Interface en anglais) est incontestablement la bibliothèque la plus utilisée pour le calcul parallèle et distribué. En fait, MPI est une spécification[1] pour développer des programmes parallèles. Il existe de nombreuses implantations de MPI, libres ou développées par un constructeur informatique [91, 74, 33, 83, 106]. Cette spécification a subi des améliorations afin de répondre plus précisément aux attentes de la communauté des développeurs d'applications parallèles et distribuées. MPI2 [98] est la seconde version de la norme. Il n'existe pas, à l'heure actuelle, d'implantation qui respecte totalement cette nouvelle norme, car elle comprend de très nombreuses fonctions. La norme de MPI impose simplement aux développeurs d'une version MPI de proposer des fonctions aux utilisateurs. Elle n'impose pas de choix quant à l'implantation de ces fonctions ; ceci explique que des versions différentes de MPI puissent apporter des performances assez variées.

MPICH[74] est une version libre de MPI. Elle est utilisée par de nombreuses personnes en raison de sa stabilité et de ces bonnes performances. Actuellement MPICH n'est pas multithreadée. Pour la rendre utilisable sur des réseaux de nature diverse ayant des protocoles différents (TCP, Gigabit-Ethernet, Myrinet, GigaNet, SCI, ...), Raymond Namyst et son équipe ont remplacé la partie communication de MPICH par la bibliothèque Madeleine qu'ils ont développée. Par la même occasion, ils ont inclut leur bibliothèque permettant la gestion des threads : Marcel. Ainsi, la version de MPICH/Madeleine [10] offre la possibilité d'utiliser des protocoles différents et, nous concernant, permet d'effectuer des communications dans des threads différents.

Le principal avantage, de notre point de vue, de MPI/Mad, concerne la facilité de programmation. En effet, le passage d'une application synchrone utilisant des techniques de multidécomposition (ou multisplitting) (cf chapitres 6 et 7) à une application asynchrone est relativement facile et répétitif. Il nécessite de modifier la détection de convergence (cf chapitre 4.4.2) et la manière de gérer les communications afin de les effectuer dans des threads (sans modifier les messages échangés).

Au niveau performance, le couplage de Madeleine et de MPICH n'a pas la prétention de rivaliser avec les meilleures implantations de MPI. On peut cependant

[1] par abus de langage MPI est considéré comme un environnement de programmation alors que c'est une norme

noter que des tests[2] ont été réalisés afin de mesurer les débits sur différents types de réseaux.

5.1.1 MPI/Mad pour les IACA

Avec MPI/Mad, tous les threads sont créés par le programmeur. Concernant les communications du calcul, il est possible de créer un seul thread pour les envois et un seul thread pour les réceptions. Mais il est possible de créer autant de threads que de voisins, en émission et en réception. Dans ce cas, il est préférable d'avoir une application dans laquelle un nœud n'a qu'un nombre limité de voisins. Pour les applications ayant un schéma de communications $all - to - all$, une solution restreignant le nombre de threads paraît indispensable. Plus le nombre de threads est élevé, plus la politique de gestion des threads de l'ordonnanceur est coûteuse.

Pour programmer un algorithme IACA avec des communications flexibles, il est possible d'utiliser un message d'acquittement indiquant à l'émetteur que son message est traité par le récepteur. Dans ce cas, il est pratique d'utiliser un système de mutex garantissant qu'un seul message sera envoyé à un voisin particulier, à tout instant.

5.2 PM2

PM2 [99] a été développé principalement par Raymond Namyst et ses collaborateurs. L'objectif initial de PM2 est de fournir un cadre pour le développement d'applications parallèles irrégulières sur des architectures distribuées. Cette bibliothèque est composée de Marcel et Madeleine. Marcel est une bibliothèque de gestion efficace de threads. Elle est prévue pour supporter un grand nombre de threads sur une seule machine. Un mécanisme de migration de threads permet d'obtenir un degré de parallélisme élevé, ainsi que de l'équilibrage de charge dynamique. Madeleine, comme signalé dans la section précédente, a pour but d'offrir une interface de communication rapide et multi-protocole.

Une particularité intéressante de PM2 concerne les envois de messages qui sont effectués sans copie la plupart du temps. Au niveau performance, on évite l'utilisation de tampon mémoire ralentissant la latence et les débits des communications.

Les communications au sein d'une application PM2 sont basées sur le mode RPC (remote procedure call). L'émetteur choisit la fonction qui sera exécutée sur la machine distante ; celle-ci réceptionnera les données et appliquera le traitement de la fonction appelée. Ce principe de communication est légèrement différent de celui de MPI et requiert peu de temps d'adaptation de la part du programmeur pour une parfaite utilisation. Un thread scrute les messages arrivants et peut, selon le choix du programmeur, créer un thread pour traiter ce message. La gestion des communications

[2] http ://www-user.tu-chemnitz.de/$\tilde{d}$anib/cluster-benchmarks/

par un thread est recommandée pour les applications nécessitant des communications longues et coûteuses au niveau traitement.

5.2.1 PM2 pour les IACA

La programmation d'algorithmes IACA avec PM2 ressemble beaucoup à celle de MPI/Mad en raison de l'utilisation commune des bibliothèques Madeleine et Marcel. À part la gestion des communications qui utilise une syntaxe légèrement différente, seule la réception des messages gérée par un thread dans PM2 impose des modifications. Avec PM2 il est impossible de créer des threads de réceptions qui soient à l'écoute de certains messages.

Un mécanisme propre à PM2 permet de gérer simplement les acquittements de messages, il s'agit des *completions*.

Suivant le nombre de voisins à contacter lors d'une itération, il peut être nécessaire de regrouper les threads afin de réduire leur nombre.

5.3 Corba OmniOrb 4.0

Corba est également une norme conçue pour les applications basées sur le paradigme objet. Il existe de nombreuses implantations. Le mécanisme de communication de Corba est appelé ORB (object request broker). Les communications sont réalisées par des invocations de méthodes distantes. L'ORB possède la signature de tous les objets qu'on peut invoquer à distance. Corba est peu utilisé pour le calcul scientifique. Par contre, il peut se révéler intéressant pour coupler différents codes de calcul [102].

Comme Corba n'est pas prévu pour le calcul scientifique, il ne possède pas d'outil pour démarrer les tâches de calcul sur les processeurs. Concernant OmniOrb 4.0 nous avons écrit un outil pour effectuer ce travail. De même, Corba n'intègre pas de mécanisme pour savoir combien de nœuds se trouvent impliqués dans le calcul et il ne possède pas de fonction de synchronisations couramment utilisées avec les outils pour le calcul distribué.

5.3.1 OmniOrb 4.0 pour les IACA

La version OmniOrb 4.0 permet d'effectuer des envois par l'intermédiaire d'invocations des méthodes distantes. À l'invocation d'une telle méthode, OmniOrb crée un thread qui exécute la méthode invoquée. Ce mécanisme est similaire à un appel RPC pour PM2 transposé au monde de l'objet. On peut contrôler le nombre de threads pour les envois de données. Néanmoins, les threads utilisés pour la réception sont créés à la demande par le système en fonction des réceptions. Donc il n'est pas possible de les

contrôler directement[3].

5.3.2 JACE

Contrairement aux trois précédents environnements, JACE a été écrit pour développer des applications parallèles itératives asynchrones. L'abréviation signifie en anglais : Java Asynchronous Computing Environment. Afin de pouvoir comparer le gain apporté pour l'asynchronisme, il est également possible de développer des algorithmes synchrones. Le fait d'être écrit en Java le rend portable et plus facilement déployable. Le principe de fonctionnement de JACE est le suivant.

Un fichier de configuration contient l'ensemble des machines. Avec celui-ci JACE construit une machine virtuelle constituée des différentes ressources hétérogènes distantes. Cette tâche consiste à exécuter un *daemon* sur toutes les machines. Lorsqu'un utilisateur a fini de développer une application, il peut la lancer sur les machines grâce au *spawner*. À la fin du calcul, l'utilisateur peut récupérer le résultat de son application.

De manière similaire à PVM [69], JACE est composé de trois composants : la tâche, le *spawner* et le *daemon*. La thèse de Kamel Mazouzi [92] explique en détail l'architecture de JACE. La communication entre les processus est assurée par RMI [54]. Par rapport à une bibliothèque de communication standard, JACE est adapté au contexte des algorithmes asynchrones. En effet, il permet, d'une part, de dissocier le calcul des communications, grâce à l'utilisation de threads. D'autre part, il tient compte du fait que les algorithmes IACA, à chaque itération, n'ont besoin que de la dernière version d'un message. Dans la pratique, suivant la vitesse des processeurs, certains messages ne seront pas émis ou seront supprimés sur le récepteur. Cette fonctionnalité est possible par l'utilisation de files d'attente. À l'envoi et à la réception, un message étiqueté est placé dans une file d'attente. L'étiquette permet de le référencer. Si à l'émission, un autre message possède déjà cette étiquette, le nouveau message écrase l'ancien. Ainsi, l'ancien message ne sera jamais envoyé. Le même mécanisme se déroule à la réception. Lorsqu'un processeur appelle une fonction de réception en précisant l'émetteur et l'étiquette du message, suivant l'état de la file d'attente, plusieurs cas sont possibles. Si aucun message ne correspond, le processeur continue son calcul sans nouvelle information. S'il existe un message, il s'agit de la dernière version, puisque les versions précédentes ont été supprimées, automatiquement par JACE.

Un autre atout de JACE concerne le fait que le passage d'une version itérative synchrone à une version itérative asynchrone ne demande que peu de temps : il suffit de remplacer une variable à l'initialisation.

[3]La seule manière de les contrôler est de limiter le nombre de communications des processeurs émetteurs.

5.4 Avantages et inconvénients des environnements

Les différents environnements décrits précédemment possèdent des avantages et des inconvénients que nous allons détailler de la manière la plus objective qui soit. Dans [15, 18] nous présentons une comparaison de la programmation de deux algorithmes IACA avec MPI/Mad, PM2 et Corba et nous analysons les temps d'exécution des algorithmes. Pour pouvoir établir une comparaison, voici les critères qui nous ont semblé importants. Pour chacun d'eux, nous donnons une petite explication.

- La *portabilité* est un caractère important afin de pouvoir exécuter un programme sur des architectures différentes sans le recompiler. Sur ce sujet, seul JACE est portable, le fait d'être entièrement écrit en Java lui apportant les atouts de ce langage. Les trois autres environnements ne sont pas portables ; ils nécessitent une recompilation du programme et de la bibliothèque de communication pour chaque nouvelle architecture.
- La *facilité de déploiement* rend compte de la facilité que possède un utilisateur pour lancer une application sur de nombreux sites présentant des contraintes différentes (pare-feu, comptes utilisateurs différents). Il semble que JACE se montre le plus performant sur ce sujet. Ceci paraît logique puisqu'il a été conçu spécialement pour les algorithmes IACA sur grilles distantes. Les autres environnements ont été écrits pour atteindre des buts différents ; c'est pour cela qu'ils ne sont pas vraiment adaptés. Par exemple, PM2 et MPI/Mad requièrent un graphe complet entre les nœuds. Corba nécessite seulement que tous les processeurs puissent communiquer avec le processeur centralisateur. De plus, il est possible d'ajouter des processus en cours de calcul. Ce qui lui confère un avantage certain, même face à JACE.
- La *facilité d'écriture* d'un algorithme est, en général, assez liée au nombre de lignes de code. Pour la même raison que pour le point précédent JACE se distingue des autres bibliothèques, étant donné qu'elle a été écrite pour développer des algorithmes IACA. Pour les trois autres environnements, PM2 se place en tête devant MPI/Mad, et finalement Corba, car ce dernier demande du travail pour de nombreuses tâches, telles que le lancement des processus et les éventuelles synchronisations.
- La *gestion automatique des threads* permet à l'utilisateur de se dégager de cette contrainte qui peut se révéler source de blocage à l'exécution d'un programme. JACE, en tant qu'environnement dédié à l'asynchrone, prend encore l'avantage. Ensuite, on retrouve Corba qui permet la création d'un thread de manière automatique à l'invocation distante d'une méthode. PM2 est sensiblement identique à Corba en ce sens qu'un thread, créé automatiquement, scrute les messages des processeurs voisins, et appelle automatiquement la fonction adéquate pour le traitement du message. Finalement MPI/Mad n'offre aucun contrôle automatique des threads.

- La *gestion des mutex* permet de contrôler finement le déroulement des processus afin, par exemple, de programmer des sections critiques. Nous avons utilisé des mutex, par exemple, pour coupler l'équilibrage de charge avec les algorithmes IACA (cf. chapitre 8). MPI/Mad, Jace et PM2 offrent cette particularité, mais Corba ne le permet pas avec la version d'OmniOrb 4.
- La *possibilité d'écriture d'une version synchrone* paraît importante car il est beaucoup plus facile de développer, dans un premier temps, une version synchrone afin d'obtenir par la suite une version asynchrone. Dans certains cas, même si cette possibilité est offerte, elle n'est clairement pas facile à réaliser. MPI/Mad prend l'avantage sur ce point, car il a été conçu pour créer des applications synchrones. JACE le permet également, même si actuellement il n'existe pas de mécanisme de communications collectives. Avec PM2 et Corba, cette possibilité se trouve réduite et relève plus d'un défit.
- Le *passage simple d'une version synchrone à une version asynchrone* est important car une fois l'application synchrone développée, elle pourra être asynchrone sans gros effort. Avec JACE cette opération est prise en compte par le simple changement d'une valeur d'une variable. Avec MPI/Mad, le passage d'une version synchrone à une version asynchrone est relativement facile. Il suffit d'effectuer les communications dans des threads et de changer le principe de la détection de convergence. Avec Corba et PM2, cette question n'a pas vraiment de sens, puisque le développement d'une version synchrone n'est que difficilement envisageable.
- La *possibilité de contrôler l'ordonnanceur de processus légers* offre à l'utilisateur une finesse de programmation que certains n'apprécieront pas, mais qui, dans certains cas, peut s'avérer très intéressante. MPI/Mad et PM2 qui reposent sur la bibliothèque de gestion de threads, permettent de contrôler le gestionnaire d'ordonnancement. Corba ne le permet pas à notre connaissance. Pour JACE, le problème est légèrement différent. En fait, avec les machines virtuelles Java actuelles, les threads en Java ont un comportement "égoïste", c'est-à-dire qu'un thread qui a la main, la garde, tant que la tâche qu'il exécute n'est pas terminée. C'est un inconvénient non négligeable de Java et donc de JACE.
- L'*efficacité à l'exécution* est peut être finalement le point le plus intéressant. Concernant le calcul, PM2, MPI/Mad et Corba permettent d'exécuter des programmes écrit en C ou C++. De ce fait, les programmes sont rapides. Suivant le mode de communication choisi, le comportement du gestionnaire de threads, le nombre de threads s'il est réglable, la granuralité d'une application, le ratio calcul/communication, ces trois environnements s'avèrent plus ou moins similaires. JACE ne permet pas actuellement d'obtenir les mêmes performances, en raison du langage Java.

5.5 Conclusion

Ce chapitre a pour objectif de montrer que l'environnement de programmation pour développer un algorithme IACA, a certes une importance mais celle-ci n'est pas capitale. Il est, par contre, très pratique que la bibliothèque de communication soit multithreadée afin de dissocier le calcul des communications. Les environnements présentés, MPI/Mad, PM2, Corba et JACE permettent tous de développer des algorithmes IACA avec plus ou moins de facilité. Ceci dit, la méthodologie que nous recommandons est d'abord de réaliser une application synchrone avant de développer une application asynchrone. Si l'environnement de programmation n'est pas adapté à l'écriture d'algorithmes synchrones, ce n'est pas spécialement problématique. Nous avons plusieurs fois écrit une application synchrone en MPI, qui nous a permis par la suite d'obtenir une version asynchrone avec PM2 ou Corba. Le passage du synchrone à l'asynchrone est relativement aisé, étant donné que seuls deux points essentiels doivent retenir l'attention du développeur : la gestion des communications et la détection de convergence.

Les deux chapitres suivants décrivent la mise en œuvre des méthodes de multidécomposition (ou multisplitting) pour résoudre respectivement un système linéaire et un système non linéaire. Ces méthodes permettent de concevoir des algorithmes à gros grains.

Chapitre 6

Multidécomposition pour système linéaire

6.1 Introduction

Les techniques de multidécomposition (ou multisplitting) présentent des points communs avec les algorithmes basés sur la décomposition de domaine. Ces derniers décomposent le modèle physique en différentes régions reliées par des interfaces. On associe à chaque processeur un domaine et les communications ont lieu aux différentes interfaces. En général, on découpe le modèle physique de manière à minimiser les échanges aux interfaces. Les méthodes de multisplitting ont pour objectif de découper un problème, linéaire ou non linéaire, afin de pouvoir résoudre ce problème sur plusieurs processeurs. De plus, elles permettent de prouver que la résolution itérative du problème en parallèle sera identique à la résolution du même problème en séquentiel, modulo un seuil de précision choisi. Les méthodes de multisplitting ont l'avantage de permettre le recouvrement d'une partie des données, c'est-à-dire que certaines données sont calculées sur plusieurs processeurs. Ceci permet de réduire le nombre d'itérations et donc éventuellement le temps d'exécution.

Dans ce chapitre, notre objectif est d'expliquer comment mettre en œuvre les techniques de multisplitting afin de résoudre des systèmes linéaires. Nous avons appliqué ces techniques afin d'élaborer un outil pour résoudre des systèmes linéaires, également appelé solveur par anglicisme. Pour résoudre des systèmes linéaires, il existe deux catégories de solveurs, les solveurs directs et les solveurs itératifs. Dans ce travail, nous avons choisi de proposer une technique de parallélisation facile à mettre en œuvre pour résoudre des systèmes linéaires, basée sur l'utilisation de solveurs linéaires LU. La méthode LU permet de résoudre un système linéaire en utilisant deux étapes. La première consiste à factoriser le système en deux matrices triangulaires (l'une inférieure et l'autre supérieure). Cette partie est longue en terme de calcul

et requiert des efforts de programmation pour être efficace[1] [52]. La seconde étape consiste à résoudre le système et cette partie est facile une fois la matrice originale factorisée.

Comme évoqué dans tout notre travail, les méthodes itératives sont très utiles pour résoudre les systèmes creux, que l'on rencontre très souvent dans la pratique. La parallélisation des méthodes directes que nous proposons dans ce chapitre est applicable avec les mêmes conditions que pour les méthodes itératives.

Dans la suite de ce chapitre nous détaillons l'algorithme appelé multisplitting-LU. Nous expliquons succinctement les conditions de convergence. Finalement nous commentons les résultats d'expérimentations.

6.2 Algorithme multisplitting-LU

Pour résoudre un système linéaire, la méthode LU est certainement la méthode directe la plus employée. Il existe de nombreuses références à ce sujet, en voici quelques unes [80, 3, 86]. La dernière référence explique dans les détails le fonctionnement de base de l'algorithme, alors que les autres papiers proposent des versions parallèles optimisées pour les matrices creuses. Il existe de nombreuses implantations de cet algorithme séquentiel ou parallèle. L'algorithme que nous proposons fonctionne aussi bien en version synchrone qu'asynchrone. Seules les parties concernant les communications et la détection de convergence sont sujettes à modifications. À partir de cet algorithme, il est possible de construire plusieurs variantes selon le mode de synchronisation choisi.

Soit un système linéaire de dimension n

$$Ax = b \tag{6.1}$$

Supposons que l'équation (6.1) admette une unique solution. Notre approche consiste à découper la matrice en bandes horizontales. Chaque bande est assignée à un processeur comme sur la figure 6.1. Avec une telle distribution un processeur connaît le décalage qu'il doit appliquer pour connaître le début des données qui le concerne. Ce décalage est utilisé pour déterminer la sous-matrice, notée *ASous*, dont chaque processeur a la charge. La partie avant cette sous-matrice représente les dépendances de gauche, nous l'appelons *DepGauche*. Similairement, *DepDroite* représente les dépendances de droite après la sous-matrice *ASous*. On définit également les sous-vecteurs *XSous* et *BSous* comme les parties compatibles avec *ASous* de x et b.

