Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle

Pierre-Eric Bernard

▶ To cite this version:

Pierre-Eric Bernard. Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle. Autre [cs.OH]. Institut National Polytechnique de Grenoble - INPG, 1997. Français. NNT : . tel-00004919

HAL Id: tel-00004919

https://theses.hal.science/tel-00004919

Submitted on 20 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée par

Pierre-Eric BERNARD

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Mathématiques Appliquées**

Parallélisation et multiprogrammation pour une application irrégulière de dynamique moléculaire opérationnelle

---

Date de soutenance : mercredi 29 octobre 1997

Composition du jury :

Président :	Brigitte	PLATEAU
Rapporteurs :	Olivier	COULAUD
	Cécile	GERMAIN
Examinateurs :	Luc	BOUGÉ
	Yves	CHAPRON
	Denis	TRYSTRAM (directeur de thèse)

Thèse préparée au sein du
Laboratoire de Modélisation et Calcul
de Institut d'Informatique et de Mathématiques Appliquées de Grenoble
et en coopération avec le
Laboratoire de Biophysique Moléculaire et Cellulaire du C.E.A.-Grenoble

À mes parents
François et Marie-José

Remerciements

Je tiens d'abord à remercier les membres du jury :

Mme. Brigitte PLATEAU pour l'honneur qu'elle m'a fait en acceptant de présider mon jury de thèse,
Mme. Cécile GERMAIN et M. Olivier COULAUD pour avoir trouvé le temps de rapporter mon travail,
M. Luc BOUGÉ pour sa participation au jury,
Yves CHAPRON pour sa participation au jury et pour m'avoir initié à la dynamique moléculaire lors de mon stage de DEA,
enfin M. Denis TRYSTRAM pour avoir été mon directeur de thèse pendant ces trois années.

Je tiens également à remercier Gilles VILLARD pour l'aide qu'il m'a apporté pour obtenir une bourse de thèse, Yves CHAPRON et Serge CROUZY pour l'aide qu'il m'ont apporté dans la partie biologique de ce travail.

Je voudrais également remercier les personnes du LMC et du BMC pour leur accueil dans leurs laboratoires respectifs. Un merci particulier à Jean-Marc pour ses conseils éclairés et à Jean-Louis pour les discussions enflammées. Merci également à Helmut G., Helmut H. et Markus pour leur accueil en Allemagne.

J'exprime également ma gratitude à mes voisins de bureau de première année, Christophe et Yannick pour leurs conseils “d'anciens” ; à mes voisins de bureau de deuxième années, Fred et Éric pour les discussions passionnantes sur une grande variété de sujets ; à ma voisine de troisième années, Andréa pour son calme malgré mon agitation à l'approche de la fin ; à Titou qui rentre d'Helvétie. Merci également à Alain, Alfredo, Alexandre, Benhur, Christophe, Denis N., Ekbel, François G., François O., Gerson, Gregory, Gustavo, Ilan, Jacques B., Jacques C., Jean-Guillaume, Jean-Luc, Joëlle, Khadija, Marcelo, Mathias D., Mathias G., Michel C., Michel R., Nicolas, Patrick, Paulo, Philippe A., Philippe W., Roberta, Thierry et honte à moi si j'ai oublié quelqu'un !

Finalement je tiens à remercier mes parents et ma soeur jumelle Sophie pour m'avoir toujours encouragé à poursuivre mes études, et grâce à qui j'en suis arrivé là. Un merci particulier à mon père pour sa contribution à la correction de ce document par sa relecture minutieuse.

Table des matières

I Présentation **5**

1 Le parallélisme **7**
1.1 Introduction 7
1.2 Évolution des ordinateurs 8
1.2.1 Les microprocesseurs 8
1.2.2 Architectures des ordinateurs parallèles 9
1.2.3 Les réseaux 11
1.3 Méthodologie de parallélisation 12
1.3.1 Sources de parallélisme 12
1.3.2 Choix de la granularité 13
1.3.3 Caractérisation des algorithmes parallèles 14
1.3.4 Ordonnancement 16
1.3.5 Placement 16
1.4 Évaluation pratique des programmes parallèles 18
1.4.1 Accélération et efficacité 19
1.5 Conclusion 19

2 Expression du parallélisme par processus communicants **21**
2.1 Programmation par échange de messages 21
2.1.1 Processus communicants 21
2.1.2 Communication point-à-point 22
2.1.3 Communication collective 23
2.1.4 La bibliothèque de passage de messages PVM 24
2.1.5 La spécification MPI 26
2.2 Processus léger et échange de messages 27
2.2.1 Parallélisme et multiprogrammation 27
2.2.2 Description de la multiprogrammation 28
2.2.3 La librairie ATHAPASCAN-0A 31
2.2.4 La librairie ATHAPASCAN-0B 35
2.3 Programmation synchrone, asynchrone avec ou sans processus légers : Un exemple simple 36
2.4 Conclusion 38

II Modèles et Approximations 41

3 Modèle classique de dynamique moléculaire 43
3.1 Introduction 43
3.2 Principe 44
3.3 Forces d'interactions non-liées 45
3.3.1 Énergie de VAN DER WAALS 45
3.3.2 Énergie de COULOMB 46
3.4 Forces d'interactions géométriques 47
3.4.1 Énergie des liaisons 47
3.4.2 Énergie des angles 48
3.4.3 Énergie des angles dièdres 48
3.4.4 Listes d'exclusion 50
3.5 Potentiel des forces de contraintes (*restraint*) 50
3.5.1 Potentiel des contraintes harmoniques 50
3.5.2 Contraintes aux bords du système 51
3.6 Équations du mouvement 51
3.6.1 Positions et vitesses initiales 51
3.6.2 Calcul de la température 52
3.6.3 Dynamique de NEWTON 52
3.6.4 Dynamique de LANGEVIN 53
3.7 Autres modèles de dynamique moléculaire 53

4 Approximations pour la dynamique moléculaire 55
4.1 Introduction 55
4.2 Méthode du rayon de coupure 57
4.2.1 Approximation des forces non-liées 58
4.2.2 Construction de la liste des voisins 59
4.3 Conclusion 61
4.4 Description des quatres structures test 61

III Parallélisation 65

5 Parallélisation et répartition initiale des calculs 67
5.1 Méthodes usuelles de parallélisation 67
5.1.1 Parallélisation par distribution des atomes 69
5.1.2 Parallélisation par distribution des forces 71
5.1.3 Parallélisation par partition de l'espace de simulation 72
5.2 Construction du graphe de tâches 77
5.2.1 Identification des tâches 77
5.2.2 Contraintes de précédence 78
5.2.3 Contraintes de placement 79
5.2.4 Estimation des coûts 81

5.3 Stratégies de placement . . . 82
5.3.1 Placement aléatoire . . . 82
5.3.2 Placement quantitatif . . . 82
5.3.3 Placement par bi-partition récursive du graphe de tâches . . . 83
5.3.4 Placement de la structure . . . 89
5.3.5 Schémas de communication . . . 90
5.3.6 Critères pour l'évaluation des placements . . . 91
5.4 Comparaison des placements . . . 94
5.4.1 Propriétés des placements . . . 94
5.4.2 Résultats . . . 95
5.4.3 Comparaison plus fine . . . 96
5.4.4 Comparaison des placements par bi-partition . . . 99
5.4.5 Résultats pour les autres systèmes . . . 100
5.5 Influence de la granularité et du rayon de coupure . . . 101
5.5.1 Modification de la granularité . . . 101
5.5.2 Résultats . . . 103
5.6 Conclusion . . . 106

6 Ajustement dynamique 107
6.1 Ordonnancement entrelacé . . . 107
6.1.1 Recouvrement des communications . . . 108
6.1.2 Expression en ATHAPASCAN-0A . . . 110
6.1.3 Expression en ATHAPASCAN-0B . . . 114
6.2 Comparaison entre les bibliothèques de communications . . . 116
6.3 Ajustement dynamique . . . 118
6.3.1 Principe . . . 118
6.3.2 Résultats . . . 124
6.4 Déplacement des atomes et ajustement dynamique . . . 127
6.4.1 Principe . . . 128
6.4.2 Résultats . . . 129
6.5 Extensibilité, mesures sur le CRAY-T3E . . . 131
6.6 Conclusion . . . 132

IV Conclusion 135

7 Conclusions et perspectives 137

Table des figures

1.1 (a) Ordinateur à mémoire partagée. (b) Ordinateur à mémoire distribuée. 10
1.2 (a) Ordinateur à mémoire distribuée avec un processeur de communication. (b) Ordinateur à mémoire distribuée dont les noeuds sont des machines à mémoire partagée. 10