L'algorithme 6.1 décrit notre solveur basé sur les techniques de multisplitting, appliqué à l'algorithme LU. Par la suite, nous appelons cet algorithme multisplitting-LU.

[1] surtout pour les systèmes linéaires creux pour lesquels on souhaite optimiser l'occupation mémoire

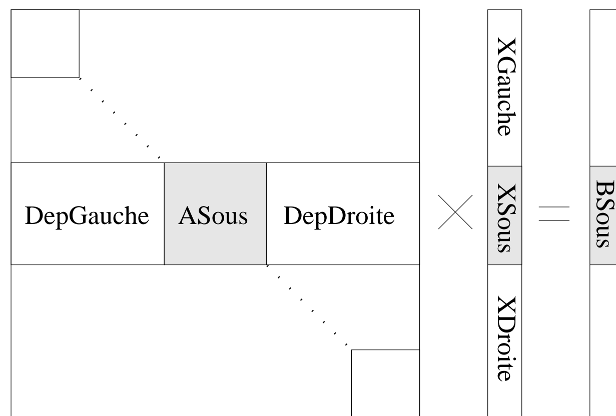

FIG. 6.1 – Décomposition de la matrice et des vecteurs

À chaque itération, un processeur résoud :

$$Asous * XSous = BSous - DepGauche * XGauche - DepDroite - XDroite \quad (6.2)$$

Les quatre étapes principales de l'algorithme sont les suivantes :

1. **Initialisation**
   Le chargement de la matrice n'est pas fixé. Soit un processeur s'occupe de charger toute la matrice, et ensuite il envoie les bandes de matrices aux différents processeurs, soit chaque processeur charge directement la matrice bande qui le concerne (dans l'algorithme la matrice bande correspond à $DepGauche + ASous + DepDroite$). Ensuite les processeurs exécutent les itérations suivantes jusqu'à la détection de convergence.
2. **Calcul**
   À chaque itération, un processeur calcule la partie droite de (6.2) qu'il stocke dans une variable appelée $BLoc$ de la manière suivante $BLoc = BSous - DepGauche * XGauche - DepDroite * XDroite$. Puis il détermine $XSous$ avec la fonction $LU(ASous, BLoc)$.
3. **Échange de données**
   Chaque processeur envoie ses dépendances à ses voisins. La fonction *Morceau* envoie uniquement la partie des données de $XSous$ utile au processeur i. Dans la version synchrone avec un algorithme ISCS ou ISCA (voir la section 3.2.2), les réceptions sont prises en charge directement dans le code et dans la version asynchrone avec un algorithme IACA (cf. section 4.3) elles sont effectuées par un autre thread. C'est pour cela que nous ne faisons pas apparaître les réceptions dans l'algorithme. Néanmoins, lorsqu'un processeur reçoit une partie de la solution (noté $XSous$) d'un de ses voisins, il met à jour le vecteur $XDroite$ ou $XGauche$ en fonction du rang de l'émetteur.

4. **Détection de convergence**
   Pour la version synchrone, elle est entreprise comme dans la section 3.2.2. Concernant les algorithmes IACA, il est possible de mettre en œuvre la version centralisée ou décentralisée, toutes deux décrites dans la section 4.4.

Le modèle algorithmique permettant de prouver la convergence est introduit dans [20]. Avec ce modèle, nous montrons la convergence en mode synchrone et en mode asynchrone. Les conditions de convergence sont les suivantes :
– Le rayon spectral[2] de la matrice d'itération associé à chaque processeur est strictement inférieur à 1 pour la version synchrone.
– Le rayon spectral de la valeur absolue de la matrice d'itération associé à chaque processeur est strictement inférieur à 1 pour la version asynchrone.

Ces conditions sont classiques pour les algorithmes itératifs concernant les systèmes linéaires, si l'on considère le système dans sa globalité [38]. Dans [20], nous énonçons les classes de matrices pour lesquelles notre solveur est utilisable et donnons une preuve de convergence. Dans le présent document, nous donnons uniquement les résultats. Si la matrice du système est diagonale dominante, alors la convergence est assurée, il en est de même pour les Z matrices. Ce sont des matrices pour lesquelles les éléments hors diagonale sont négatifs ou nuls. Ces matrices interviennent dans les domaines de la physique, de la biologie ou des sciences sociales [103, 37].

Remarque 3 *La décomposition de la matrice en bandes n'implique pas que les bandes sont disjointes. Lorsque celles-ci ont des parties communes, on parle alors de recouvrement, une partie des données est calculée par au moins deux processeurs.*

Remarque 4 *La première étape des solveurs directs consiste à factoriser la matrice, une étape coûteuse. Dans notre cas, la factorisation est entreprise uniquement à la première itération sur des matrices de tailles inférieures.*

Après avoir étudié cet algorithme et ses variantes, analysons son comportement dans un contexte de grille.

6.3 Expérimentations

Dans la suite de cette étude, nous avons comparé un algorithme direct permettant de résoudre un système linéaire creux en parallèle, SuperLU, avec nos solveurs, également basés sur SuperLU [73, 89]. Nous avons choisi cette bibliothèque car elle permet de résoudre des systèmes linéaires creux, non symétriques, sur des architectures hautes performances. Il existe d'autres bibliothèques possédant ces caractéristiques, néanmoins SuperLU est bien connue [4, 76] ; elle est considérée comme performante

[2] plus grande valeur propre en valeur absolue

Algorithme 6.1 algorithme solveur multisplitting-direct

Initialisation de l'interface de communication
MonRang = *Rang du processeur*
NbProcs = *Nombre de processeurs*
Taille= *Taille de la matrice*
TailleSous = *Taille de la sous matrice*
Deca = *Décalage de la sous matrice*
ASous[TailleSous][TailleSous] = *Sous matrice*
DepGauche[TailleSous][Deca] = *Sous matrice des dépendances gauche*
DepDroite[TailleSous][Taille-Deca-TailleSous] = *Sous matrice des dépendances droite*
DependDeMoi[NbProcs] = *Tableau des processeurs dépendants*
BSous[TailleSous] = *Tableau avec la valeur de B du sous système*
XSous[TailleSous] = *Tableau avec la solution X du sous système*
XGauche[Deca] = *Tableau avec la solution gauche du système*
XDroite[Taille-Deca-TailleSous] = *Tableau avec la solution droite du système*
BLoc[TailleSous] = *Tableau avec des valeurs locales de B*
Initialisation de toutes les variables
repeat
  BLoc = BSub
  if MonRang !=0 **then**
    BLoc = BLoc-DepGauche*XGauche
  end if
  if MonRang !=NbProcs-1 **then**
    BLoc = BLoc-DepDroite*XDroite
  end if
  XSous = LU(ASous,BLoc)
  for i=0 jusqu'à NbProcs-1 **do**
    if i !=MonRang et DependDeMoi[i] **then**
      Envoi(Morceau(XSous,i),i)
    end if
  end for
  Détection de convergence
until Convergence globale détectée

même si elle n'est pas forcément la plus rapide du marché. La bibliothèque SuperLU utilise une décomposition LU avec un pivot partiel, la résolution du système triangulaire est effectuée au choix par des substitutions avant ou arrière. Il existe trois versions de cette bibliothèque : une version séquentielle, une version pour les architectures parallèles à mémoire partagée, et enfin une version pour les architectures distribuées avec la bibliothèque MPI.

Avec notre approche présentée dans la section précédente, nous avons développé deux solveurs parallèles. L'un fonctionne avec des itérations synchrones et utilise MPI pour les communications. L'autre est basé sur le modèle IACA et il est implanté avec Corba. Ces deux solveurs utilisent la version séquentielle de SuperLU[3].

Ces expérimentations ont été entreprises pour mettre en évidence les différentes propriétés de ces algorithmes. Tout d'abord, nous avons étudié le comportement des trois algorithmes dans une situation de passage à l'échelle. Ensuite, nous avons comparé les trois programmes avec des matrices ayant des propriétés différentes dans un contexte hétérogène au niveau des processeurs, avec un réseau local et deux sites distants. Pour le contexte distant, nous avons utilisé uniquement deux sites car nous n'avons pas pu, pour des raisons de sécurité, bénéficier de plus de sites. Il est en effet fréquent que les sites de calculs soient protégés par des pare-feu, et les versions de MPI et Corba que nous avons utilisées ne permettent pas de franchir les pare-feu. C'est pourquoi nous avons choisi de mesurer l'impact des solveurs face à des perturbations volontaires du réseau, afin d'étudier la robustesse des algorithmes. Finalement, nous avons mis en évidence l'impact sur les performances du recouvrement permis par les méthodes de multisplitting.

Cinq matrices ont été utilisées à cet effet. Tout d'abord nous avons choisi une série de trois matrices, appelées *cage10.rua*, *cage11.rua* et *cage12.rua*. Elles sont disponibles dans la collection des matrices creuses de l'Université de Floride [51]. Ces trois matrices sont issues d'un modèle d'électrophorèse d'ADN. Leur degré respectif est 11397 (ie. 11397*11397 éléments), 39082 et 130228. Afin d'obtenir d'autres types de matrices, nous avons développé un générateur de matrices diagonales dominantes et nous avons généré deux matrices. La première est de degré 500000 et la seconde de degré 100000. La seconde a été choisie spécialement pour mesurer l'influence du recouvrement, c'est pourquoi le rayon spectral de sa matrice d'itération après décomposition est très proche de 1, tout en étant légèrement inférieur.

Trois configurations de grappes ont étudiées :

- une grappe locale homogène , appelée grappe1, composée de 20 machines Pentium IV 2.6Ghz avec 256Mo de mémoire. Le réseau est un réseau éthernet 100Mb standard.
- une grappe locale hétérogène, appelée grappe2, composée de 8 machines, dont la puissance varie entre le Pentium IV 1.7Mhz au Pentium IV 2.6Mhz avec 512Mo

[3]Ils utilisent la version 3.0 (séquentielle) alors que la version distribuée de SuperLU utilisée à titre de comparaison est la version 2.0.

de mémoire. Le réseau est également un réseau éthernet 100Mb standard.
- une grappe distante hétérogène, appelée grappe3, composée de 10 machines dispersées sur deux sites. Le réseau intra-site est un réseau éthernet 100Mb standard et la connection entre les deux sites est réalisée par une fibre optique 20Mb. La configuration des machines varie du Pentium IV 1.7Mhz au Pentium IV 2.6Mhz avec 512Mo de mémoire.

Dans la suite, les résultats obtenus sont les moyennes d'une série de 10 exécutions. Les temps sont exprimés en secondes. La précision a été fixée à 10^{-8}. Pour les solveurs basés sur les techniques de multisplitting, nous faisons apparaître le temps de factorisation. Ce temps est identique pour les versions synchrone et asynchrone.

6.3.1 Expérimentations avec une grappe homogène locale

Les tableaux 6.1 et 6.2 résument les accélérations des trois algorithmes considérés par la suite : la version distribuée de SuperLU et notre algorithme appelé multisplitting-LU en version synchrone et asynchrone. La version distribuée de SuperLU fournit des accélérations relativement bonnes jusqu'à 10 processeurs dans le premier tableau et jusqu'à 20 processeurs dans le second. Cependant, les deux versions de l'algorithme multisplitting LU sont incontestablement beaucoup plus rapides, la factorisation étant la partie la plus longue du calcul. La matrice utilisée dans le second tableau nécessite trop de mémoire pour pouvoir être traitée avec moins de 4 processeurs. Ces expérimentations mettent aussi en évidence que les versions synchrone et asynchrone de notre algorithme ont sensiblement le même comportement sur une grappe locale homogène, avec un léger avantage pour la version synchrone lorsque le ratio calcul/communication diminue. Nous pensons que la version asynchrone serait avantagée dans un contexte possédant un très grand nombre de machines. Les mauvaises performances de la version asynchrone ayant 16 et 20 processeurs sont dûes à deux raisons. La première est liée à la détection de convergence qui prend davantage de temps lorsque le nombre de processeurs augmente. La seconde raison provient du fait que le nombre d'itérations augmente considérablement lorsque les parties de calculs deviennent négligeables. En conclusion, la version asynchrone tire profit, de notre point de vue, des situations pour lesquelles le nombre de processeurs est important et le ratio de calcul par rapport aux communications n'est pas négligeable.

6.3.2 Expérimentations sur grappes hétérogènes (locale et distante)

Le tableau 6.3 reporte les temps d'exécution des trois algorithmes avec trois matrices différentes. Les gains de performances sont encore plus accentués que précédemment, ils illustrent pleinement l'adéquation de nos algorithmes au contexte de grille. En outre, comme précédemment, on peut remarquer que l'étape de factorisation nécessite un temps important. Dans le contexte distant, la version asynchrone

nombre de processeurs	SuperLU distribué	multisplitting-LU synchrone	multisplitting-LU asynchrone	temps de factorisation
1	157.63s	-	-	-
2	89.27s	34.15s	33.38s	32.61s
3	69.24s	19.14s	19.90s	18.26s
4	50.32s	8.43s	8.05s	7.82s
6	39.77s	2.14s	2.16s	1.84s
8	34.34s	1.05s	1.04s	0.84s
9	30.77s	0.60s	0.60s	0.45s
12	33.36s	0.29s	0.36s	0.19s
16	33.71s	0.20s	1.05s	0.11s
20	45.99s	0.14s	1.84s	0.06s

TAB. 6.1 – Expérimentations avec grappe1 pour caractériser le support de la mise à l'échelle des algorithmes SuperLU distribués et multisplitting-LU avec la matrice cage10.rua

nombre de processeurs	SuperLU distribué	multisplitting-LU synchrone	multisplitting-LU asynchrone	temps de factorisation
4	1496.28s	131.69s	131.45s	126.78s
6	949.20s	44.29s	44.17s	41.73s
8	762.76s	12.44s	12.25s	11.09s
9	679.17s	11s	11s	9.91s
12	540.49s	3.77s	3.78s	3.16s
16	456.54s	1.24s	2.34s	0.71s
20	471.70s	1.01s	2.03s	0.30s

TAB. 6.2 – Expérimentations avec grappe1 pour caractériser le support de la mise à l'échelle des algorithmes SuperLU distribués, et multisplitting-LU avec la matrice cage11.rua

prend légèrement l'avantage face à la version synchrone.

Un autre point essentiel à considérer concerne la mémoire nécessaire pour le calcul. Les algorithmes directs de résolution de systèmes linéaires consomment beaucoup plus de ressources mémoires que les deux versions de multisplitting-LU. Cette caractéristique est mise en évidence avec la matrice cage12, pour laquelle la grappe3 a échoué par manque de mémoire, alors que nos algorithmes n'ont pas rencontré ce problème. De plus, nous n'avons pas réussi à faire apparaître l'accélération avec la version séquentielle de SuperLU, car même avec la matrice cage11 (la plus petite des trois), une machine mono-processeur disposant d'un 1Go de mémoire n'arrive pas à la

résoudre.

matrice	grappe utilisé	SuperLU distribué	multisplitting-LU synchrone	multisplitting-LU asynchrone	temps de factorisation
cage11	grappe2	1212s	12.7s	12.1s	11s
cage12	grappe3	pam	441.5s	441.2s	430.3s
matrice 500000	grappe3	15145s	17.44s	15.76s	4.05s

TAB. 6.3 – Comparaison des trois solveurs avec grappe2 et grappe3 (pam signifie pas assez de mémoire)

6.3.3 Impact de la charge du réseau

Pour mesurer la robustesse des algorithmes face à une dégradation des performances du réseau, nous avons choisi de le perturber volontairement en ajoutant des tâches générant des communications entre les deux sites. Nous avons utilisé iperf [107] pour perturber le réseau. Cet outil permet de mesurer le débit de communications entre deux machines. Nous avons placé cet outil sur deux machines non utilisées pour le calcul sur chacun des sites. En réglant le nombre de threads utilisés par iperf, on agit sur les perturbations du réseau. Il faut toutefois noter que le nombre de threads utilisés n'interfère absolument pas de manière linéaire sur le débit du réseau, iperf étant perturbé par l'exécution d'un algorithme et réciproquement. De plus, il faut prendre également en compte que certaines autres tâches se sont probablement exécutées en même temps (serveur ftp, mail, web, ...), même si ce trafic est souvent négligeable.

Le tableau 6.4 met en évidence la robustesse de l'algorithme multisplitting-LU face à la dégradation des communications : même en cas de forte dégradation de la bande passante, les performances ne sont que très légèrement perturbées. C'est pourquoi nous pouvons affirmer que nos algorithmes requièrent peu de bande passante par rapport à la version distribuée de SuperLU. En cas de forte perturbation, la version asynchrone de multisplitting-LU se comporte mieux que la version synchrone.

6.3.4 Influence du recouvrement

Les techniques de recouvrement de données permettent d'accélérer la convergence d'un algorithme numérique en réduisant le nombre d'itérations. Au niveau informatique, cette réduction peut se traduire par un gain de temps. Dans notre cas, cela se traduit par une diminution du nombre d'itérations, mais aussi par une augmentation du temps de factorisation qui, nous l'avons vu précédemment, est un facteur important.

nombre de threads perturbateurs	SuperLU distribué	multisplitting-LU synchrone	multisplitting-LU asynchrone
0	15145s	17.44s	15.76s
1	18321s	33.50s	18.60s
5	20296s	63.4s	29.33s
10	22600s	99.35s	44.13s

TAB. 6.4 – Impact de la charge du réseau avec grappe3 et la matrice générée 500000 (les perturbations ont été effectuées avec iperf)

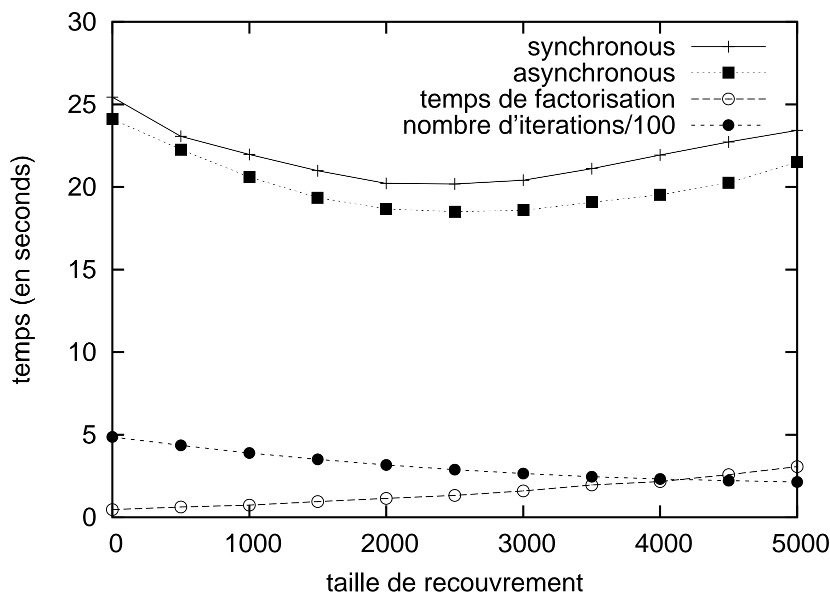

FIG. 6.2 – Impacts du recouvrement avec grappe3 et la matrice générée 100000

La figure 6.2 montre l'impact du recouvrement sur les versions synchrones et asynchrones de l'algorithme multisplitting-LU. La matrice a été générée de telle sorte que le rayon spectral de sa matrice d'itérations soit très proche de 1 ; c'est pourquoi le nombre d'itération pour atteindre la convergence est beaucoup plus important que pour les autres matrices. Plus le recouvrement est important, plus le temps de factorisation est conséquent. Il faut donc définir une taille de recouvrement en fonction de la taille de la matrice et en fonction de divers paramètres tels que le temps de factorisation, le rayon spectral de la matrice d'itération, etc. Sur la figure 6.2, nous avons indiqué le nombre d'itérations pour la version synchrone. Ce temps est divisé par 100 pour avoir une échelle commune. Sur notre exemple, le meilleur choix est une taille de recouvrement égale à 2500. Nous n'avons pas fait apparaître le nombre d'itérations pour la version asynchrone car ce nombre varie selon les processeurs (puisqu'ils sont hétérogènes) et selon les exécutions. Cependant, comme les itérations se déroulent rapidement, le

nombre d'itérations dans la version asynchrone est systématiquement plus important que pour la version synchrone.

6.4 Conclusion

Ce chapitre présente une nouvelle approche pour paralléliser un algorithme direct, adaptée au contexte de la grille. L'algorithme consiste à découper la matrice initiale en bandes afin que chaque processeur impliqué dans le calcul, s'occupe d'une bande et résolve une partie du vecteur solution. Le gain apporté par l'algorithme provient du fait que la complexité de factorisation est extrêmement réduite avec le découpage. De plus, le fait d'effectuer des itérations afin de converger vers la solution n'est finalement pas coûteux au regard du coût de la factorisation de la matrice entière requis par l'algorithme LU standard.

Dans [20], nous montrons que notre approche englobe certains algorithmes connus tels que la méthode d"'O'Leary et White" et la méthode "Schwarz alternée", même si ces approches n'ont pas été définies pour paralléliser des algorithmes directs de résolution de matrices. De plus, nous montrons dans ce papier que l'algorithme converge vers la solution du problème en version synchrone et asynchrone.

Les expérimentations entreprises montrent clairement que notre approche est adaptée au contexte distribué et plus spécialement aux grilles de calcul. La classe d'application est large mais ne couvre pas la totalité des applications scientifiques. La version synchrone de notre algorithme est efficace dans un contexte de grappes locales homogènes alors que la version asynchrone se montre plus robuste lorsque les débits de communications sont variables et soumis à des perturbations imprévisibles. De plus, nous avons étudié divers aspects importants comme le temps de factorisation, la taille du recouvrement, la mémoire requise et l'impact des perturbations du réseau sur le déroulement des algorithmes.

Il paraît intéressant par la suite de poursuivre ces travaux afin d'appliquer cette approche sur d'autres solveurs directs. Le couplage de plusieurs solveurs directs, un différent par processeur, semble également une piste intéressante à étudier. Une autre piste consiste à essayer d'appliquer les techniques de renumérations et de préconditionnements.

Dans le chapitre suivant, nous étudions les méthodes de multisplitting pour des problèmes non linéaires.

Chapitre 7

Multidécomposition pour système non linéaire

7.1 Introduction

Cette partie introduit une méthode de multidécomposition (ou multisplitting) pour résoudre un système non linéaire avec la méthode des différences finies. La méthode de Newton permet la résolution de tels systèmes. L'approche que nous proposons utilise la méthode de Newton, conjointement avec une méthode de multisplitting, ainsi nous l'appelons multisplitting-Newton.

Les problèmes visés sont les systèmes d'Équations aux Dérivées Partielles (EDP). Ils interviennent très fréquemment dans la modélisation de nombreux phénomènes complexes physiques ou chimiques, citons par exemple la mécanique Newtonienne et quantique, la thermodynamique, les problèmes d'écoulement, d'ondes, d'électromagnétisme, de pollution, les problèmes financiers, ... La résolution d'un système d'EDP est possible avec la méthode des éléments finis et la méthode des différences finies. La première méthode, la plus générale pour résoudre ce type de problème, permet de traiter des problèmes pour lesquels la géométrie est complexe. Cependant, pour les problèmes dont le domaine est géométrique, la méthode des différences finies s'avère très efficace. Par ailleurs, elle présente l'avantage d'être beaucoup plus facile à mettre en œuvre que la méthode des éléments finis.

À partir d'un système d'EDP non stationnaires et après discrétisation du temps, la méthode des différences finies conduit à l'obtention d'un système d'Équations Différentielles Ordinaires (EDO). La résolution d'un tel problème en parallèle est possible selon [87] par deux approches :

- Le parallélisme de la méthode. Cette approche consiste à paralléliser la méthode de résolution. Dans ce cas, à chaque pas de temps, le système est résolu globalement par la parallélisation de la méthode de résolution, entraînant ainsi des communications à chaque pas de temps.

– Le parallélisme du système. Cette approche consiste à découper le problème et à le traiter localement sur chaque processeur, dans sa totalité sur l'espace temps, sans communication à chaque pas de temps. La convergence vers la solution nécessite bien évidemment plusieurs itérations sur l'ensemble de l'intervalle de temps. Les communications interviennent uniquement à la fin d'une itération, c'est-à-dire à la fin de l'intervalle de temps. Cette méthode porte aussi le nom de relaxation d'ondes car les composantes du systèmes sont traitées sur tout l'espace temps avant que les communications ne soient entreprises.

Dans la suite de ce chapitre nous considérons uniquement l'approche qualifiée de parallélisme de la méthode, utilisée aujourd'hui majoritairement pour résoudre un système d'EDO.

Précédemment, nous avons volontairement omis le mode de discrétisation du temps. On distingue deux modes de discrétisations : une discrétisation explicite et une discrétisation implicite. Selon le phénomène étudié, chacune de ces méthodes possède des avantages et des inconvénients. Cependant, sur les problèmes que nous avons étudiés, une discrétisation implicite est préférable. Elle conduit dans ce cas, à chaque pas de temps, à la résolution d'un système non linéaire.

La méthode de Newton est une méthode itérative qui permet de résoudre un système non linéaire. Elle "linéarise" le problème à chaque itération. Ainsi, à chaque itération, on est amené à résoudre un système linéaire.

Deux approches sont envisageables pour paralléliser la méthode de Newton. La méthode la plus courante utilise un algorithme parallèle, direct ou itératif pour résoudre le système linéaire. Il est également possible d'utiliser la méthode de multisplitting présentée dans le chapitre précédent. L'inconvénient majeur de cette solution est de synchroniser les itérations de Newton après la résolution du système linéaire, même si on utilise une méthode asynchrone pour le résoudre. C'est pourquoi une telle approche n'est pas nécessairement adaptée au contexte des grilles.

Une autre solution consiste à utiliser les techniques de multisplitting qui permettent de "découper" le problème initial en plusieurs problèmes. Les avantages sont multiples :

– Chaque processeur utilise un solveur séquentiel pour résoudre son système linéaire.
– Dans la version asynchrone de la méthode, il n'y a plus de synchronisation dans le cas stationnaire. Dans le cas d'un système non stationnaire, il y a une seule synchronisation à chaque pas de temps.
– On peut bénéficier des techniques de recouvrement pour accélérer les calculs.

Dans la suite, nous expliquons l'algorithme et sa mise en œuvre. Puis nous décrivons nos expérimentations.

7.2 Algorithme de multisplitting-Newton

L'algorithme que nous présentons permet de résoudre un problème non linaire non stationnaire, c'est-à-dire dans lequel le temps intervient. Si le problème est stationnaire, le temps n'intervient pas ; la modélisation est légèrement plus simple. La plupart des problèmes que nous avons traités sont dépendants du temps, c'est pourquoi nous préférons présenter cette version.

Soit un système d'EDP, après discrétisation de l'espace avec la méthode des lignes (MOL : method of line), on obtient un système d'EDO de la forme :

$$\frac{dy(t)}{dt} = f(y(t), t) \tag{7.1}$$

Suivant le problème, il existe des conditions aux limites du domaine considéré. Elles sont généralement de la forme de Dirichlet si la condition porte sur y, ou de Neumann si la condition porte sur la dérivée de y. Pour résoudre cette équation, il faut approximer la partie gauche de l'équation. La méthode d'Euler est la plus simple pour effectuer cette tâche. D'autres méthodes sont également utilisables ; citons les méthodes de Crank-Nicolson, de Runge Kutta, d'Adams, la méthode BDF ou l'algorithme du prédicteur correcteur. Il existe de nombreux ouvrages à ce sujet, le lecteur intéressé est invité à consulter par exemple [78, 8, 41, 8, 44, 48, 97, 101].

Selon le mode de discrétisation du temps, on obtient un schéma d'Euler implicite ou explicite. Le premier retiendra notre attention car il permet d'utiliser des pas de temps plus gros qu'avec un schéma explicite. Après approximation de la dérivée par la méthode d'Euler implicite, l'équation (7.1) est de la forme :

$$\frac{y(t+h) - y(t)}{h} = f(y(t+h), t+h) \tag{7.2}$$

Cette équation est implicite car le terme $y(t+h)$ intervient des deux cotés de l'équation. Avec un schéma explicite la partie droite de l'équation est de la forme $f(y(t), t)$, et dans ce cas, il est facile de déterminer $y(t+h)$. Pour déterminer la solution de (7.2), on utilise la méthode de Newton, mais auparavant il est pratique de réécrire cette équation en :

$$F(y(t+h), y(t), t) = y(t) + h * f(y(t+h), t+h) - y(t+h) \tag{7.3}$$

La méthode de Newton permet de déterminer quand une fonction s'annule. Nous l'utilisons pour déterminer quand $F(y(t+h), y(t), t) = 0$, ce qui nous donnera la solution de l'équation (7.3) et donc de (7.2). La méthode de Newton étant itérative, il faut initialiser la méthode avec une valeur approchée de la solution, prenons par exemple $y(t+h)^0 = y(t)$. En prenant k pour représenter les itérations, la méthode de Newton s'écrit alors :

$$y(t+h)^{k+1} = y(t+h)^k - [DF(y(t+h)^k, y(t), t)]^{-1} * F(y(t+h)^k, y(t), t) \tag{7.4}$$

La matrice $DF(y(t+h)^k, y(t), t)$ doit être inversible, il s'agit de la Jacobienne de la fonction F. En posant $\delta y(t+h)^{k+1} = y(t+h)^{k+1} - y(t+h)^k$, on peut alors réécrire (7.4) en :

$$DF(y(t+h)^k, y(t), t) * \delta y(t+h)^{k+1} = -F(y(t+h)^k, y(t), t) \tag{7.5}$$

La méthode de quasi-Newton permet d'économiser le calcul de $DF(y(t+h)^k, y(t), t)$ à chaque itération en prenant uniquement $DF(y(t+h)^0, y(t), t)$. En renommant $DF(y(t+h)^0, y(t), t)$ en DF, $\delta y(t+h)^{k+1}$ en δ^{k+1} et $F(y(t+h)^k, y(t), t)$ en $F(y^k)$, on obtient :

$$DF * \delta^{k+1} = -F(y^k) \tag{7.6}$$

Ainsi, l'algorithme séquentiel de quasi-Newton 7.1 consiste, à chaque pas de temps, à déterminer la Jacobienne DF à la première itération et puis à itérer en déterminant δ^{k+1}, la solution du système linéaire (7.6) qui permet de mettre à jour $F(y^k)$. Il faut noter que le premier pas de temps d'une méthode implicite doit être calculé précisément avec une méthode explicite paramétrée, possédant un pas de temps très petit.

Algorithme 7.1 algorithme séquentiel de quasi-Newton

```
YAncien = Tableau contenant le vecteur y à l'itération précédente
YCourant = Tableau contenant le vecteur y à l'itération courante
F = Fonction utilisée pour approximer l'EDO calculée selon (7.3)
DF = Jacobienne calculée en fonction de la fonction F
δ = Solution du système linéaire obtenu par Newton
Initialisation de YAncien et YCourant
for chaque pas de temps de l'intervalle étudié do
  repeat
    Prise en compte des conditions aux limites
    Calcul de la Jacobienne DF à la première itération en fonction de F, YCourant et
    YAncien
    Calcul de -F en fonction de YCourant et YAncien
    δ=SolveurLinéaire(DF,-F)
    YCourant=YCourant+δ
  until Convergence du processus de Newton
  YAncien=YCourant
end for
```

La parallélisation de cet algorithme repose sur l'utilisation de la méthode de multisplitting. Dans le chapitre précédent, nous avons vu qu'il était possible de "découper" la matrice en bandes afin que chaque processeur traite la partie qui le concerne. Concernant l'algorithme de quasi-Newton, l'approche est globalement la même. La matrice Jacobienne est découpée en blocs, chaque processeur s'occupe d'un bloc et échange des données avec ses voisins. La figure 7.1 illustre la décomposition de la matrice Jacobienne DF en $DFLoc$ et du vecteur δ en δLoc et de la fonction F en $FLoc$. La fonction

$FLoc$ utilise le vecteur Y en paramètre, ce dernier est décomposé en $YGauche$, $YLoc$ et $YDroite$. Ainsi le vecteur Y est noté après décomposition :

$$Y = \begin{pmatrix} YGauche \\ YLoc \\ YDroite \end{pmatrix}. \tag{7.7}$$

Cette décomposition dans son ensemble peut sembler surprenante car une partie de la matrice Jacobienne est complètement ignorée, alors qu'elle ne contient pas que des 0. Sur la figure, cette partie est représentée par les deux 0 situé à coté de la partie $DFLoc$.

À chaque itération, un processeur impliqué dans la résolution de l'algorithme multisplitting-Newton résoud :

$$DFLoc * \delta Loc = -FLoc \begin{pmatrix} YGauche \\ YLoc \\ YDroite \end{pmatrix}. \tag{7.8}$$

Les processeurs impliqués dans la mise en œuvre de cet algorithme échangent des parties du vecteur Y. La fonction $FLoc$ utilise une partie du vecteur Y qui est calculée localement et les parties $YGauche$ et $YDroite$ sont calculées par les processeurs voisins. Le calcul de cette fonction $FLoc$ retourne un vecteur de la taille de δLoc (donc aussi de la taille de $YLoc$ et de $DFLoc$).

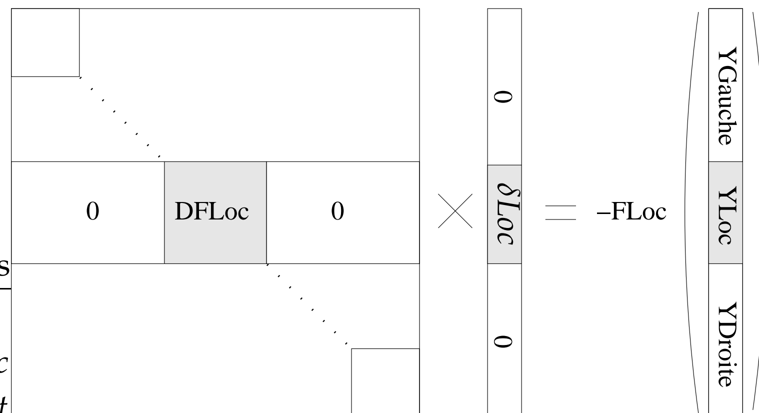

FIG. 7.1 – Décomposition de la Jacobienne du vecteur et de la fonction

L'algorithme 7.2 présente la synthèse des 3 variantes de l'algorithme de quasi-Newton que nous proposons. La décomposition du problème est effectuée selon le schéma de la figure 7.1. Il est possible que certaines parties de la matrice Jacobienne soient calculées par plus d'un processeur. Dans ce cas, le recouvrement permet de diminuer le nombre d'itérations comme présenté dans le chapitre 6.

La première partie de l'algorithme 7.2 a pour but d'initialiser les différentes variables. Il est à noter que le vecteur $YAncien$ contient les données du pas de temps

Algorithme 7.2 algorithme parallèle multisplitting-Newton

NbProcs = *Nombre de processeurs*
MonRang = *Rang du processeur*
YAncien = *Tableau contenant le vecteur y au pas de temps précédent*
YCourantLoc = *Tableau contenant la partie locale du vecteur y à l'itération courante*
YGauche = *Tableau contenant la partie gauche du vecteur y à l'itération courante*
YDroite = *Tableau contenant la partie droite du vecteur y à l'itération courante*
YCourant = *Tableau contenant le vecteur y à l'itération courante*
FLoc = *Partie locale de la fonction utilisée pour approximer l'EDO calculée selon (7.3)*
DFLoc = *Partie locale de la Jacobienne calculée en fonction de la fonction FLoc*
δLoc = *Partie locale de solution du système linéaire obtenu par Newton*
DependDeMoi[NbProcs] = *Tableau des processeurs dépendants*
Initialisation de toutes les variables
for chaque pas de temps de l'intervalle étudié **do**
  repeat
    Prise en compte des conditions aux limites si le processeur est concerné
    Assemblage du vecteur YCourant à partir de YGauche, YLoc et YDroite
    Calcul de la partie de Jacobienne DFLoc à la première itération en fonction de Floc, YCourant et YAncien
    Calcul de la sous-partie -FLoc en fonction de YCourant et YAncien
    δLoc=SolveurLinéaire(DFLoc,-FLoc)
    YCourantLoc=YCourantLoc+δLoc
    for i=0 jusqu'à NbProcs-1 **do**
      if i !=MonRang et DependDeMoi[i] **then**
        Envoi(Morceau(YCourantLoc,i),i)
      end if
    end for
    Détection de convergence
  until Convergence globale du processus de Newton
  YAncien = YCourant
end for

précédent, alors que les vecteurs *YCourant*, *YCourantLoc*, *YGauche* et *YDroite* correspondent à l'itération courante, donc au pas de temps courant. Ensuite l'algorithme calcule les pas de temps les uns après les autres. Les processeurs situés sur les limites du domaine doivent tenir compte des conditions aux limites. Ensuite chaque processeur assemble le vecteur *YCourant* à partir du vecteur *YLoc* qu'il possède et les vecteurs *YGauche* et *YDroite* qu'il a recu de ses voisins. À la première itération, le calcul local de la Jacobienne est effectué par chaque processeur. Chaque nœud calcule ensuite la partie de la fonction locale *FLoc* qui le concerne. Dès lors, il résout le système linéaire et, de la sorte, il met à jour la partie locale du vecteur *YCourantLoc* avec la solution du système linéaire calculée précédemment.

Puis, selon le mode de fonctionnement de l'algorithme (synchrone ou asynchrone), les réceptions seront implantées en respectant respectivement la description des algorithmes ISCS et ISCA (cf. section 3.2.2) ou IACA (cf. section 4.3). Les réceptions n'apparaissent pas dans l'algorithme afin que celui-ci devienne suffisamment générique et utilisable avec les trois versions d'algorithmes parallèles. Un processeur n'envoie pas les mêmes données à tous ses voisins. La fonction $Morceau(YCourantLoc, i)$ permet de calculer la partie de *YCourantLoc* dont le processeur i a besoin pour son calcul. Cette partie spécifique du vecteur *YCourantLoc* est envoyée au processeur i. Les réceptions ont pour effet de placer les données d'un processeur au bon endroit, dans le vecteur *YGauche* ou *YDroite*. En fonction du recouvrement, les envois et les réceptions ne s'effectuent pas en employant les mêmes valeurs. Cependant, les 3 variantes d'algorithmes itératifs parallèles utilisent le même schéma de communication ; seules les valeurs des données varient entre une version synchrone et asynchrone en fonction des différences d'itérations et des délais de communications pour la version asynchrone.

Finalement, la dernière étape d'une itération de quasi-Newton consiste à détecter la convergence. Une fois celle-ci atteinte, le vecteur *YCourant* est recopiée dans *YAncien*.

Pour des informations complémentaires sur la convergence des méthodes de multisplitting, nous invitons le lecteur intéressé à consulter [31, 12, 7, 65] et les articles auxquels ils font référence.