2.1 Organisation des processus dans une machine PVM, et chemin des communications entre deux processus utilisateurs. 25
2.2 Exemple de graphe de tâches série/parallèle, et décomposition en flots d'exécution correspondante pour ATHAPASCAN-0A. 32
2.3 Squelette d'un modèle de service en ATHAPASCAN-0A. 33
2.4 Squelette d'un appel de service en ATHAPASCAN-0A. 33
2.5 Décomposition en deux sous-domaines et graphe de tâches correspondant. 37
2.6 Diagramme espace-temps de chacune des façons de programmer. 39

3.1 Schéma de l'itération fondamentale de dynamique moléculaire. 45
3.2 Forme de l'énergie de Van der Waals entre deux atomes de type I et J. . 47
3.3 (a) Liaison covalente. (b) Angle entre deux liaisons covalentes. 48
3.4 Angle dièdre propre. 49
3.5 Angle dièdre impropre. 50

4.1 (a) Calcul des forces suivant la distance. (b) Découpage de l'espace de simulation pour l'algorithme des moments multipolaires. 57
4.2 (a) Utilité de la distance de tolérance. (b) Construction de la liste des voisins. Les voisins d'un atome se trouvent dans sa boîte et dans les boîtes voisines. 60
4.3 Structure de 35 349 atomes d'un canal ionique dans une membrane de phospholipides POPE. 62

5.1 Découpage en lignes de la matrice des forces pour les algorithmes qui distribuent les atomes. 71
5.2 Découpage en blocs de la matrice des forces pour les algorithmes qui distribuent les forces. 71
5.3 Découpage spatial de l'espace de simulation. 73

5.4 Nombre d'atomes par boîte (ou processeur) pour un système réel de 3 625 atomes et temps de calcul moyen d'une itération sur les processeurs correspondants. 75
5.5 Dépendances de la tâche $T_{\text{g}}(5)$ de calcul des forces géométriques des atomes de la boîte 5. 79
5.6 Dépendances des tâches de calcul des forces non-liées entre les atomes des boîtes voisines de la boîte 2 : $T_{\text{nv}}(2,1)$, $T_{\text{nv}}(2,3)$, $T_{\text{nv}}(2,4)$, $T_{\text{nv}}(2,5)$, et $T_{\text{nv}}(2,6)$. 80
5.7 Phases de calculs et de communications d'une itération de dynamique moléculaire. 80
5.8 Principe en deux dimensions d'un découpage récursif grossier de l'ensemble des boîtes pour quatre processeurs. 84
5.9 Principe en deux dimensions d'un découpage récursif fin de l'ensemble des boîtes pour quatre processeurs. 85
5.10 Partition simple de l'ensemble des boîtes, et des ensembles correspondants. 87
5.11 Les trois étapes d'une partition fine de l'ensemble des boîtes, et des ensembles correspondants. 90
5.12 Comparaison du temps d'exécution moyen d'une itération de dynamique moléculaire pour le système de 11 615 atomes en utilisant différents placements initiaux. 96
5.13 Détail de l'exécution d'une itération de dynamique moléculaire sur 16 processeurs, pour un système de 11 615 atomes. 99
5.14 Comparaison des placements par bi-partition récursive. 100
5.15 Comparaisons des temps d'exécution pour trois systèmes différents, en utilisant les différentes stratégies de placement. 101
5.16 Ensemble des $((2k+1)^3-1)$ boîtes entourant la boîte i et potentiellement voisines de i au sens du rayon de coupure. Vue en deux dimensions et pour $k=3$. 102
5.17 Mesure du temps d'exécution pour différents rayons de coupure et différentes granularités, pour le système de 11 615 atomes. 104

6.1 Phases de calculs et de communications d'une itération de dynamique moléculaire. 108
6.2 Graphe de précédences d'une itération de dynamique moléculaire après regroupement des tâches en fonction du placement et de la dépendance des données. 110
6.3 Diagramme de GANTT sur 4 processeurs de 4 itérations, sous ATHAPASCAN-0A pour le placement BPR-FIN. 112
6.4 Diagramme de GANTT sur 4 processeurs de 3 itérations, sous ATHAPASCAN-0A pour un placement aléatoire. 113
6.5 Diagramme de GANTT sur 4 processeurs de 4 itérations, sous ATHAPASCAN-0B pour le placement BPR-FIN. 115