7.3 Expérimentations sur deux problèmes

Nous avons mené plusieurs expérimentations utilisant le travail décrit dans cette section. Dans un premier temps, nous avons cherché à résoudre un système d'EDP permettant de calculer l'évolution de deux composés chimiques sur une surface d'eau peu profonde. La modélisation est effectuée en 2 dimensions et elle utilise les équations de transport. Par la suite, nous avons appliqué le même type d'expérimentation, en trois dimensions, c'est pourquoi nous présentons uniquement cette version qui est légèrement plus complexe que la précédente. Le travail en 2D est présenté dans [15]. Dans un second temps, nous présentons la résolution d'une équation des ondes non

linéaire discrétisée en 2 dimensions.

7.3.1 Résolution d'un système de transport de composés polluants

Présentation mathématique du problème

L'objectif de cette étude est de modéliser le processus de transport de composés polluants, chimiques, ..., dans une eau peu profonde, et en tenant compte des réactions bio-chimiques des composants. Le système possède une valeur initiale et il faut calculer l'évolution des composants avec le temps. Ce travail est présenté dans [22] et plus en détail dans [21]. L'évolution du système est régi par une équation d'advection-diffusion de la forme :

$$\frac{\partial c^i}{\partial t} + A(c^i, u, v, w) = D(c^i, \varepsilon(x), \varepsilon(y), \varepsilon(z)) + R^i\left(c^1, c^2, \ldots, c^m, t\right) \tag{7.9}$$

où $i \in \{1, 2, \ldots, m\}$ représente les espèces chimiques et

$$A(c^i, u, v, w) = u \cdot \frac{\partial c^i}{\partial x} + v \cdot \frac{\partial c^i}{\partial y} + w \cdot \frac{\partial c^i}{\partial z} \tag{7.10}$$

$$D(c^i, \varepsilon(x), \varepsilon(y), \varepsilon(z)) = \frac{\partial}{\partial x}\left(\varepsilon(x) \cdot \frac{\partial c^i}{\partial x}\right) + \frac{\partial}{\partial y}\left(\varepsilon(y) \cdot \frac{\partial c^i}{\partial y}\right) + \frac{\partial}{\partial z}\left(\varepsilon(z) \cdot \frac{\partial c^i}{\partial z}\right) \tag{7.11}$$

définissent respectivement les processus d'advection et de diffusion. Les différentes quantités dans les équations (7.9), (7.10) et (7.11) sont définies de la manière suivante :

- c^i représente les concentrations (inconnues) des espèces chimiques,
- u, v, w représentent les vélocités des fluides locaux,
- $\varepsilon(x), \varepsilon(y), \varepsilon(z)$ représentent les coefficients de diffusion,
- R^i décrit les interactions bio-chimiques entre les composants.

La vélocité des fluides locaux et les coefficients de diffusion sont connus à l'avance. Les termes R^i représentent les réactions chimiques inter-espèces et les émissions depuis les éventuelles sources. La non-linéarité du système provient des couplages entre les composants que l'on retrouve dans les termes R^i.

Les expérimentations que nous avons entreprises suivent le problème précédemment évoqué (7.9)-(7.11) dans un espace à 3 dimensions comprenant deux espèces chimiques, pour lesquelles nous utilisons les différentes quantités :

- $u = v = -V = -10^{-3}$ et $w = 0$
- $\varepsilon(x) = \varepsilon(y) = K_h = 4.0 \times 10^{-6}$ et $\varepsilon(z) = K_v(z) = 10^{-8} \times \exp\left(\frac{z}{5}\right)$
- $R^1\left(c^1, c^2, t\right)$ et $R^2\left(c^1, c^2, t\right)$ suivent respectivement :

$$R^1\left(c^1, c^2, t\right) = -q_1 \cdot c^1 \cdot c^3 - q_2 \cdot c^1 \cdot c^2 + 2 \cdot q_3(t) \cdot c^3 + q_4(t) \cdot c^2$$

$$R^2\left(c^1, c^2, t\right) = q_1 \cdot c^1 \cdot c^3 - q_2 \cdot c^1 \cdot c^2 - q_4(t) \cdot c^2$$

avec $c^3 = 3.7 \times 10^{16}$, $q_1 = 1.63 \times 10^{-16}$, $q_2 = 4.66^{-16}$ et $q_3(t)$, $q_4(t)$ sont définis par $(j = 3, 4)$:

$$q_j(t) = \exp\left[\frac{-a_j}{\sin(\omega t)}\right] \quad \text{si } \sin(\omega \cdot t) > 0$$
$$q_j(t) = 0 \quad \text{si } \sin(\omega \cdot t) \leq 0$$

avec les paramètres suivants : $\omega = \pi/43200$, $a_3 = 22.62$ et $a_4 = 7.601$.

De la sorte, le problème 3D s'exprime grâce à deux EDP :

$$\frac{\partial c^1}{\partial t} = K_h \cdot \frac{\partial^2 c^1}{\partial x^2} + K_h \cdot \frac{\partial^2 c^1}{\partial y^2} + V \cdot \frac{\partial c^1}{\partial x} + V \cdot \frac{\partial c^1}{\partial y} + \frac{\partial}{\partial z}\left(K_v(z) \cdot \frac{\partial c^1}{\partial z}\right) + R^1\left(c^1, c^2, t\right)$$

$$\frac{\partial c^2}{\partial t} = K_h \cdot \frac{\partial^2 c^2}{\partial x^2} + K_h \cdot \frac{\partial^2 c^2}{\partial y^2} + V \cdot \frac{\partial c^2}{\partial x} + V \cdot \frac{\partial c^2}{\partial y} + \frac{\partial}{\partial z}\left(K_v(z) \cdot \frac{\partial c^2}{\partial z}\right) + R^2\left(c^1, c^2, t\right)$$

Les conditions initiales sont les suivantes :

$$c^1(x, y, z, t = 0) = 10^6 \cdot \alpha(x) \cdot \beta(y) \cdot \gamma(z)$$

$$c^2(x, y, z, t = 0) = 10^{12} \cdot \alpha(x) \cdot \beta(y) \cdot \gamma(z)$$

où α, β et γ sont définis par :

$$\alpha(x) = 1 - (0.1 \cdot x - 1)^2 + (0.1 \cdot x - 1)^4/2$$

$$\beta(y) = 1 - (0.1 \cdot y - 1)^2 + (0.1 \cdot y - 1)^4/2$$

$$\gamma(z) = 1 - (0.1 \cdot z - 1)^2 + (0.1 \cdot z - 1)^4/2$$

Expérimentations

Les expérimentations suivantes ont été réalisées avec une version synchrone et asynchrone de l'algorithme de multisplitting-Newton. Pour l'implantation nous avons utilisé MPI/Madeleine [10].

Toutes les expérimentations ont été réalisées avec 16 machines localisées sur deux sites distants (Belfort et Monbéliard). Ces deux sites sont reliés par un réseau à 20mb/s. Sur chaque site, les machines sont reliées par un réseau local ethernet standard à 100mb/s. Sur le site de Belfort, nous avons utilisé 13 machines possédant les caractéristiques suivantes :

- 2 Pentium IV 1.6Ghz,
- 1 Pentium IV 1.7Ghz,
- 1 Pentium IV 1.8Ghz,
- 6 Pentium IV 2.4Ghz,
- 1 Pentium IV 2.6Ghz,
- 2 Pentium IV 2.66Ghz.

À Montbéliard, nous avons utilisé 3 machines Athlon 1.66Ghz. Les tailles de problèmes varient de $60 \times 60 \times 60$ jusqu'à $100 \times 100 \times 100$ (avec une discrétisation identique dans chaque dimension) pour chaque expérimentation. Avec 16 processeurs la grille du problème est décomposée en 4 en X, en 2 en Y et en 2 en Z. Dans la suite, les résultats représentent la moyenne d'une série de 10 exécutions. Nous avons mené les expérimentations afin de mettre en évidence différentes propriétés de nos algorithmes.

Dans un premier temps, nous souhaitons comparer le comportement des versions synchrones (ISCS) et asynchrones (IACA). Toujours en raison des restrictions dûes aux problèmes de sécurité, nous n'avons pas pu utiliser plus de deux sites. Le passage des pare-feu avec les bibliothèques de communications usuelles pose des difficultés avec MPI/Mad. Pour simuler des conditions d'expérimentations avec des communications longues distances, nous avons, comme dans le chapitre précédent, choisi de perturber artificiellement les communications. De la sorte, nous pouvons comparer, dans un second temps, la robustesse des différentes versions de notre algorithme en fonction des perturbations que nous avons engendrées de manière reproductible.

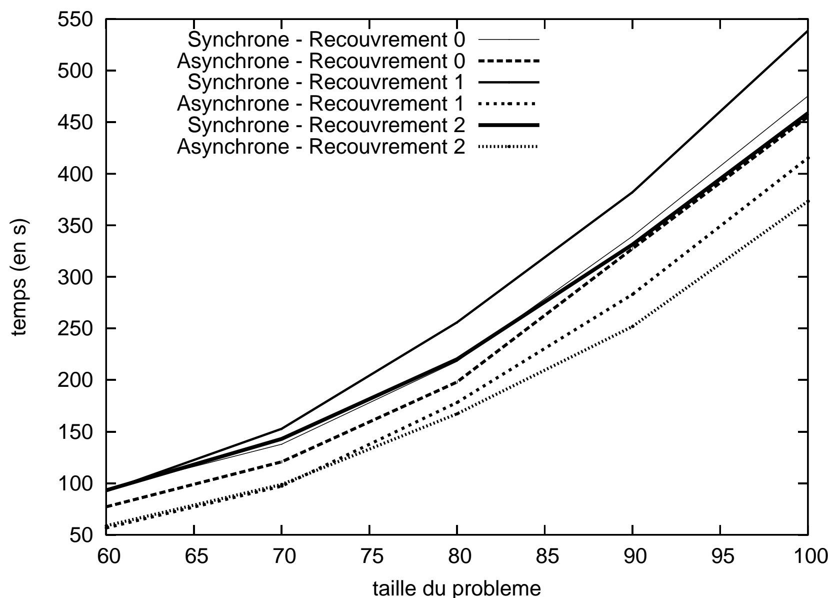

FIG. 7.2 – Influence du recouvrement

Dans la figure 7.2, nous illustrons l'influence du recouvrement sur les temps d'exécution. Deux caractéristiques sont à signaler. La première est que la version asynchrone est toujours plus rapide que la version synchrone. La seconde concerne l'impact du paramètre de recouvrement. La croissance de ce paramètre entraîne éventuellement une réduction du nombre d'itérations nécessaire pour atteindre la convergence, mais le temps d'exécution d'une itération est plus grand. Pour illustrer ce fait, prenons une grille de taille $100 \times 100 \times 100$ avec 16 processeurs (comme nous le faisons), avec la décomposition que nous avons choisie. Dans ce cas, un processeur

est responsable d'une sous-grille de taille $25 \times 50 \times 50$ (avec une décomposition de 4 processeurs en X et 2 processeurs en Y et Z). Comme dans le problème considéré, il y a 2 espèces chimiques, le système linéaire qu'un processeur doit résoudre à chaque itération de Newton est de taille $25 \times 50 \times 50 \times 2 = 125000$ sans recouvrement. Si un processeur possède des voisins dans chaque direction, c'est-à-dire 6 voisins, et si le paramètre de recouvrement est fixé à 2, alors le système linéaire à résoudre est de taille de $29 \times 52 \times 52 \times 2 = 156832$, ce qui correspond environ à une augmentation de 25%. Dans le contexte synchrone, avec une taille de recouvrement égale à 1, le nombre d'itération n'est pas suffisamment réduit, c'est pourquoi le temps d'exécution ne diminue pas. Ceci s'explique par le fait que tous les processeurs doivent s'adapter à la vitesse du plus lent d'entre eux. Dans le contexte asynchrone, cet inconvénient est en partie évité, car les processeurs plus rapides ne sont pas trop ralentis par ceux qui sont plus lents. En fait, les processeurs rapides mettent à jour les processeurs plus lents avec des données plus précises. Avec une taille de recouvrement de 2, les versions synchrones et asynchrones sont plus rapides que sans recouvrement, néanmoins la version asynchrone est clairement plus efficace.

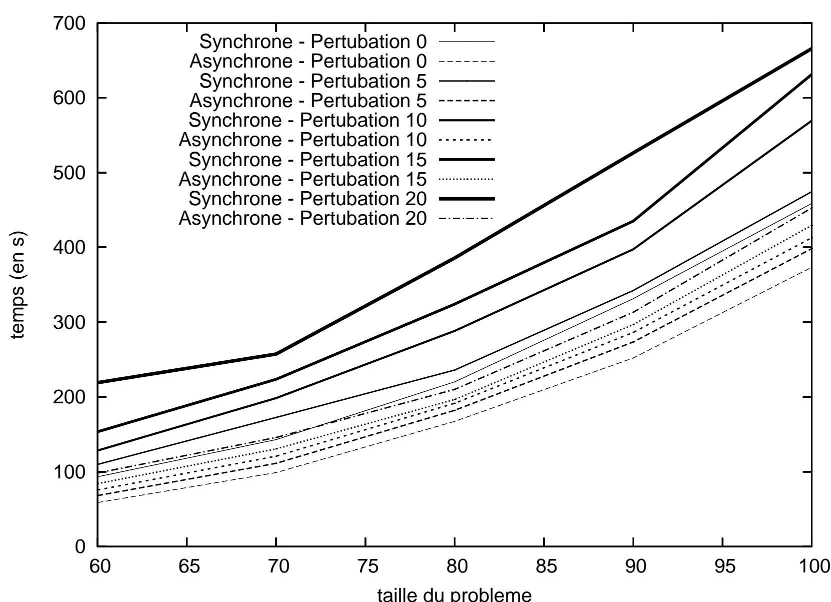

FIG. 7.3 – Influence des communications perturbantes

La seconde série d'expérimentations a pour but d'étudier le comportement des deux versions d'algorithmes en présence de perturbations des communications. Comme dans le chapitre 6, nous utilisons l'outil iperf [107] (mesurant la bande passante entre deux machines) entre deux machines situées sur les deux sites de calculs qui ne servent pas directement au calcul. Ainsi, la bande passante entre les deux sites est très perturbée. Le nombre de processus d'iperf n'influence pas linéairement les temps d'exécution. Sur la figure 7.3, le nombre *Perturbation* indique le nombre de

processus utilisés par iperf pour “perturber” la bande passante. De l’expérimentation précédente, nous déduisons que le paramètre de recouvrement idéal est 2. C’est pourquoi nous choisissons ce paramètre. Sur la figure, les exécutions synchrones sont illustrées par des traits pleins, alors que les exécutions asynchrones sont représentées par des traits pointillés. Comme précédemment, la version asynchrone est manifestement plus rapide que la version asynchrone, ce qui montre la robustesse de l’asynchronisme face à une mauvaise bande passante. De plus, même en présence de fortes perturbations, la version asynchrone se révèle quand même plus rapide que la version synchrone sans perturbation. Ce point est réellement très intéressant.

Bilan

Cette série de tests montre clairement la supériorité de la version asynchrone de notre algorithme de multisplitting-Newton, qui est spécialement adapté à un contexte de grille dans lequel les bandes passantes et les latences peuvent être sujettes à des perturbations.

7.3.2 Résolution d’un système d’équations d’ondes non linéaires

Présentation mathématique du problème

Le problème d’équations des ondes considéré rentre dans la classe des équations de Klein-Gordon [49] : il s’agit d’un système d’équations du second ordre non linéaires. Le problème étudié est en 2 dimensions et l’objectif est de définir $u(x, y, t)$ satisfaisant :

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f(u(t)),\ t \in [0, T],\ x, y \in \Omega \times \Omega \tag{7.12}$$

avec $u(x, y, 0)$, $u \mid_{\partial\Omega}$ et $\frac{\partial u}{\partial t} \mid_{\partial\Omega}$ connu.

En fonction de la valeur de $f(u(t))$, l’équation (7.12) se comporte différemment. En prenant le cas particulier pour lequel $f(u(t))$ est égal à 0, on retombe sur une simple équation des ondes linéaires. Pour davantage d’informations sur l’équation de Klein Gordon, nous invitons le lecteur intéressé à consulter [84, 66].

Par la suite, nous fixons $f(u(t))$ et le système considéré est le problème appelé Sine Gordon qui fait partie de la classe de Klein Gordon. En 2D il s’écrit de la forme suivante :

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - sin(u(t)) \tag{7.13}$$

Conditions d’expérimentations

Dans la suite, les expérimentations que nous avons menées, ont été réalisées en fixant le pas de temps à 0.1 seconde. Tous les résultats suivants sont la moyenne de 10

expérimentations. Selon les comportements que nous avons voulu mettre en évidence, nous avons fait varier certains paramètres (comme pour le problème précédent) :

- la taille de discrétisation spatiale pour mesurer l'impact du ratio *temps de calcul / temps de communication*,
- le nombre de processeurs afin d'étudier le passage à l'échelle en fonction de la configuration de la grappe de calcul,
- la taille de recouvrement pour mesurer son influence,
- le nombre de communications perturbantes pour tester la robustesse des algorithmes.

Pour ces expérimentations, nous faisons figurer le temps d'exécution des algorithmes (en secondes) et le nombre d'itérations de la version synchrone sur l'axe des ordonnées. Pour cette raison, nous ne faisons pas figurer d'unité. Le nombre d'itérations de la version asynchrone n'y figure pas, car il varie d'une exécution à une autre, et parce qu'il diffère suivant les processeurs, si ces derniers sont hétérogènes (ou si le problème est irrégulier ce qui n'est pas le cas ici).

Expérimentations dans un contexte local

Cette première série d'expérimentations a pour objectif de comparer les temps d'exécution des versions synchrones et asynchrones de l'algorithme sur une grappe locale et homogène afin d'étudier le passage à l'échelle. Pour cela, nous choisissons une grille de discrétisation de taille 700×700 et nous n'utilisons pas de paramètre de recouvrement. Disposant d'uniquement 16 processeurs homogènes (des Pentium IV 2.4Ghz), nous mesurons les temps d'exécution des algorithmes avec un nombre de processeurs variant de 2 à 16, les machines étant reliées par un réseau 100Mb/s.

La figure 7.4 montre que le nombre d'itérations croit lorsqu'on augmente le nombre de processeurs bien que le temps d'exécution diminue. Avec 9 processeurs, le nombre d'itérations diminue, mais s'explique en raison de la configuration symétrique. En effet, avec 9 processeurs, la grille est divisée en 3 par 3 (3 horizontalement et 3 verticalement), alors que la plupart des autres configurations sont divisées en $2 \times n$ avec n entier. De plus, nous remarquons que les temps d'exécution diminuent fortement avec l'augmentation du nombre de processeurs. La version synchrone est plus rapide dans ce contexte homogène que la version asynchrone, mais ceci n'est pas surprenant, car ce contexte est favorable à la version synchrone.

Une seconde série d'expérimentations nous permet de mesurer l'influence de la taille du recouvrement sur la même grappe locale homogène (16 Pentium IV 2.4Ghz, réseau 100Mb/s). La figure 7.5 montre qu'une petite taille de recouvrement permet de réduire sensiblement les temps d'exécution des versions synchrone et asynchrone de l'algorithme. De plus, on remarque que la taille idéale pour le recouvrement est égale à 3. L'avantage est encore pour la version synchrone. Le faible nombre d'itérations nécessaire à la version synchrone pour atteindre la convergence explique probable-

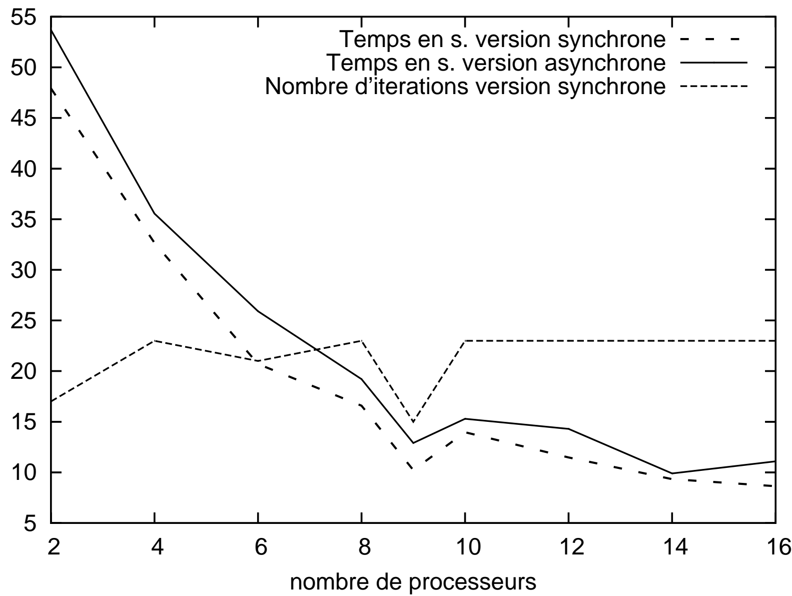

FIG. 7.4 – Temps d'exécution des versions synchrones et asynchrones sur une grille de taille 700*700 sans recouvrement.

ment ce résultat.

Le but de la troisième série d'expérimentations est de comparer les comportements des versions synchrone et asynchrone avec une grappe locale de 40 machines hétérogènes (15 AMD Duron 800 Mhz, 15 Pentium IV 1.7Ghz et 10 Pentium IV 2.4Ghz) reliées par un réseau standard 100Mb/s. Nous avons réalisé les tests avec 3 tailles de discrétisation spatiale (700*700, 1000*1000 et 1300*1300). Pour chaque configuration, nous mesurons l'influence de la taille du recouvrement sur les temps d'exécution.

Les figures 7.6, 7.7 et 7.8 illustrent le fait que la taille optimale du recouvrement varie selon la taille de la grille de discrétisation. Les tailles optimales sont respectivement de 2 pour la grille de taille $700 * 700$, de 3 pour la grille de taille $1000 * 1000$ et de 5 pour la grille de taille $1300 * 1300$. De plus, la version asynchrone est toujours légèrement plus rapide que la version synchrone. Bien que le nombre d'itérations soit plus important pour le cas asynchrone, les machines rapides tendent à accélérer les machines plus lentes en leur fournissant des résultats plus proches de la solution. On constate, de plus, que le temps d'exécution avec 40 machines hétérogènes est presque identique à celui obtenu avec seulement 16 machines homogènes plus rapides. Ce point indique que l'homogénéité des machines influe de manière significative sur les temps d'exécution, notamment pour des petites grilles de discrétisation.

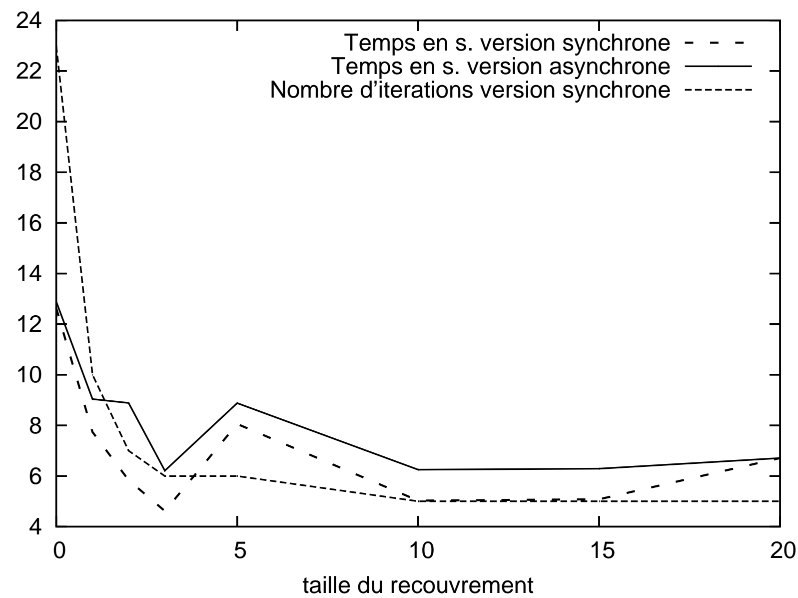

FIG. 7.5 – Influence du recouvrement sur une grille de taille 700*700 avec 16 processeurs homogènes.

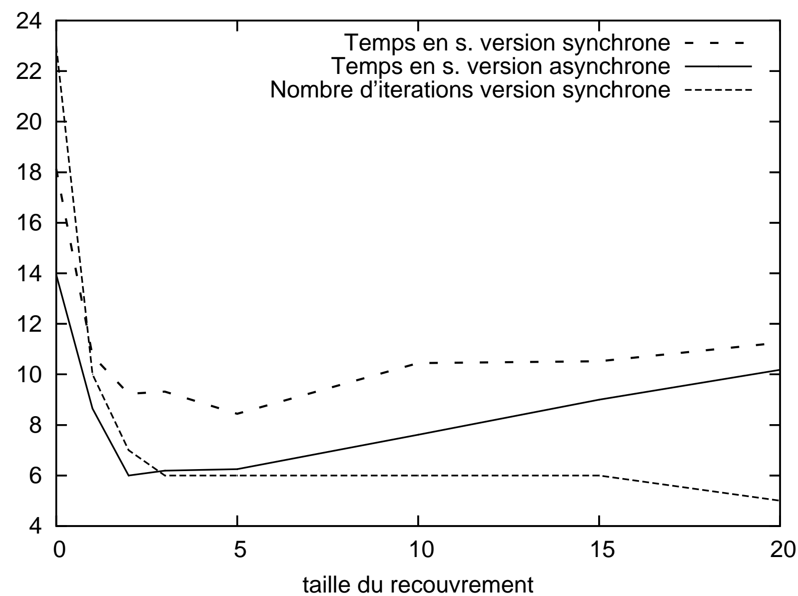

FIG. 7.6 – Influence du recouvrement sur une grille de taille 700*700 avec une grappe locale hétérogène de 40 processeurs.

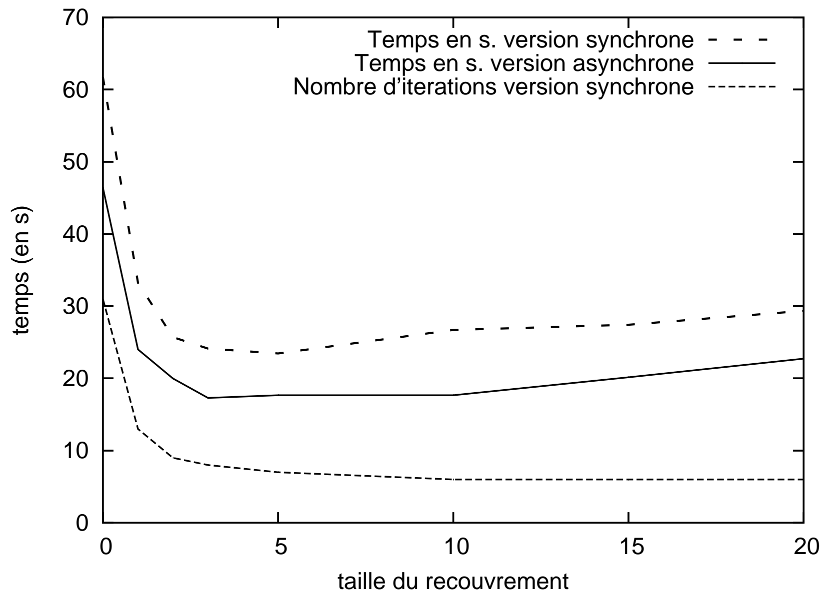

FIG. 7.7 – Influence du recouvrement sur une grille de taille 1000*1000 avec une grappe locale hétérogène de 40 processeurs.

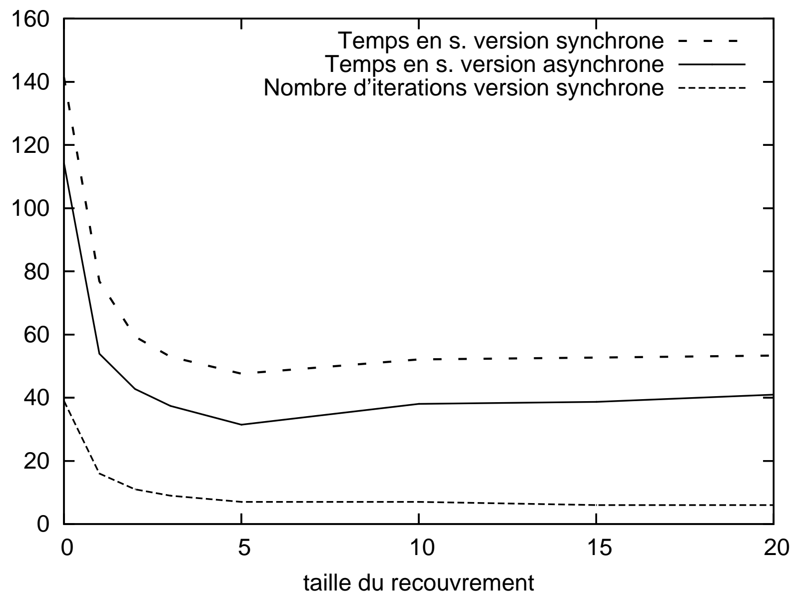

FIG. 7.8 – Influence du recouvrement sur une grille de taille 1300*1300 avec une grappe locale hétérogène de 40 processeurs.

Expérimentations distantes

Après l'étude des deux versions d'algorithmes dans un contexte local, nous avons mené des expérimentations dans un contexte de grappes distantes. Pour cela, nous avons utilisé des machines des sites de Belfort et de Montbéliard reliées par un réseau de 20Mb/s. Une première série d'expérimentations nous permet de mesurer l'influence du recouvrement sur une grille de discrétisation spatiale de taille 1000*1000 avec une grappe de 16 processeurs hétérogènes. La configuration est la suivante. À Montbéliard, nous avons utilisé 6 AMD Athlon XP 2Ghz, à Belfort nous avons choisi 1 Pentium IV 2.66Ghz, 6 Pentium IV 2.4Ghz, 2 Pentium IV 1.6 Ghz et 1 AMD Athlon XP 2.8 Ghz. La figure 7.9 illustre que les deux versions ont un comportement relativement similaire, avec un léger avantage pour la version asynchrone.

Notons que cette configuration (avec 16 machines distantes) offre des temps assez proches de ceux obtenus avec la grappe locale précédente (composée de 40 machines moins puissantes). C'est pourquoi une grappe distante peut s'avérer être une solution performante pour les scientifiques qui n'ont pas accès à des grappes locales de grande envergure.

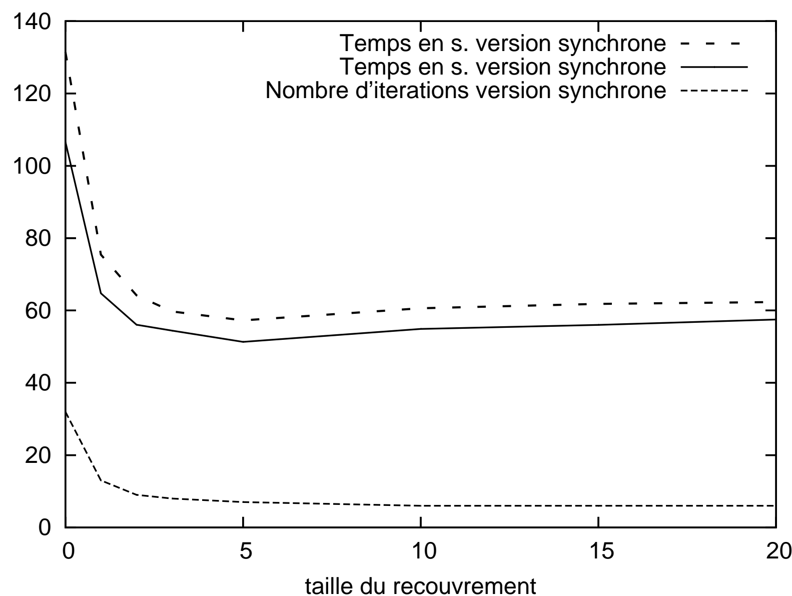

FIG. 7.9 – Influence du recouvrement sur une grille de taille 1000*1000 avec une grappe distante hétérogène de 16 processeurs.

Finalement, la figure 7.10 illustre l'influence des communications perturbantes pour la grappe hétérogène et distante. Pour ce test, une grille de discrétisation spatiale de taille 1000*1000 a été choisie avec une taille de recouvrement optimale, c'est-à-dire 5. Ainsi, sans communication perturbatrice, ce comportement est identique à celui qui a été observé sur la figure 7.9. Cependant, la figure 7.10 met en avant la robustesse de

la version synchrone dans un contexte de communications soumis à des perturbations.

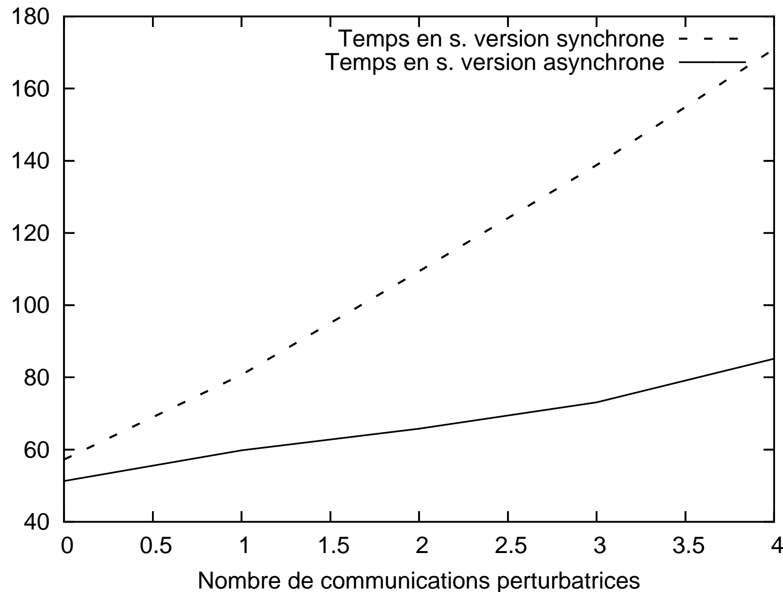

FIG. 7.10 – Influence des communications perturbantes sur une grille de taille 1000*1000 avec une grappe distante hétérogène de 16 processeurs.

Bilan

Ce problème d'équation des ondes non linéaires montre que la version synchrone de l'algorithme multisplitting-Newton est plus efficace, avec une grappe locale et homogène, que la version asynchrone. Par contre, la version asynchrone est plus robuste lorsque le nombre de processeurs devient important et surtout face à des machines hétérogènes et distantes.

7.4 Conclusion

Ce chapitre décrit la parallélisation d'un système d'équations non linéaires. La technique choisie repose sur l'utilisation du multisplitting qui permet de "découper" le problème en plusieurs sous-problèmes. La résolution de la partie linéaire est effectuée avec la méthode de Newton. Ainsi, l'algorithme que nous présentons dans ce chapitre, est appelé multisplitting-Newton ; il est itératif. Chaque processeur impliqué dans la résolution du système résout une partie de celui-ci et échange des informations avec ses voisins, une seule fois par itération. C'est pourquoi ce type de parallélisation est à gros grains. L'algorithme de multisplitting-Newton fonctionne en mode synchrone et

asynchrone. Ce dernier lui confère une robustesse face au contexte des grilles de calcul pour lesquelles la distance entre les machines et les communications, sujettes à des perturbations, pénalisent la version synchrone.

Nous avons appliqué avec succès cet algorithme pour résoudre deux applications. La première a pour objectif de résoudre une équation d'advection-diffusion en trois dimensions. La seconde permet de résoudre une équation d'ondes non linéaires en deux dimensions.

Le dernier chapitre établit le lien entre les deux parties de ce mémoire en présentant le couplage d'un algorithme IACA avec un algorithme d'équilibrage de charge. Ces deux techniques ont pour objectif commun de réduire les temps de synchronisations d'une application numérique.

Chapitre 8

Équilibrage de charge pour les algorithmes IACA

8.1 Introduction

Nous avons évoqué dans la première partie de ce mémoire l'intérêt de l'équilibrage de charge pour les algorithmes distribués. On peut ainsi répartir la charge de manière optimale (lorsque l'algorithme a convergé) afin de réduire au maximum les temps d'attente durant l'exécution d'un algorithme. Si l'équilibrage de charge se déroule de manière optimale, à chaque synchronisation d'un algorithme, tous les processeurs arrivent au même moment à l'instruction qui synchronise l'ensemble des machines. Cependant, dans un contexte distribué sans centralisation, la répartition optimale de la charge n'est pas simple et elle requiert éventuellement de nombreuses itérations (sans prendre en compte les environnements dynamiques), et ceci même sans modéliser les pertes de liens dans un réseau.

D'autre part, comme nous l'avons étudié dans les deux précédents chapitres, les algorithmes IACA offrent la particularité de supprimer les synchronisations d'un algorithme et de recouvrir les communications par du calcul.

Au premier abord, il peut paraître surprenant de mélanger asynchronisme et équilibrage de charge, puisque ces deux méthodes ont des modes opératoires différents et sont censées solutionner des problèmes liés au calcul distribué de manière autonome. Cependant, ces deux méthodes ont pour objectif commun de réduire (ou supprimer) les synchronisations d'une application numérique, en utilisant des mécanismes différents. Voici les éléments qui nous semblent intéressants à prendre en compte pour montrer l'intérêt d'un tel couplage.

- D'une part, les algorithmes d'équilibrage de charge décentralisés ne peuvent pas répartir la charge en une seule itération, ils convergent plus ou moins rapidement vers une distribution uniforme de la charge, s'ils ne sont pas perturbés. En cas de perturbations, dûes par exemple à l'algorithme de calcul qui fait varier la

charge ou à l'environnement d'exécution, la convergence est retardée. Pendant ce temps, une partie des ressources de calcul est perdue. De plus, les algorithmes d'équilibrage de charge ne permettent pas de supprimer les synchronisations ou les communications bloquantes d'un algorithme, contrairement aux algorithmes IACA.

– D'autre part, même si l'exécution d'un algorithme IACA semble optimale en raison de la suppression des communications bloquantes, le calcul peut être déséquilibré. L'asynchronisme ne peut pas corriger ce déséquilibre, il permet de laisser les processeurs évoluer à leur propre vitesse. Enfin, il faut garder en tête que le contexte des grappes distantes est propice à l'hétérogénéité les machines et des réseaux.

Compte tenu des particularités des algorithmes IACA et de la diversité des algorithmes d'équilibrage de charge, il est clair qu'il n'est pas possible d'appliquer tous les algorithmes d'équilibrage de charge avec les algorithmes asynchrones. Pour intégrer un algorithme d'équilibrage de charge à un algorithme IACA, il faut que l'algorithme d'équilibrage de charge soit décentralisé et asynchrone. De plus, il paraît important qu'il prenne en compte les dépendances du calcul et qu'il soit le moins bloquant possible, afin de ne pas trop ralentir les itérations.

Dans la première partie de ce document, nous avons présenté différentes variantes des algorithmes de diffusion ainsi que leur adaptation aux réseaux dynamiques. Ces algorithmes sont synchrones et, par le fait, ne sont pas adaptés au contexte des algorithmes IACA. Dans [38], Bertsekas et Tsitsiklis présentent un modèle d'algorithme d'équilibrage de charge décentralisé et asynchrone. L'algorithme que nous avons implanté peut être modélisé par ce modèle.

Ce chapitre commence par présenter l'algorithme d'équilibrage que nous proposons de coupler avec un algorithme IACA. La définition de la charge nous amène ensuite à proposer un estimateur très intéressant qui tend à faire converger tous les processeurs au même instant.

8.2 Description de l'algorithme

L'algorithme que nous présentons décrit précisément comment coupler une méthode d'équilibrage de charge à un algorithme IACA. Suivant les méthodes numériques employées, il est possible d'adapter ce couplage afin d'améliorer l'interaction entre l'algorithme d'équilibrage et l'algorithme asynchrone. Pour synthétiser notre démarche nous présentons un algorithme général.