6.6 Comparaisons de l'évolution des temps d'exécution en statique et en dynamique, pour la structure de 35 349 atomes sur 8 processeurs en utilisant ATHAPASCAN-0B. Les graphiques du haut indiquent le retard ou l'avance relative de chacun des processeurs. Les graphiques du bas indiquent la moyenne, sur les 8 processeurs, des temps d'exécutions de l'itération courante. La zone en gris indique la part du temps d'attente. 119
6.7 Circulation des informations de charge entre le processeur maître et deux processeurs esclaves. 123
6.8 Comparaisons de l'évolution des temps d'exécution en statique et en dynamique, pour la structure de 35 349 atomes sur 16 processeurs en utilisant ATHAPASCAN-0B. Les graphiques du haut indiquent le retard ou l'avance relative de chacun des processeurs. Les graphiques du bas indiquent la moyenne, sur les 16 processeurs, des temps d'exécutions de l'itération courante, La zone en gris indique la part du temps d'attente. . . 126
6.9 Comparaisons de l'évolution des temps d'exécution en statique et en dynamique avec une redistribution des atomes tout les 40 pas, pour la structure de 35 349 atomes sur 8 processeurs, et en utilisant ATHAPASCAN-0B. Les graphiques du haut indiquent le retard ou l'avance relative de chacun des processeurs. Les graphiques du bas indiquent la moyenne, sur les 8 processeurs, des temps d'exécution de l'itération courante. La zone en gris indique la part du temps d'attente. 129

Liste des tableaux

5.1 Critères de comparaison des placements. 94
5.2 Détail des temps d'exécution d'une itération de dynamique moléculaire, pour un système de 11 615 atomes. 97
5.3 Moyenne des temps d'exécution totale, des temps d'attente et des efficacités pour les 100 premières itérations de dynamique de la structure de 11 615 atomes, avec un rayon de coupure de 20 Å. Et évaluation du déséquilibre de charge pour deux granularités différentes. 105
5.4 Moyenne des temps d'exécution totale, pour les 100 premières itérations de dynamique de la structure de 3 625 atomes, avec un rayon de coupure de 15 Å. 105

6.1 Moyenne sur les premières itérations des temps d'exécution totale d'une itération pour la structure de 11 615 atomes. Comparaisons des librairies de communications, pour un placement initial de type BPR-FIN. 117
6.2 Critère de comparaison de la qualité des placements pour la structure de 11 615 atomes, en fonctions du nombre de processeurs. 121
6.3 Détail des temps d'exécution d'une itération de dynamique moléculaire, pour un système de 35 349 atomes en statique et en dynamique, et en utilisant ATHAPASCAN-0B. 125
6.4 Temps d'exécution (t_{tot}), accélération (Acc) et efficacité (Eff) du calcul d'une itération de dynamique moléculaire, pour un système de 11 615 atomes, un système de 35 349 atomes et un système de 413 039 atomes. Les chiffres entre parenthèses indiquent les temps d'exécution sans ajustement dynamique de la charge de calcul. 131

Introduction

L'utilisation des machines parallèles pour les gros consommateurs de puissance de calcul que sont les mécaniciens, les physiciens, les biologistes, etc., est essentielle pour le développement de leurs disciplines. En particulier la dynamique moléculaire classique qui consiste à simuler le mouvement des atomes d'un système, est employée en biologie pour étudier les conformations des macromolécules, et pour la compréhension des mécanismes réactionnels des protéines dans les structures biologiques.

Actuellement, dans ce domaine du calcul numérique, de nombreuses méthodes parallèles ont été développées, mais elles utilisent relativement peu les mécanismes de régulation et perdent en efficacité sur des problèmes non structurés. En particulier, la méthode d'approximation des forces par un rayon de coupure, très employée pour l'étude dynamique des structures biologiques, est un problème irrégulier peu structuré d'un point de vue du parallélisme.

Pour tenter d'offrir des réponses à ces problèmes, ce travail de thèse s'inscrit dans une collaboration entre le Laboratoire de Modélisation et Calcul de l'Institut de Mathématiques Appliquées de Grenoble (LMC-IMAG), et le Laboratoire de Biophysique Moléculaire et Cellulaire du Département de Biologie Moléculaire et Structurale du CEA-Grenoble (BMC/DBMS/DSV/CEA).