L'algorithme 8.1 reprend l'algorithme 4.1 présenté dans la partie précédente. Au début de chaque itération, il faut mettre à jour les données reçues ou envoyées lors de

l'itération précédente (si c'est le cas). La variable *MiseAJourEquiCharge* permet de tester s'il faut mettre des données à jour en raison d'un équilibrage. Ensuite, l'algorithme vérifie s'il a reçu de la charge. Dans ce cas, il est nécessaire de modifier la taille des tableaux en conséquence. Cette opération est protégée par un mutex afin d'empêcher la réception de données pour le calcul pendant ce temps. Quand un processeur reçoit de la charge de la part d'un de ses voisins, il la réceptionne dans un tableau temporaire. Comme il est possible que plusieurs processeurs envoient une partie de leur charge aux mêmes nœuds, il faut mettre à jour les données du calcul en prenant en compte les différents envois. Si un processeur a émis de la charge à l'itération précédente, il doit également mettre à jour les données. Comme il est possible qu'un processeur émette de la charge d'une part, et en reçoive d'autre part, les deux mises à jour sont effectuées l'une après l'autre.

S'il n'y a pas de mise à jour des données dûe à l'équilibrage, l'algorithme teste s'il faut effectuer un équilibrage de charge. Pour éviter de perdre trop de temps à s'équilibrer trop souvent, nous contraignons l'algorithme à ne pas s'équilibrer à toutes les itérations. La variable *ProchainEquiChargePossible* est utilisée à cet effet. Lorsque cette dernière a la valeur 0, l'algorithme active le ou les threads chargés de ces envois.

La fonction *EnvoisDonneesCharge* est décrite dans l'algorithme 8.2. Elle parcourt les voisins concernés, ou un sous-ensemble de ceux-ci, s'il y a plusieurs threads. Si un voisin n'est pas déjà en cours d'équilibrage, la fonction calcule le ratio de charge entre la charge du processeur et celle du nœud voisin. Ensuite, elle détermine le nombre de données à envoyer si le ratio est supérieur à une certaine valeur. Pour éviter qu'un processeur ne possède presque plus de charge, l'algorithme effectue un test. En cas de réussite de ce dernier, l'envoi de charge non bloquant est entrepris. La variable *MiseAJourEquiCharge* est mise à vrai afin qu'à l'itération suivante, la modification des tailles des variables utilisées pour le calcul soient effectuées. De plus, comme évoqué précédemment, la variable *ProchainEquiChargePossible* est affectée par la constante *NbIterSansEqui*. Cette constante permet de contrôler le nombre d'itérations que va effectuer l'algorithme sans équilibrer la charge. Suivant l'algorithme numérique employé, la taille du problème et le nombre de machines utilisées, donc le temps moyen d'exécution d'une itération, il faut éventuellement empêcher les équilibrages de charge trop fréquents. Finalement, lorsque toutes les conditions sont réunies, une partie de la charge est transférée au voisin i.

La fonction 8.3 est exécutée en boucle par un thread et elle prend en charge la réception d'une partie de la charge d'un voisin. Cette fonction est activée par le thread qui reçoit la charge. La réception des données de l'équilibrage se fait dans un tableau temporaire et les variables pour la mise à jour sont positionnées comme il se doit, au début d'une nouvelle itération de calcul.

La fonction 8.4 est utilisée pour recevoir les données durant le calcul. Par rapport à l'algorithme sans équilibrage, il faut veiller à utiliser les données du calcul, si elles correspondent à des données encore valides sur le processeur. En effet, si les

Algorithme 8.1 Algorithme IACA avec équilibrage de charge

```
Initialisation de la bibliothèque de communication
AnciennesDonnees = Tableau des valeurs de l'itération précédente
NouvellesDonneesLoc = Tableau des nouvelles valeurs locales
MiseAJourEquiCharge = Booléen pour savoir s'il faut mettre à jour les données
ProchainEquiChargePossible = Entier pour compter le nombre d'itération avant le pro-
chain équilibre
ChargeRecue = Booléen pour tester si on a recu de la charge
ChargeEmise = Booléen pour tester si on a émis de la charge
Création des threads pour les communications entrantes (appel à EnvoisDonnees)
Création des threads pour les communications sortantes (appel à ReceptionsDon-
nees)
Création des threads pour les communications entrantes (appel à EnvoisDonnees-
Charge)
Création des threads pour les communications sortantes (appel à ReceptionsDon-
nees)
Initialisation des données
repeat
  if MiseAJourEquiCharge = vrai then
    if ChargeRecue = vrai then
      Adapter la taille des tableaux AnciennesDonnees et NouvellesDonneesLoc
      Insérer les valeurs des tableaux temporaires dans AnciennesDonnees
      ChargeRecue = faux
    end if
    if ChargeEmise = vrai then
      Adapter la taille des tableaux AnciennesDonnees et NouvellesDonneesLoc
      ChargeEmise = faux
    end if
    MiseAJourEquiCharge = faux
  else
    if ProchainEquiChargePossible = 0 then
      Activation des threads chargés des envois de charge
    else
      ProchainEquiChargePossible = ProchainEquiChargePossible-1
    end if
  end if
  Calcul de NouvellesDonnesLoc en fonction de AnciennesDonnees
  Activation des threads chargés des communications sortantes
  Recopie de NouvellesDonnesLoc dans AnciennesDonnees
  Détection de convergence
until Convergence globale atteinte
```

Algorithme 8.2 fonction EnvoisDonneesCharge

Ratio = *Ratio d'évaluation de charge entre le nœud courant et le nœud i*
NbDonneesLocales = *Nombre de données locales*
NbAEnvoyer = *Nombre de données à envoyer pour réaliser l'équilibrage*
for tous les voisins i **do**
  if il n'y a pas d'équilibrage en cours avec i **then**
    Ratio=évaluation de charge locale / évaluation de charge du nœud i
    if Ratio>SeuilRatio **then**
      Calcul le nombre de données à envoyer avec NbAEnvoyer en fonction du Ratio
      if NbDonneesLocales-NbAEnvoyer > SeuilMinDonnées **then**
        Envoi non bloquant des données au processeur i
        MiseAJourEquiCharge = vrai
        ChargeEmise = vrai
        ProchainEquiChargePossible=NbIterSansEqui
      end if
    end if
  end if
end for

Algorithme 8.3 fonction ReceptionsDonneesEquiCharge()

Réception du nombre de données à recevoir
Réception des données dans un tableau temporaire
MiseAJourEquiCharge = vrai
ChargeRecue = vrai

données ont été mises à jour dernièrement, et qu'un message plus ancien survient, il faut éventuellement ne pas tenir compte de ce dernier. Le premier test dans la fonction sert à détecter si le tableau n'est pas en cours d'actualisation pour recevoir les données du calcul. Nous n'avons pas fait apparaître la fonction *EnvoiDonnees* car elle est très similaire à celle présentée au chapitre 4 (algorithme 4.2). Les ajouts sont les suivants : avant d'envoyer les données, cette fonction doit envoyer la position globale des données et après l'envoi de ces données, la fonction envoie également l'évaluation de la charge.

Algorithme 8.4 fonction ReceptionsDonnees()

```
if le tableau AnciennesDonnees n'est pas en cours d'actualisation then
  Réception de la position globale de début et de la fin de données
  if la position globale des données correspond aux données locales du nœud then
    Mise à jour du tableau AnciennesDonnees
  else
    Les Données reçues ne sont pas utilisées
  end if
  Réception de l'évaluation de charge du nœud
end if
```

8.3 Choix de l'estimateur de la charge

Nous avons évoqué la difficulté de choisir la charge à transférer pour de nombreuses applications. L'estimation de la charge est parfois délicat. Un estimateur de charge classique, basé sur le rapport entre le nombre de composantes et la puissance de calcul de la machine, fonctionne pour un algorithme couplant équilibrage et asynchronisme. Cependant un autre estimateur de charge que nous avons défini semble mieux adapté.

Dans un premier temps, analysons le fonctionnement d'un estimateur de charge classique. Disposant de n composantes à calculer, et considérant que le temps de calcul d'une composante est constant et égal à t, un processeur dont la vitesse de calcul est définie par v, mettra $\frac{nt}{v}$ unités de temps pour effectuer une itération. Pour équilibrer la charge entre deux processeurs, la solution la plus simple consiste à effectuer un rapport entre les deux charges afin de déterminer le nombre de composantes à déplacer d'un processeur à un autre. Notons que pour rester cohérent avec les deux premiers chapitres qui concernent l'équilibrage, la charge peut être définie simplement comme le nombre de composantes n par processeur. Dans ce cas, l'équilibre de charge est atteint lorsque le rapport entre le nombre de composantes et la puissance d'une machine $\frac{t}{v}$ est identique sur l'ensemble des machines. Cet estimateur de charge classique a pour

effet de réduire les temps morts entre les itérations ; c'est pourquoi on le rencontre fréquemment avec les algorithmes synchrones.

Les algorithmes IACA ont un contexte d'exécution différent des algorithmes synchrones. Le fait de réduire les temps de synchronisation entre les itérations n'est pas important pour les algorithmes IACA. Le temps d'exécution d'un algorithme IACA est le temps nécessaire pour que tous les processeurs passent sous le seuil de convergence. Le dernier processeur qui passe sous le seuil n'est pas forcément le plus lent. En effet, si un processeur recoit rarement des mises à jour de la part de ses voisins ou si l'évolution de l'erreur n'est pas homogène entre les processeurs, il est tout à fait possible qu'un processeur rapide détecte une convergence locale après tous les autres nœuds. Ceci justifie le besoin de définir un nouvel estimateur de charge pour les algorithmes IACA.

Avec les explications précédentes, on comprend que l'exécution d'un algorithme IACA est optimale, lorsque tous les processeurs détectent leur convergence locale simultanément. Le nouvel estimateur de charge que nous proposons a pour objectif d'atteindre cet idéal. Pour ce faire, cet estimateur de charge est basé sur le résidu local de l'algorithme, c'est à dire sur son erreur locale calculée à partir des deux dernières itérations. Le nombre de composantes à transférer est fonction du rapport entre les résidus des deux processeurs impliqués dans le transfert de charge. Dans ce cas, lorsqu'un processeur possède un résidu plus petit qu'un de ses voisins, cela signifie que son calcul est plus proche de la convergence que celui de son voisin. Pour éviter de tenir compte des variations rapides de résidus locaux, dûs par exemple aux retards de communications, on peut prendre en compte le résidu de quelques itérations précédentes. Suivant la modélisation du problème mis en œuvre, il n'est pas forcément possible d'identifier le bloc de composantes responsable de la valeur élevée du résidu. C'est pourquoi, généralement, le transfert des composantes sur les frontières permet de conserver la localité des communications pour de nombreuses applications. Si le choix des composantes à transférer n'est pas optimal, c'est-à-dire si le résidu ne change pas beaucoup avec un seul transfert, alors au cours des prochaines itérations d'équilibrage le résidu finira par diminuer. En effet, les itérations s'exécuteront plus rapidement en raison de la diminution de nombre de composantes.

Nous illustrons le mécanisme d'équilibrage en fonction du résidu dans la figure 8.1 avec deux processeurs. À chaque itération l'erreur est déterminée pour l'ensemble des composantes de chaque processeur. Sur la figure, dès la première itération, le processeur 1 envoie une partie de sa charge à son voisin dont le résidu est plus faible. Le transfert de charge prend un certain temps, pendant lequel les nœuds poursuivent leurs itérations. Après 3 itérations, la charge est transférée et les processeurs répercutent le transfert sur leurs composantes. Le processeur 2, qui effectuait déjà une itération moins rapidement que le processeur 1, se retrouve avec plus de composantes et itère encore moins vite. Pendant ce temps, le processeur 1 fait évoluer son résidu rapidement. Finalement, les deux processeurs possèdent un résidu comparable après un certain temps.

Au niveau de la programmation, il paraît indispensable de garantir que les messages contenant des données à transférer soient parfaitement traités. Pour cela, il convient d'utiliser des mutex afin d'empêcher plus d'un envoi de charge de la part d'un processeur à un même voisin.

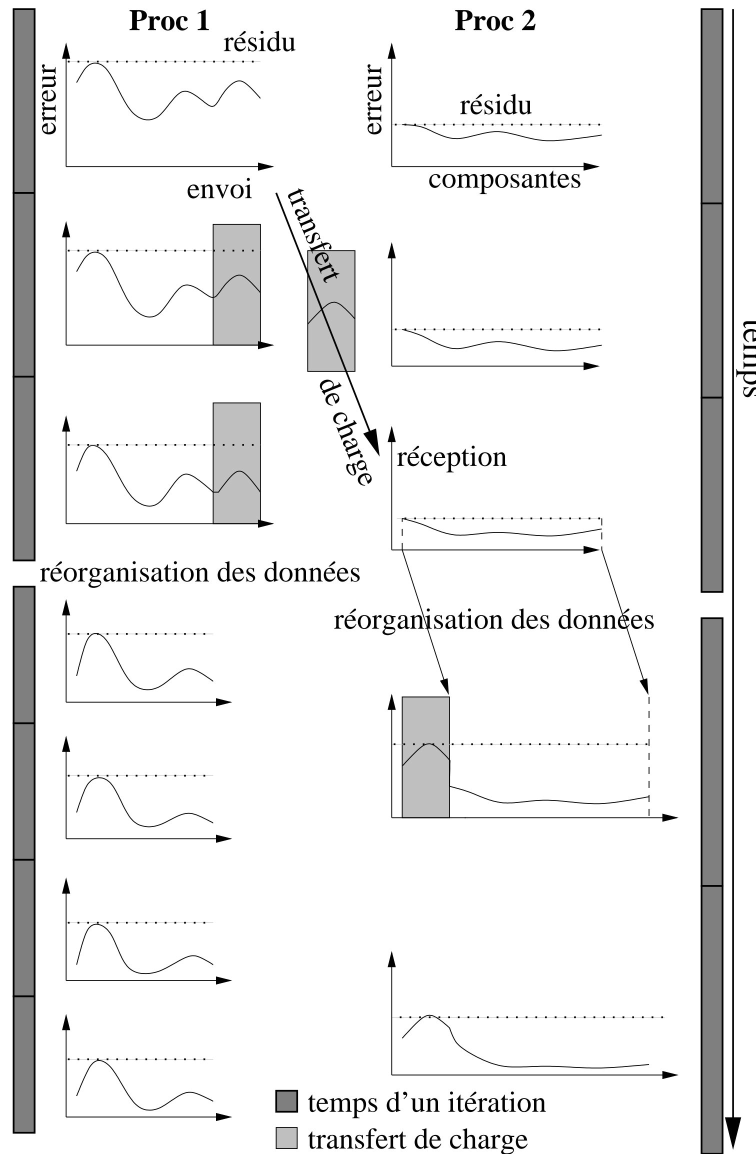

FIG. 8.1 – Transfert de charge basé sur le résidu entre 2 processeurs

Après avoir présenté l'algorithme et les deux choix possibles pour l'estimation de la charge, nous allons étudier leur comportement sur une application.

8.4 Illustration sur un exemple

Dans un premier temps nous présentons le problème, puis nous analysons les expérimentations.

8.4.1 Présentation du problème

L'application que nous avons choisie permet de modéliser un mécanisme de réaction chimique conduisant à une réaction qui oscille. Cette réaction transforme deux éléments chimiques A et B en deux autres C et D en suivant les règles suivantes :

$$\begin{array}{rcl} A & \rightarrow & X \\ 2X+Y & \rightarrow & 3Y \\ B+X & \rightarrow & Y+C \\ X & \rightarrow & D \end{array} \tag{8.1}$$

Cette réaction entraîne un phénomène d'autocatalyse et lorsque les concentrations de A et B sont maintenues constantes, les concentrations de X et Y oscillent avec le temps. Quelles que soient les concentrations initiales de X et Y, la réaction converge vers ce qu'on appelle un cycle de limite de la réaction. C'est un graphe qui représente la concentration de X par rapport à celle de Y et réciproquement.

Cette équation permet de calculer l'évolution des concentrations u et v des deux éléments X et Y sur la discrétisation de l'espace en une dimension en fonction du temps. En considérant que l'espace est discrétisé en N points, l'évolution des concentrations u_i et v_i en chaque point de l'espace pour $i = 1, \ldots, N$ est donnée par les équations suivantes :

$$\begin{array}{l} u_i' = 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1}) \\ v_i' = 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1}) \end{array} \tag{8.2}$$

Les conditions aux frontières sont :

$$\begin{array}{rcccl} u_0(t) & = & u_{N+1}(t) & = & \alpha(N+1)^2 \\ v_0(t) & = & v_{N+1}(t) & = & 3 \end{array}$$

et les conditions initiales pour chaque $i \in \{1, ..., N\}$ sont :

$$\begin{array}{l} u_i(0) = 1 + sin(2\pi x_i) \;\text{ avec }\; x_i = \dfrac{i}{N+1} \\ v_i(0) = 3 \end{array}$$

Pour ce problème, $\alpha = \frac{1}{50}$.

Pour résoudre ce problème nous utilisons la technique de relaxation d'ondes. Nous n'allons pas présenter en détail cette méthode. Elle est très facilement parallélisable, cependant elle nécessite de nombreuses itérations. Dans le chapitre 7, nous présentons

une autre méthode plus performante. Pour davantage d'explications sur la méthode de relaxation d'ondes, nous invitons le lecteur intéressé à consulter [87, 67, 68, 43]. Les explications concernant l'implantation de l'algorithme sont détaillées dans [14] et [16].

8.4.2 Expérimentations

Pour réaliser nos expérimentations, nous avons utilisé PM2. L'espace est discrétisé en 60000 points, l'intervalle de temps étudié est $[0, 10]$ avec un pas de temps de 0.05. Dans la suite, tous les résultats sont les moyennes de 20 expérimentations.

Afin de mesurer la différence entre les deux estimateurs de charge, nous comparons leur comportement avec une première série d'expérimentations. Pour cela, nous avons utilisé une grappe distante composée de 10 machines hétérogènes. Les machines utilisées varient d'un Pentium II 450Mhz jusqu'à un Pentium IV 2.4Ghz et sont réparties sur les sites de Belfort, Besançon et Montbéliard qui sont reliés entre eux en 10Mb/s. Pour cette expérimentation nous mesurons les temps d'exécution en fonction de la précision choisie. Les résultats sont illustrés sur la figure 8.2.

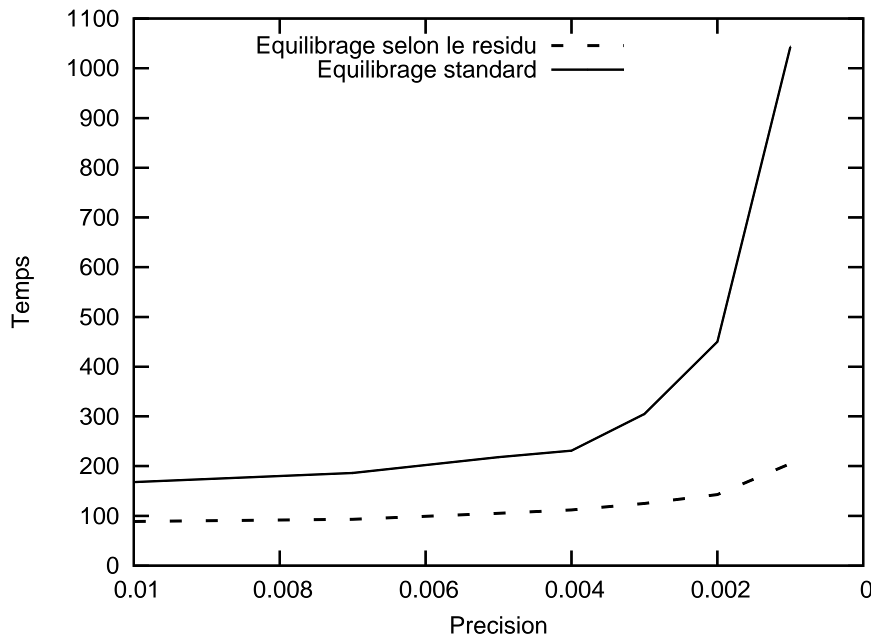

FIG. 8.2 – Temps d'exécution (en secondes) avec les deux estimateurs de charge

Cette courbe illustre clairement que l'estimateur basé sur le résidu est plus efficace que la version standard, basée sur le nombre de composantes. En outre, on constate que plus la précision est importante, plus la différence entre les deux versions se creuse. Ceci s'explique par le fait qu'une précision plus fine nécessite un plus grand nombre d'itérations pour atteindre le seuil de convergence, et donc d'un temps plus important. La version basée sur le résidu tire pleinement profit d'un grand nombre d'itérations

pour équilibrer au mieux l'ensemble des processeurs. En effet, dans ce cas, les processeurs effectuent des itérations qui font progresser le calcul le plus rapidement possible vers la solution.

Afin de mesurer l'efficacité de l'équilibrage de charge en fonction du résidu par rapport à une version sans équilibrage, nous comparons les temps d'exécution des deux versions sur une grappe locale homogène. Il s'agit du Icluster installée à Grenoble[1]. Les machines utilisées sont des Pentium III 733Mhz avec 256Mo de mémoire, le réseau est composé de 5 switch 1Gb/s, chaqun reliant environ 40 machines avec un débit de 100mb/s. La précision du problème est fixée à $7e-4$.

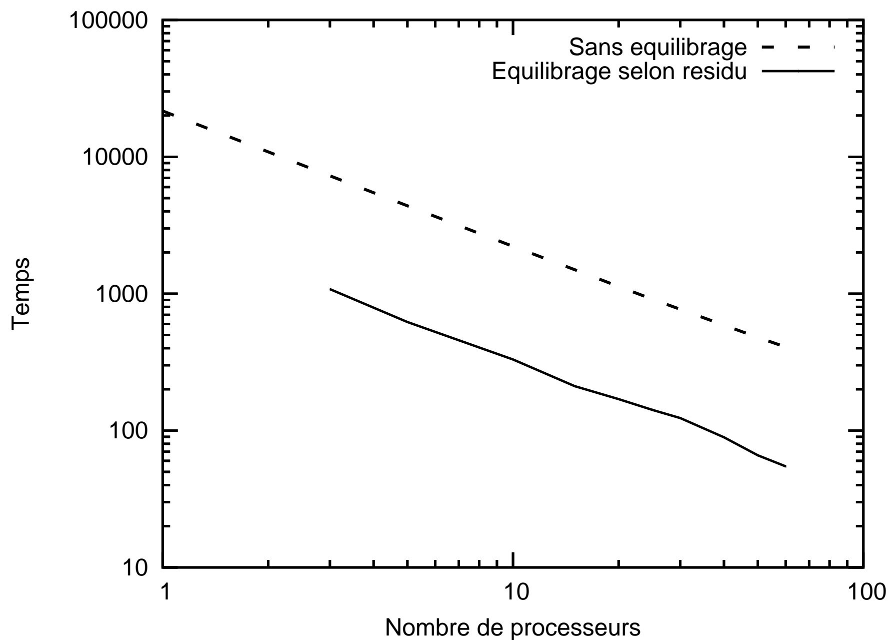

FIG. 8.3 – Temps d'exécution sur une grappe homogène

La figure 8.3 illustre clairement que les deux versions supportent très bien le passage à l'échelle. L'algorithme de relaxation d'ondes est bien connu pour cela. Cependant, on constate que l'équilibrage de charge n'a pas de surcoût visible sur cet algorithme. Cela provient du fait que notre équilibrage de charge n'est pas centralisé. De plus, la différence entre les deux versions d'algorithmes reste stable lorsque le nombre de processeurs croit. En fait, le ratio entre les deux courbes varie entre 6.2 et 7.4 avec une moyenne de 6.8. Ce résultat démontre l'efficacité de notre algorithme de couplage d'équilibrage de charge basé sur le résidu avec un algorithme IACA.

Finalement, nous avons effectué un dernier test sur une grappe distante hétérogène. Cette fois-ci, nous avons utilisé 15 machines réparties sur trois sites : Belfort, Montbéliard et Grenoble. La configuration des machines varie entre un Pentium II 400Mhz et un Athlon 1.6Ghz et nous les avons connectées de telle manière que

[1]Cette grappe n'est plus en service actuellement

chacune possède des voisins distants. Ceci permet de mieux simuler la distance entre les sites. Nous utilisons toujours la version d'équilibrage basée sur le résidu. Les résultats sont synthétisés dans la table 8.1.

version	sans équilibrage	équilibrage	ratio
temps d'exécution	515.3	105.5	4.88

TAB. 8.1 – Temps d'exécution sur une grappe distante hétérogène

Dans ce contexte, la version équilibrée s'exécute également plus vite que la version sans équilibrage. Le ratio devient plus faible que sur la grappe homogène car les transferts de charge prennent plus de temps, en raison de la bande passante plus faible. Ceci dit, les algorithmes IACA sont plus performants dans un contexte distant. C'est pourquoi la possibilité de réduire leur temps d'exécution les rend encore plus attractifs.

8.5 Conclusion

Ce chapitre illustre l'intérêt du couplage de deux techniques utilisées pour accélérer l'exécution d'algorithmes parallèles. L'asynchronisme permet de recouvrir les communications par du calcul, et l'équilibrage de charge permet de répartir au mieux la charge. Dans le contexte des algorithmes IACA, une version standard de l'équilibrage selon le nombre de composantes du problème n'est pas forcément la meilleure stratégie. L'estimateur que nous avons défini est basé sur le résidu local de chaque processeur ; il tend à faire converger tous les processeurs en même temps. Nous avons expérimenté ce couplage dans un contexte local et homogène et dans un contexte distant et hétérogène. Pour ces deux contextes, les résultats sont très intéressants.

Conclusion - Perspectives

Les travaux présentés dans ce document reflètent mes activités de recherche depuis ma nomination en tant que maître de conférences à Belfort au sein du LIFC en Septembre 2000. Ces travaux s'inscrivent dans la thématique du calcul distribué sur grappes de processeurs. Ils ont pour objectif de contribuer aux développements d'applications efficaces et robustes sur grappes distantes. Ce document est structuré en deux parties.

La première partie présente les algorithmes que nous avons définis pour équilibrer la charge d'un réseau de machines sur topologie dynamique. Définir de tels algorithmes, tolérants la surcharge ou la coupure des liens de communication, est plus que d'actualité, avec le développement des applications distribuées. Les algorithmes que nous avons définis sont des adaptations des algorithmes existants pour l'équilibrage de charge sur topologie statique. Pour le moment, nous supposons que le nombre de processeurs ne varie pas au cours du temps. L'hypothèse permettant d'établir la preuve de convergence vers l'équilibre des algorithmes est réaliste. Il est même possible que le graphe de communication soit toujours non connecté.

La seconde partie de ce document concerne les algorithmes itératifs asynchrones. Nous les appelons IACA pour algorithmes à Itérations Asynchrones et Communications Asynchrones. Leur particularités sont les suivantes : ils permettent le recouvrement des communications par du calcul ; ils supportent les pertes de messages ; les processeurs calculent à leur propre vitesse sans se soucier de l'avancement de leurs voisins, parce qu'il n'y a pas de synchronisation entre les processeurs. Ces propriétés confèrent aux algorithmes IACA un intérêt particulier pour développer des applications scientifiques capables d'être exécutées sur des grappes de processeurs. Ces algorithmes n'ont pratiquement jamais été utilisés dans un contexte de grilles composées de sites distants. Notre travail a consisté à montrer qu'il était relativement simple de les programmer avec différents environnements de programmation multithreadés.

De plus, nous avons montré comment coupler un algorithme d'équilibrage de charge à un algorithme IACA. Dans ce cas, nous avons défini un évaluateur de charge basé sur le résidu du calcul. Cette technique a pour but d'essayer de faire converger les processeurs dans le même laps de temps. L'objectif escompté est ainsi de diminuer au maximum le temps d'exécution, et ce, de manière dynamique au cours du déroulement de l'algorithme.

Afin de valider notre travail sur les algorithmes IACA, nous avons toujours cherché à étudier le comportement d'applications scientifiques sur grilles composées de sites distants. Les techniques de multisplitting permettant d'élaborer des algorithmes parallèles à gros grains, donc adaptés au contexte distant, procurent une certaine facilité pour développer de nouveaux algorithmes à partir d'algorithmes séquentiels. À travers les expérimentations que nous avons menées, nous avons pu comparer le comportement des algorithmes exécutés, en version synchrone et asynchrone, dans diverses configurations.

Finalement, le développement d'un algorithme IACA est, de notre point de vue, plus facile à entreprendre en procédant de la manière suivante. Dans un premier temps, il faut étudier le passage d'un algorithme séquentiel à un algorithme parallèle synchrone capable d'être désynchronisé. Lorsque celui-ci est opérationnel, il suffit de modifier les deux points différenciant le mode synchrone du mode asynchrone, à savoir la gestion des communications et la détection de convergence.

Perspectives

L'aspect dynamique des réseaux de communications ainsi que des applications scientifiques et numériques est un élément de plus en plus important. Il se décline sous plusieurs formes selon le contexte étudié.

Les applications scientifiques utilisant un très grand nombre de ressources processeurs (plusieurs milliers ou millions) sont forcément confrontées à l'apparition ou à la disparition de nœuds. Concernant l'équilibrage de charge, il semble intéressant d'étudier comment ajouter ou supprimer des processeurs au cours de l'exécution et assurer la convergence vers l'équilibre de la charge sur les nœuds. L'émergence des réseaux sans fils ou réseaux ad-hoc [90, 77, 82, 88] et des réseaux de capteurs [62, 2] illustre pleinement le besoin de ce caractère dynamique. Ce type de réseaux assurera certainement un avenir prometteur aux algorithmes d'équilibrage de charge décentralisés sur réseaux dynamiques.

Cependant, l'équilibrage de charge avec nœuds volatiles n'a d'intérêt que si les environnement de programmation et les applications scientifiques supportent eux aussi la volatilité des processeurs. Actuellement, les bibliothèques de communications tentent, de plus en plus, d'être tolérantes aux fautes. Ces mécanismes permettent de supporter la volatilité des nœuds si l'application effectue une sauvegarde régulière des données [85, 70]. Ceci engendre des communications supplémentaires et bien souvent des synchronisations. Les travaux actuels sur les algorithmes asynchrones n'ont pas encore étudié la volatilité des processeurs. Ces algorithmes semblent bien adaptés au nouveau support de calcul que représentent les systèmes de calculs globaux [45, 6] et les systèmes de calculs pair-à-pair [95, 108]. Nous différencions un système de calcul global d'un système de calcul pair-à-pair, par le fait que le premier possède un système

centralisateur que le second n'a pas. Ces systèmes sont amenés à manipuler un très grand nombre de processeurs. Actuellement, seules des applications relativement découplées, c'est-à-dire, sans communication, ou très peu entre les nœuds de calcul, ont montré leur efficacité. La raison provient peut-être du fait que la communauté scientifique connaît mal les algorithmes IACA, ou tout du moins, la manière de les implanter. Il est naturel de penser que les algorithmes IACA sont plus adaptés au contexte de calcul à grand échelle, mais la contrainte de volatilité des processeurs introduit de nombreux problèmes qu'il faudra étudier. Parmi ces problèmes, nous pouvons d'ores et déjà relever la difficulté de concevoir une bibliothèque facilement déployable sur des architectures et des systèmes d'exploitation hétérogènes. Une telle bibliothèque devra être multithreadée, comme nous l'avons vu dans ce document. Ensuite, comme nous l'avons évoqué, il semble nécessaire d'avoir un mécanisme de sauvegarde ou de réplication pour gérer les pertes de nœuds. Les algorithmes centralisés sont à proscrire sur ce type d'infrastructure. En définitive, il reste un travail très important à effectuer dans cette voie.

Concernant les systèmes linéaires, l'utilisation d'un algorithme itératif à l'intérieur de l'itération de multisplitting pourrait se révéler efficace. D'autre part, l'élargissement des matrices pouvant être traitées par ce principe nous semble intéressant. Cet objectif est envisageable par l'intermédiaire, entre autres, des techniques de renumérotation ou de préconditionnement de la matrice. Concernant les algorithmes non linéaires, il est certainement possible d'étudier d'autres algorithmes basés sur l'algorithme de Newton. De manière générale, tout laisse à penser que de nombreux autres algorithmes peuvent s'adapter au contexte de l'asynchronisme, même si ceci nécessite forcément une étude minutieuse de la convergence.

Bibliographie

Mes travaux présentés dans cette bibliographie sont [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28].

[1] Y. Afek and M. Saks. Detecting global termination conditions in the face of uncertainty. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 109–124. ACM Press, 1987.

[2] I. F. Akyildiz, S. Weilian, Y. Sankarasubramaniam, and E. E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8) :102–114, 2002.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Eng.*, 184 :501–520, 2000.

[4] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4) :388–421, 2001.

[5] 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, 1 May 1996.

[6] L. Aouad and S. Petiton. Experimentations and programming paradigms for matrix computing on peer to peer grid. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 306 – 311, 2004.

[7] J. Arnal, V. Migallon, and J. Penadés. Nonstationary parallel newton iterative methods for nonlinear problems. In *VECPAR'2000*, volume 1981, pages 380–394. Lectures Notes in Computer Science, 2001.

[8] J. Y. Astic, A. Bihain, and M. Jerosolimski. The mixed adams-bdf variable step size algorithm to simulatetransient and long term phenomena in power systems. *IEEE Transactions on Power Systems*, 9(2) :929–935, 1994.

[9] O. Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes*. PhD thesis, École normale supérieure de Lyon, 2002.

[10] O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine : a True Multi-Protocol MPI for High-Performance Networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, April 2001. IEEE.

[11] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[12] J. M. Bahi. Méthodes itératives dans des espaces produits. applications au calcul parallèle, 1998. Habilitation à Diriger des Recherches.

[13] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Asynchronism for iterative algorithms in a global computing environment. In *International Conference on High Performance Computing Systems and Applications, HPCS'02*, pages 90–97. IEEE Computer Society Press, 2002.

[14] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *17th IEEE and ACM int. conf. on International Parallel and Distributed Processing Symposium, IPDPS 2003*, pages 40a, 9 pages. IEEE computer society press, 2003.

[15] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. In *18th IEEE and ACM International Conference on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 247b, 8 pages. IEEE Computer Society Press, 2004.

[16] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(4) :289–299, 2005. Version étendue de [14].

[17] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5) :439–461, 2005.

[18] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *Journal of Supercomputing*, 2005. Sous presse, version étendue de [15].

[19] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(1) :4–13, 2005.

[20] J. M. Bahi and R. Couturier. Parallelization of direct algorithms using multisplitting methods in grid environments. In *19th IEEE and ACM International Conference on Parallel and Distributed Processing Symposium, IPDPS 2005*, pages 254b, 8 pages. IEEE Computer Society Press, 2005.

[21] J. M. Bahi, R. Couturier, K. Mazouzi, and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Applied Mathematical Modelling*, 2005. Sous presse.

[22] J. M. Bahi, R. Couturier, and M. Salomon. Solving three-dimensional transport models with synchronous and asynchronous iterative algorithms in a grid computing environment. In *19th IEEE and ACM International Conference on Parallel and Distributed Processing Symposium, IPDPS 2005*, pages 253b, 7 pages. IEEE Computer Society Press, 2005.

[23] J. M. Bahi, R. Couturier, and F. Vernier. Accelerated diffusion algorithms on general dynamic networks. In *Proceedings of 5th International Conference, PPAM Czestochowa, Poland*, volume 3019 of *LNCS*, pages 77–82. PPAM, Springer-Verlag, 2003.

[24] J. M. Bahi, R. Couturier, and F. Vernier. Broken edges and dimension exchange algorithm on hypercube topology. In *Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 140–145. Euromicro, IEEE Computer Society Press, February 5–7, 2003.

[25] J. M. Bahi, R. Couturier, and F. Vernier. Synchronous distributed load balancing on dynamic networks. *Journal of Parallel and Distributed Computing*, 65(11) :1397–1405, 2005.

[26] J. M. Bahi, R. Couturier, and P. Vuillemin. Asynchronous iterative algorithms for computational science on the grid : three case studies. In *Vecpar 2004*, volume 2, pages 597–609, Valencia, Spain, 2004.

[27] J. M. Bahi, R. Couturier, and P. Vuillemin. Asynchronous iterative algorithms for computational science on the grid : three case studies. In *procs. of Vecpar 2004*, volume 3402 of *LNCS*, pages 302–314, Valencia, Spain, June 2004. Springer-Verlag. Sélection des meilleurs papiers de Vecpar, version améliorée de [26].

[28] J. M. Bahi, R. Couturier, and P. Vuillemin. Solving nonlinear wave equations in the grid computing environment : an experimental study. *Journal of Computational Acoustics*, 2005. Sous presse.

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

[30] J. M. Bahi and J. Gaber. Load balancing on networks with dynamically changing topology. In *Europar 2001 conference, Lecture Notes on Computer Science*, pages 175–182, Manchester, UK, 2001.

[31] J. M. Bahi, J.-C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3,4) :315–345, 1997.

[32] J.M. Bahi, S. Domas, and K. Mazouzi. Combination of java and asynchronism for the grid : a comparative study based on a parallel power method. In *18th IEEE and ACM Int. Conf. on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 158a, 8 pages, Santa Fe, USA, April 2004. IEEE computer society press.

[33] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI : An efficient implementation of MPI for IBM RS/6000 SP systems. *IEEE Trans. Parallel and Distrib. Systems*, 12(10) :1081–1093, 2001.

[34] Gérard M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 25(2) :226–244, 1978.

[35] D. El Baz. *Contribution à l'algorithmique parallèle, le concept d'asynchronisme : étude théorique, mise en oeuvre, et application. HDR*. LAAS CNRS, Institut National Polytechnique de Toulouse, 1998.

[36] D. El Baz, P. Spiteri, J.C. Miellou, and D. Gazen. Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. *Journal of Parallel and Distributed Computing*, 38(1) :1–15, 1996.

[37] A. Berman and R. J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences*. Academic Press, SIAM, Philadelphia, third edition, 1979 edition, 1994.

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

[39] D. P. Bertsekas and J. N. Tsitsiklis. Parallel and distributed iterative algorithms : a selective survey. *Automatica*, 25 :3–21, 1991.

[40] D. A. Bini. Numerical computation of polynomial zeros by means of alberth's method. *Numerical Algorithms*, 13 :179–200, 1996.

[41] L. G. Birta and O. Abou-Rabia. Parallel block predictor-corrector methods for ode's. *IEEE Transactions on Computers*, 36(3) :299–311, 1987.

[42] J. E. Boillat. Load balancing and poisson equation in a graph. *Concurrency : Practice and Experience.*, 2(4) :289–313, 1990.

[43] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press Inc., New York, 1995.

[44] G. D. Byrne and A. C. Hindmarsh. Pvode, an ode solver for parallel computers. *Int. J. High Perf. Comput. Apps.*, 14(4) :354–365, 1999.

[45] F. Cappello, A. Djilali, G. Fedak, C Germain, O. Lodygensky, and V. Néri. *Calcul réparti à grande échelle Metacomputing*, chapter XtremWeb, une plate-forme de recherche sur le Calcul Global et Pair à Pair. Hermes Science - Lavoisier, 2002.

[46] A. S. Charão. *Multiprogrammation parallèle générique des méthodes de décomposition de domaine*. PhD thesis, Institut National Polytechnique de Grenoble, 2001.