Au début de ce travail, la simulation du mouvement des atomes de systèmes comportant environ 30 000 atomes pendant 10 000 itérations, nécessitait une semaine de calcul sur une station de travail séquentielle de l'époque avec les programmes traditionnels. Le calcul parallèle offrait alors la perspective de pouvoir traiter de tels systèmes en quelques heures. Il permettait également d'espérer pouvoir aborder des problèmes plus grands, et en augmentant le nombre de processeurs, de conserver des temps de traitements raisonnables de quelques heures.

Beaucoup de travaux existent sur la parallélisation d'algorithmes de dynamique moléculaire. Mais soit ils utilisent des modèles trop simplifiés pour la simulation du mouvement de protéines et ne sont pas utilisables; car ils ne tiennent pas compte des problèmes d'équilibre de charge et de localités des données spécifiques à l'étude des protéines. Soit ils utilisent des modèles complets mais ils ne sont pas extensibles et limités dans la taille des systèmes qu'ils peuvent simuler.

Le premier objectif de ce travail est de réaliser un programme parallèle de dynamique moléculaire qui soit opérationnel pour l'étude des protéines. C'est-à-dire comportant les

fonctionnalités suffisantes pour permettre la simulation du mouvement d'assemblages de molécules et de protéines. Le programme doit répondre aux exigences d'efficacité et de portabilité. Il doit aussi être extensible pour autoriser le traitement des assemblages de grande taille.

D'autre part, l'équipe de calculs parallèles du LMC, étudie et développe, dans le cadre du projet APACHE[1], un environnement portable pour la programmation des applications parallèles irrégulières. Il est constitué de ATHAPASCAN-0, une première couche logicielle qui combine un noyau de processus léger pour la multiprogrammation des processeurs et une bibliothèque de primitives de communications. ATHAPASCAN-0 sert de support d'exécution pour la programmation explicite d'applications parallèles multiprogrammées. La deuxième couche logicielle, ATHAPASCAN-1, est en cours de réalisation. Elle a pour objectif principal de simplifier la programmation des applications irrégulières. Pour cela, elle offre une plus grande abstraction de la machine cible en séparant la description du parallélisme du mécanisme d'exécution. Le projet APACHE comprend également le développement d'outils d'analyse des performances et de recherche de bogue. Le calcul parallèle offrait alors la perspective de pouvoir traiter de tels systèmes en quelques heures. Il permettait également d'espérer pouvoir aborder des problèmes plus grands, et en augmentant le nombre de processeurs, de conserver des temps de traitements raisonnables de quelques heures.

Ici aussi, beaucoup d'équipes de recherche s'orientent vers l'utilisation de la multiprogrammation pour faciliter la mise en oeuvre des applications irrégulières. Mais il existe encore aujourd'hui, peu d'applications qui tirent parti de la multiprogrammation légère.

Ainsi, le second objectif de ce travail est de montrer, à l'aide d'une application en vraie grandeur, comment la multiprogrammation peut être employée pour faciliter la mise en oeuvre efficace et portable des algorithmes parallèles irréguliers, et de valider ainsi la plate forme d'exécution ATHAPASCAN-0.

Organisation du document

Le document est organisé en quatre parties. La première partie présente le parallélisme et la multiprogrammation. Elle est divisée en deux chapitres. Le chapitre 1 introduit les concepts de base du parallélisme. Basé sur notre expérience, il propose ensuite une méthodologie pour aborder la parallélisation d'une application en fonction des caractéristiques du problème. Enfin, il donne des références vers les outils théoriques permettant de caractériser les algorithmes parallèles.

Le chapitre 2 introduit le paradigme de programmation par processus communicants. Ensuite nous définirons la multiprogrammation en insistant sur ces avantages pour la programmation des applications parallèles irrégulières. Nous décrirons également les deux versions de la plate-forme d'exécution ATHAPASCAN-0.

[1] Le projet APACHE est financé par le CNRS, l'INRIA, l'INPG et l'UJF.

La deuxième partie de ce document présente le modèle et les techniques d'approximations utilisés pour la dynamique moléculaire. Une partie importante de notre travail a consisté à isoler les éléments du modèle indispensable à la simulation du mouvement des protéines. Le chapitre 3 présente ses éléments que nous utiliserons par la suite dans notre programme. Le chapitre 4 donne les principales méthodes d'approximation des forces non-liées. Nous insisterons sur la méthode du rayon de coupure que nous avons utilisée par la suite.

La troisième partie constitue la plus grosse part de notre travail, elle présente la parallélisation de l'algorithme du rayon de coupure en utilisant la multiprogrammation. Plus précisément, le chapitre 5, après un rappel des principales méthodes de parallélisation utilisées en dynamique moléculaire, présente plus précisément la méthode de partition de l'espace de simulation, théoriquement la plus prometteuse. Nous montrerons ensuite la nécessité d'obtenir un bon équilibre de charge. Enfin, après une analyse plus précise du problème, nous proposerons une étude comparative des différentes stratégies de placement et de découpage des calculs et des données sur les processeurs au début de la simulation.

Dans le chapitre 6, nous montrerons comment nous avons tiré avantage de la multiprogrammation dans notre application. Nous étudierons ensuite un mécanisme d'ajustement dynamique de la charge. En particulier, nous verrons comment il arrive à corriger les imperfections du placement statique initial en introduisant un coût supplémentaire de calculs et de communications minimum.

Dans la dernière partie, nous conclurons en résumant les points importants de ce travail, puis nous proposerons des développements pour les travaux futurs.

Première partie

Présentation

Chapitre 1

Le parallélisme

Dans ce chapitre, nous introduirons les concepts de base du parallélisme. En particulier, nous discuterons de l'évolution des super-ordinateurs. Puis nous ferons un tour d'horizon des méthodologies à adopter pour construire et implanter des algorithmes parallèles pour ces super-ordinateurs. Le but de ce chapitre n'est pas de faire une description complète de toutes ces méthodes de parallélisation, mais de donner au lecteur non-spécialiste des références vers les outils théoriques permettant de caractériser les algorithmes parallèles. En fonction de ces caractéristiques, nous indiquerons les stratégies qui nous semblent les plus prometteuses pour aborder la parallélisation d'une application. Nous insisterons sur la méthodologie que nous avons adoptée par la suite pour notre application. Nous terminerons en donnant un moyen pratique pour évaluer les différents algorithmes parallèles.

1.1 Introduction

Les deux principes à l'origine du calcul parallèle sont issus de deux motivations simples. La nécessité de travailler vite donne le premier principe : un travail avance plus vite à plusieurs que seul. La nécessité de réaliser des gros travaux donne le deuxième principe : il est possible de réaliser à plusieurs des travaux dont la taille dépasse les capacités d'un seul. Ces deux principes dépassent largement le cadre du calcul parallèle. En effet, bien que plus complexes, nombre d'organisations des sociétés humaines illustrent ces principes. Plus précisément dans le domaine scientifique, le calcul parallèle ou parallélisme est l'ensemble des méthodes qui permettent d'organiser une série de calculs sur un ensemble d'unités de calcul concurrentes.

La simulation numérique est un des grands domaines d'application du calcul parallèle. Nouvel outil d'investigation scientifique, la simulation numérique consiste à réaliser des expériences virtuelles sur ordinateurs à partir d'un modèle mathématique de la réalité physique. Il est ainsi possible de simuler des expériences impossibles à réaliser par des moyens classiques. Que se soit pour des raisons économiques, politiques, éthiques ou techniques, la simulation numérique est quelquefois la seule approche réaliste. Les

domaines d'application sont nombreux [104] : simulation de mouvement d'étoiles en astrophysique, conception d'avions en aéronautique, simulation d'accident nucléaire, prévision météorologique [124], étude des protéines en biologie... Toutes ces applications nécessitent des puissances de calcul de tout premier ordre. Mais là n'est pas leur seul besoin. Ces applications touchent plusieurs disciplines scientifiques. En amont du calcul parallèle, il faut construire un modèle mathématique du problème physique. Puis il faut trouver des techniques d'approximation et de résolution des équations du modèle mathématique qui soient les plus efficaces et les plus précises possibles. Le calcul parallèle consiste ensuite à exprimer la résolution des équations sous la forme d'algorithmes parallèles. Enfin il faut implanter ces algorithmes sur un ordinateur. En aval de l'algorithmique parallèle se trouve le matériel, la conception des ordinateurs parallèles, le développement des environnements de programmation et d'expression du parallélisme.