[47] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2 :199–222, 1969.

[48] N. H. Cong, K. Strehmel, R. Weiner, and H. Podhaisky. *Advances in Computational Mathematics*, volume 10, chapter Runge Kutta Nyström-type parallel block predictor-corrector methods, pages 115–133. Springer Science+Business Media B.V., 1999.

[49] W. Craig and C. Wayne. Newton's method and periodic solutions of nonlinear wave equations. *Commun. Pure Appl. Math.*, 46 :1409–1498, 1993.

[50] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7 :279–301, 1989.

[51] T. Davis. University of florida sparse matrix collection. voir http ://www.cise.ufl.edu/research/sparse/matrices/.

[52] M. Daydé and I. Duff. Porting industrial codes and developing sparse linear solvers on parallel computers. *Computing Systems in Engineering*, 6(4-5) :295–305, 1995.

[53] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25 :289–313, 1998.

[54] T. B. Downing. *Java RMI : Remote Method Invocation*. IDG Books, 1998.

[55] 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.

[56] M. El Tarazi. *Contraction et ordre partiel pour l'étude d'algorithmes synchrones et asynchrones*. PhD thesis, Université de Franche-Comté, 1981.

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

[58] R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35(3) :305–320, 2002.

[59] R. Elsässer, B. Monien, G. Rote, and S. Schamberger. Toward optimal diffusion matrices. In *16th International Parallel and Distributed Processing Symposium. IPDPS 2002, Proceedings.* IEEE Computer Society Press, May 2002.

[60] R. Elsässer, B. Monien, and S. Schamberger. Load balancing in dynamic networks. In *7th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2004)*, pages 193–200. IEEE Computer Society, May 2004.

[61] N. Emad. Méthodes hybrides et métacomputing pour le calcul intensif scientifique, 2001. Habilitation à Diriger des Recherches.

[62] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges : Scalable coordination in sensor networks. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom-99)*, pages 263–270, 1999.

[63] S. Fiorini and R. J. Wilson. Edge-coloring of graphs. In *L.W. Beineke and R.J. Wilson, editors, Selected topics in graph theory.* Academic Press, 1978.

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

[65] A. Frommer and D. B. Szyld. On asynchronous iterations. *J. Comp. Appl. Math.*, 123 :201–216, 2000.

[66] D. Furihata. 'finite-difference schemes for nonlinear wave equation that inherit energy conservation property. *Journal of Computational and Applied Mathematics*, 134 :37–57, 2001.

[67] C. W. Gear. *Numerical initial value problems in ordinary differential equations*. Prentice-Hall Inc., New Jersey, 1971.

[68] C. W. Gear. Massive parallelism across space in ODEs. *Applied Numerical Mathematics : Transactions of IMACS*, 11(1–3) :27–43, January 1993.

[69] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[70] C. Germain, G. Fedak, V. Néri, and F. Cappello. Global computing systems. *Lecture Notes in Computer Science*, 2179 :218–227, 2001.

[71] B. Ghosh, S. Muthukrishnan, and M. H. Schultz. First and second order diffusive methods for rapid, coarse, distributed load balancing. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 72–81, Padua, Italy, June 24–26, 1996. SIGACT/SIGARCH. Extended abstract.

[72] S. J. Gortler, M. Cohen, and P. Slusallek. Radiosity and relaxation methods. *IEEE Computer Graphics and Applications*, 14(6) :48–58, 1994.

[73] L. Grigori and X. S. Li. A new scheduling algorithm for parallel sparse lu factorization with static pivoting. In *Super Computing 2002*. IEEE computer society press and ACM sigarch, 2002. paper 139 on CD.

[74] W. Gropp and E. Lusk. Sowing MPICH : A case study in the dissemination of a portable environment for parallel scientific computing. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :103–114, 1997.

[75] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message passing interface*. MIT Press, 1994.

[76] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3) :301–324, 2002.

[77] P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, 46(2) :388–404, 2000.

[78] E. Hairer and G. Wanner. *Solving ordinary differential equations II : Stiff and differential-algebraic problems*, volume 14 of *Springer series in computational mathematics*, pages 5–8. Springer-Verlag, Berlin, 1991.

[79] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing*, 10 :160–166, 1990.

[80] P. Hénon, P. Ramet, and J. Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2) :301–321, January 2002.

[81] S. L. S. Jacoby, J. S. Kowalik, and J. T. Pizzo. *Iterative Methods for Nonlinear Optimization Problems*. Prentice Hall, 1972.

[82] D. B. Johnson and D. A. Maltz. *In Mobile Computing*, chapter Dynamic Source Routing in Ad Hoc Wireless Networks, pages 153–181. Kluwer Academic Publishers, 1996.

[83] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2 : A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5) :551–563, 2003.

[84] K. R. Khusnutdinova. Nonlinear waves in a bi-layer and coupled klein gordon equation. In A.B. Movchan, editor, *Symposium "Asymptotics, Singularities and Homogenisation in Problems of Mechanics"*, pages 537–546. Kluwer Academic Publishers, 2003.

[85] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, 13(1) :23–31, 1987.

[86] A. Legrand and Y. Robert. *Algorithmique parallèle - Cours et exercices corrigés*. Dunod, 2003.

[87] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli. The wavefront relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, CAD-1 :131–145, 1982.

[88] J. Li, C. Blake, D. S .J. De Couto, H. I. Lee, and R. Morris. Capacity of ad hoc wireless networks. In *MobiCom '01 : Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 61–69. ACM Press, 2001.

[89] X. S. Li and J. W. Demmel. SuperLU_DIST : A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2) :110–140, June 2003.

[90] C. R. Lin and M. Gerla. Adaptive clustering for mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 15(7) :1265–1275, 1997.

[91] T. Loos and R. Bramley. MPI performance on the SGI Power Challenge. In IEEE, editor, *Proceedings. Second MPI Developer's Conference : Notre Dame, IN, USA, 1–2 July 1996*, pages 203–206, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[92] K. Mazouzi. *JACE : Un environnement d'exécution distribué pour le calcul itératif asynchrone*. PhD thesis, Université de Franche-Comté, 2005. En cours de finalisation.

[93] G. Mercier. *Communications à hautes performances portables en environnements hiérarchiques, hétérogènes et dynamiques*. PhD thesis, Université de Bordeaux, 2004.

[94] J.-C. Miellou. Algorithmes de relaxation chaotique à retards. *R.A.I.R.O.*, 1 :55–82, 1975.

[95] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, , and Z. Xu. Peer-to-peer computing. Technical report, HP Laboratories Palo Alto, 2002. HPL-2002-57.

[96] M. Mitchell. *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. MIT-Press, Cambridge, 1996.

[97] B. Mohammadi and J.-P. Saïac. *Pratique de la simulation numérique*. Dunod, 2003.

[98] MPI Forum. Special issue : MPI2 : A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 12(1–2) :1–299, Spring–Summer 1998.

[99] R. Namyst and J.-F. Méhaut. PM^2 : Parallel multithreaded machine. A computing environment for distributed architectures. In *Parallel Computing : State-of-the-Art and Perspectives, ParCo'95*, volume 11, pages 279–285. Elsevier, North-Holland, 1996.

[100] A. Pope. *The CORBA Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Reading, MA, USA, December 1997.

[101] W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery. *Numerical Recipes in C++ : the art of scientific computing*. Cambridge University Press, 2002.

[102] C. Pérez, T. Priol, and A. Ribes. A parallel corba component model for numerical code coupling. In *3rd International Workshop on Grid Computing*, volume 2536 of *LNCS*, pages 88–99. Springer-Verlag, 2002.

[103] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, New York, 1996.

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

[105] IEEE Computer Society. Ieee standard for a high performance serial bus. Technical Report Std 1394-1995, IEEE, August 1996.

[106] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, 2003. Springer-Verlag.

[107] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf, 2004. http ://dast.nlanr.net/Projects/Iperf/.

[108] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. *Lecture Notes in Computer Science*, 2536 :1–12, 2002.

[109] F. Vernier. *Algorithmique itéative pour l'équilibrage de charge dans les réseaux dynamiques*. PhD thesis, Université de Franche-Comté, 2004.

[110] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965.

[111] C. Z. Xu and F. C. M. Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16(4) :385–393, 1992.

[112] C. Z. Xu and F. C. M. Lau. Optimal parameters for load balancing with the diffusion method in mesh networks. *Parallel Processing Letters*, 4(1-2) :139–147, 1994.

[113] C. Z. Xu, B. Monien, R. Lüling, and F. C. M. Lau. An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers. In *Proc. of the 9th International Parallel Processing Symposium*, pages 472–479. IEEE Computer Society Press, 1995.

[114] Y. Zhao, W. Zhou, J. Huang, S. Yu, and E. J. Lanham. Self-adaptive clock synchronization for computational grid. *Journal of Computer Science and Technology archive*, 18 :434 – 441, 2003.

Liste des publications

Revues internationales

[1] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(4) :289–299, 2005. Version étendue de [19].

[2] J. M. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(1) :4–13, 2005.

[3] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5) :439–461, 2005.

[4] J. M. Bahi, R. Couturier, and F. Vernier. Synchronous distributed load balancing on dynamic networks. *Journal of Parallel and Distributed Computing*, 65(11) :1397–1405, 2005.

[5] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *Journal of Supercomputing*, 2005. Sous presse, version étendue de [15].

[6] J. M. Bahi, R. Couturier, K. Mazouzi, and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Applied Mathematical Modelling*, 2005. Sous presse.

[7] J. M. Bahi, R. Couturier, and P. Vuillemin. Solving nonlinear wave equations in the grid computing environment : an experimental study. *Journal of Computational Acoustics*, 2005. Sous presse.

[8] R. Couturier and C. Chipot. Parallel molecular dynamics using OpenMP on a shared memory machine. *Computer Physics Communications*, 124 :49–59, 2000.

Revues nationales

[9] R. Couturier and F. Spies. Extraction de racines dans des polynômes creux de degré élevés. *Réseaux et Systèmes Répartis, Calculateurs Parallèles*, 14(1) :67–81, 2001.

[10] R. Couturier. Trois expérimentations de simulations parallèles. *Technique et science informatique*, 19(5) :625–648, 2001.

Chapitres de livres

[11] R. Gras, R. Couturier, J. Blanchard, H. Briand, P. Kuntz, and P. Peter. *Mesures de qualité pour la fouille de données*, chapter Quelques critères pour une mesure de qualité de règles d'association. Un exemple : l'implication statistique, pages 3–32. RNTI-E-1, Cepaduès Editions, 2004.

[12] R. Couturier. *Learning in mathematics and science and educational technology*, chapter Learning in mathematics and science and educational technology, pages 369–376. A. Gagatsis, University of Chypres, 2001.

Conférences internationales avec comité de lecture

[13] J. M. Bahi and R. Couturier. Parallelization of direct algorithms using multisplitting methods in grid environments. In *19th IEEE and ACM International Conference on Parallel and Distributed Processing Symposium, IPDPS 2005*, pages 254b, 8 pages. IEEE Computer Society Press, 2005.

[14] J. M. Bahi, R. Couturier, and M. Salomon. Solving three-dimensional transport models with synchronous and asynchronous iterative algorithms in a grid computing environment. In *19th IEEE and ACM International Conference on Parallel and Distributed Processing Symposium, IPDPS 2005*, pages 253b, 7 pages. IEEE Computer Society Press, 2005.

[15] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. In *18th IEEE and ACM International Conference on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 247b, 8 pages. IEEE Computer Society Press, 2004.

[16] J. M. Bahi, R. Couturier, and P. Vuillemin. Asynchronous iterative algorithms for computational science on the grid : three case studies. In *procs. of Vecpar 2004*, volume 3402 of *LNCS*, pages 302–314, Valencia, Spain, June 2004. Springer-Verlag. Sélection des meilleurs papiers de Vecpar, version améliorée de [18].

[17] R. Couturier, R. Gras, and F. Guillet. Reducing the number of variables using implicative analysis. In *International Federation of Classification Societies, IFCS 2004*, pages 277–285. Springer Verlag : Classification, Clustering, and Data Mining Applications, 2004.

[18] J. M. Bahi, R. Couturier, and P. Vuillemin. Asynchronous iterative algorithms for computational science on the grid : three case studies. In *Vecpar 2004*, volume 2, pages 597–609, Valencia, Spain, 2004.

[19] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *17th IEEE and ACM int. conf. on International Parallel and Distributed Processing Symposium, IPDPS 2003*, pages 40a, 9 pages. IEEE computer society press, 2003.

[20] J. M. Bahi, R. Couturier, and F. Vernier. Accelerated diffusion algorithms on general dynamic networks. In *Proceedings of 5th International Conference, PPAM Czestochowa, Poland*, volume 3019 of *LNCS*, pages 77–82. PPAM, Springer-Verlag, 2003.

[21] J. M. Bahi, R. Couturier, and F. Vernier. Broken edges and dimension exchange algorithm on hypercube topology. In *Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 140–145. Euromicro, IEEE Computer Society Press, February 5–7, 2003.

[22] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Asynchronism for iterative algorithms in a global computing environment. In *International Conference on High Performance Computing Systems and Applications, HPCS'02*, pages 90–97. IEEE Computer Society Press, 2002.

[23] R. Couturier, P. Canalda, and F. Spies. Iterative algorithms on heterogeneous network computing : parallel polynomial root extracting. In *9th International Conference of High Performance Computing, HiPC 2002*, pages 283–291. Springer Verlag, Lecture Notes in Computer Science 2552, 2002.

[24] R. Couturier, B. Couturier, and D. Méry. A compiler for parallel Unity programs using OpenMP. In *Parallel and Distributed Processing Techniques and Applications - PDPTA'99, Las Vegas, USA*, pages 1992–1998, July 1999.

[25] R. Couturier and D. Méry. An experiment in parallelizing an application using formal methods. In *International Conference on Computer Aided Verification - CAV'98, Vancouver, Canada*, pages 345–356. Spinger Verlag, Lecture Notes in Computer Science 1427, June 1998.

[26] R. Couturier and D. Méry. Parallelization of a Monte Carlo simulation of a spins system. In *Parallel and Distributed Processing Techniques and Applications - PDPTA'98, Las Vegas, USA*, pages 1533–1537, July 1998.

[27] R. Couturier. Formal engineering of the bitonic sort using PVS. In Andrew Butterfield and Sharon Flynn, editors, *2nd Irish Workshop in Formal Methods - IWFM'98, Cork, Irland*, pages 20–39, July 1998.

[28] R. Couturier and D. Méry. Coordination of abstract machines. In *CSIT'97, Yerevan, Armenia*, pages 192–200, September 1997.

[29] D. Galmiche and R. Couturier. Guarded Programs and Proofs in Linear Logic. In *Workshop on Proof Theory of Concurrent Object-Oriented Programming - ECOOP'96, Linz, Austria*, July 1996.

[30] R. Couturier and D. Galmiche. Guarded commands and proofs in linear logic. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 525–528. Dpunkt, 1997. Sélection des meilleurs papiers de Vecpar, version améliorée de [29].

Conférences nationales avec comité de lecture

[31] R. Couturier. Un système de recommandation basé sur l'a.s.i. In *Troisième rencontre internationale de l'Analyse Statistique Implicative (ASI3)*, pages 157–162, 2005.

[32] R. Couturier and R. Gras. CHIC : traitement de données avec l'analyse implicative. In *Extraction et gestion des connaissances (EGC'2005)*, volume 2, pages 679–684, 2005.

[33] R. Gras, P. Kuntz, R. Couturier, and F. Guillet. Une version entropique de l'intensité d'implication pour les corpus volumineux. In *EGC'2001*, pages 69–80, 2001.

[34] R. Couturier. Traitements de l'analyse implicative avec chic. In *Journées sur l'implication statistique*, pages 33–50, 2000.

[35] R. Couturier. Couplage OpenMP / MPI : une expérimentation. In *Renpar'2000*, pages 133–138, 2000.

[36] R. Couturier and R. Gras. Introduction de variables supplémentaires dans une hiérarchie de classes et application à CHIC. In *Société Francophone de Classification, SFC'99, Nancy*, pages 87–92, 1999.

[37] R. Couturier. Parallélisation d'une simulation Monte Carlo d'un système de spins et preuve. In *Renpar'10, Strasbourg, France*, pages 179–182, June 1998.

[38] A. Bodin, R. Couturier, and R. Gras. Analyse d'une épreuve de concours par la méthode implicative, présentation interactive. In *Quatrièmes journées de la société française de classification*, Vannes, September 1996.

Mémoires

[39] R. Couturier. *Utilisation des méthodes formelles pour le développement de programmes parallèles*. PhD thesis, Université Henri Poincaré, Nancy, 2000.

[40] R. Couturier. Programmes avec commandes gardées et preuve en logique linéaire. Technical report, LORIA, Rapport de DEA, 1996.

Résumé

Ce document présente la synthèse de nos travaux concernant les algorithmes IACA (Itérations Asynchrones avec Communications Asynchrones) et l'équilibrage de charge pour réseaux dynamiques.
Avec le développement des réseaux à large échelle pour lesquels la topologie peut être dynamique, il paraît important de définir de nouveaux algorithmes d'équilibrage de charge. Partant des algorithmes décentralisés pour topologie statique, tels que la diffusion et GDE, nous les avons adaptés pour qu'ils supportent des liens de communication dynamiques.
Par ailleurs, l'utilisation de grappes de calcul distantes introduit plusieurs nouvelles contraintes par rapport à un contexte de grappes locales homogènes : l'hétérogénéité des machines et des réseaux, des latences de communications fluctuantes et éventuellement importantes, et des débits de communications pouvant être sujets à des perturbations. Dans ce contexte, il est préférable de réduire le nombre de synchronisations et d'utiliser des algorithmes décentralisés, à gros grains. Les algorithmes IACA permettent de répondre à ces critères. Pour implémenter de tels algorithmes efficacement, nous conseillons de partir d'une version synchrone et de modifier la gestion des communications et la détection de convergence. L'utilisation d'un environnement multithreadé permet de séparer le calcul des communications et ainsi de recouvrir les communications sujettes à des perturbations par du calcul.
De plus, nous reportons des expérimentations sur grappes hétérogènes distantes qui permettent de comparer le comportement des versions synchrones et asynchrones. Nous avons étudié des systèmes linéaires et des équations aux dérivées partielles (EDP) en utilisant les techniques de multidécomposition (ou multisplitting).
Finalement, nous étudions comment utiliser une technique d'équilibrage de charge avec les algorithmes IACA.

Mots clefs : algorithmes itératifs asynchrones, équilibrage de charge, grappes distantes, bibliothèque de communication multithreadée, systèmes linéaires, EDP

Abstract

In this document, we present a synthesis of our works about AIAC algorithms (Asynchronous Iterations with Asynchronous Communications) and load balancing for dynamic networks.
With the development of large-scale networks for which topology may be dynamic, it seems important to study new load balancing algorithms. Based on decentralized algorithms for static topologies, such as the diffusion exchange and GDE, we have adapted them for dynamic communication links.
Besides, the use of distant clusters entails some new constraints relative to the context of local homogeneous clusters : the heterogeneity of machines and networks, fluctuating and possibly high latencies, and bandwidth that may be disturbed. In this context, the number of synchronisations should be reduced and coarse-grained algorithms are preferred. AIAC algorithms are suited for those constraints. In order to implement them efficiently, we advise users to start from a synchronous version and modify the communication management and the convergence detection. Multithreading environments allow the separation of communications and computation and therefore to overlap communications, which may be disturbed, by computation.
Moreover, we report experimentations in heterogenous distant grids that allow the comparison of synchronous and asynchronous version behaviors. We have studied linear systems and partial differential equation systems (PDE) with multisplitting methods.
Finally, we have studied the coupling of load balancing and AIAC algorithms.

Key words : asynchronous iterative algorithms, load balancing, distant clusters, multithreading communication library, linear systems, PDE