1.2 Évolution des ordinateurs

1.2.1 Les microprocesseurs

Depuis l'invention du circuit intégré en 1964, l'évolution des microprocesseurs est très rapide. La loi empirique de MOORE prédit un doublement de la densité de transistors sur une puce tous les 18 mois. Malgré les barrières technologiques successives, depuis 1968, cette loi a toujours été vérifiée. Grâce aux progrès conjoints dans la miniaturisation et la conception des circuits, la puissance des microprocesseurs évolue à un rythme souvent supérieur aux prévisions les plus optimistes [75].

Aujourd'hui, malgré de nouvelles barrières technologiques il semble difficile de parier sur un arrêt brutal de cette évolution. Cependant la barrière ne semble plus seulement technologique mais également économique. L'investissement nécessaire à la construction des usines de microprocesseurs double tous les trois ans. Rentabiliser ces usines impose la fabrication de grandes séries de microprocesseurs. Actuellement, ces usines sont rentabilisées rapidement grâce au développement croissant des applications de l'informatique dans nos sociétés. De plus, l'évolution rapide des performances des ordinateurs rend ceux ci rapidement obsolètes, ce qui pousse les utilisateurs à renouveler souvent leur matériel. Mais l'évolution rapide des microprocesseurs a aussi tendance à pénaliser la programmation et le développement de nouvelles architectures d'ordinateurs.

Les premiers super-ordinateurs ont été construits autour des microprocesseurs vectoriels spécifiques. Cette famille de microprocesseurs "pipeline" les opérations élémentaires pour calculer rapidement les opérations d'algèbre linéaire courantes. En particulier les ordinateurs Cray ont largement exploité cette technique. Il existe aujourd'hui des compilateurs spécifiques qui permettent d'exploiter au mieux les performances des microprocesseurs vectoriels avec une grande facilité d'utilisation. Mais le prix des machines à base de microprocesseurs vectoriel les rend accessibles seulement aux gros centres de calcul. Bien que les microprocesseurs vectoriels soient toujours plus performants que les microprocesseurs scalaires classiques, la différence de performance a tendance à diminuer. Aujourd'hui beaucoup de constructeurs de super-ordinateurs utilisent des processeurs sca-

laires de grande série, moins coûteux.

1.2.2 Architectures des ordinateurs parallèles

Les ordinateurs parallèles sont construits autour d'un ensemble de microprocesseurs. Plusieurs points caractérisent ces machines. Historiquement, les machines parallèles ont été classifiées en fonction de la multiplicité des flots d'instructions et de données. Ainsi, FLYNN [56] distingue :

- les machines SIMD [1] où les processeurs exécutent la même opération simultanément sur des données différentes ;
- des ordinateurs MIMD [2] où les noeuds de calcul exécutent globalement le même travail mais pas forcément la même opération au même moment.

Plus simples à utiliser, les machines SIMD ne permettent cependant pas de traiter efficacement tout les types de problèmes. De ce point de vue, les machines MIMD sont d'un usage plus général. C'est principalement pour cette raison que la plupart des ordinateurs parallèles d'aujourd'hui sont des machines MIMD.

Le deuxième point qui différencie les architectures des ordinateurs parallèles, est la localisation de la mémoire dans la machine :

- **mémoire partagée** : il existe une mémoire contiguë unique accessible directement et uniformément par chacun des microprocesseurs (cf. figure 1.1 (a)) ;
- **mémoire distribuée** : chacun des microprocesseurs possède sa propre zone mémoire. Les différents noeuds de calcul (ensemble microprocesseur, mémoire) sont reliés entre eux par un réseau de communication (cf. figure 1.1 (b)). Les processeurs accèdent directement seulement à leur zone de mémoire. On inclut dans les machines à mémoire distribuée aussi les ordinateurs comme le CRAY-T3E qui ont un espace d'adressage unique et qui possèdent un mécanisme d'accès-mémoire distant. Ce mécanisme permet à un processeur d'accéder directement à la mémoire d'un autre processeur via le réseau. Mais un accès à la mémoire d'un autre processeur est plus lent qu'un accès à la mémoire locale.

L'existence d'une mémoire commune entre les processeurs simplifie le travail de parallélisation des algorithmes. Une parallélisation semi-automatique, assistée par l'utilisateur, permet, à partir d'un code séquentiel existant, d'obtenir des performances acceptables sur ces machines. En effet, les ordinateurs à mémoire partagée exploitent efficacement le parallélisme de grain fin (cf. 1.3.2). Le parallélisme à grain fin est plus facile à extraire et à contrôler que le parallélisme à grain plus gros nécessaire aux machines à mémoire distribuée. Cependant, dans les ordinateurs à mémoire partagée, le nombre de processeurs est limité. Le goulot d'étranglement que constitue l'accès à une mémoire commune limite le nombre de processeurs à une dizaine. De plus, la plupart des applications nécessitant un super-calculateur ont des besoins de mémoire vive très importants.

[1] SIMD = Single Instruction Multiple Data, instruction unique et données multiples.
[2] MIMD = Multiple Instruction Multiple Data, instructions multiples donnés multiples.

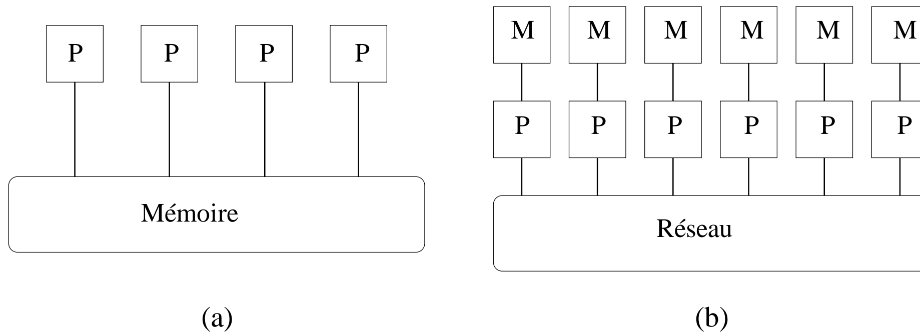


FIG. 1.1: (a) Ordinateur à mémoire partagée. (b) Ordinateur à mémoire distribuée.

Actuellement, les ordinateurs à mémoire distribuée sont les seuls ordinateurs qui permettent d'utiliser simultanément plusieurs dizaines à quelques centaines de processeurs. On parle alors de parallélisme massif. Les ordinateurs à mémoire distribuée sont, en raison de leur architecture, plus extensibles que les ordinateurs à mémoire partagée. C'est-à-dire qu'il est possible d'augmenter le nombre de noeuds de calcul.

Il existe aussi aujourd'hui des machines avec une architecture mixte. Il s'agit d'ordinateurs à mémoire distribuée dont les noeuds de calcul sont des ordinateurs parallèles à mémoire partagée. La machine SGI-ORIGIN 2 000 vendue comme une machine à mémoire partagé est en fait une machine mixte qui possède plusieurs noeuds de calcul reliés par un réseau. Les noeuds sont composés de deux processeurs qui partagent une mémoire locale et ils accèdent à la mémoire des autres noeuds par des requêtes de lectures/écritures à travers le réseau. Nous pensons que ce type d'architecture est amené à se développer, car elle permet d'exploiter plusieurs grains de parallélisme simultanément. La figure 1.2 (b) montre un exemple d'une telle architecture. Dans la suite, nous nous intéresserons essentiellement aux architectures à mémoire distribuée.

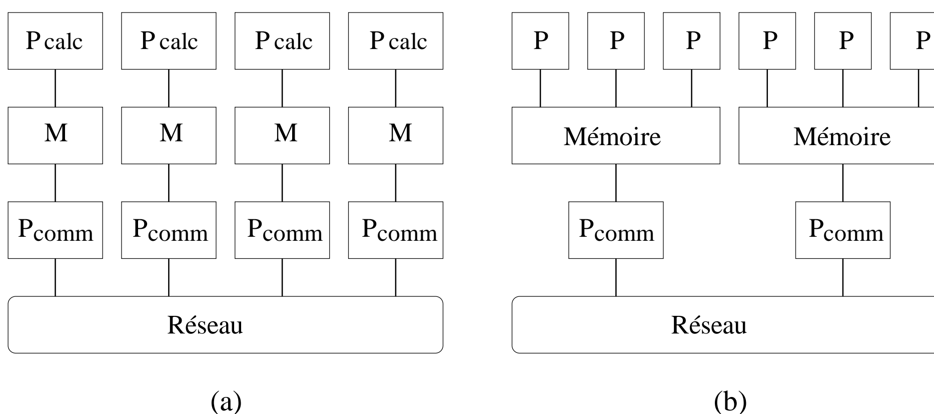


FIG. 1.2: (a) Ordinateur à mémoire distribuée avec un processeur de communication. (b) Ordinateur à mémoire distribuée dont les noeuds sont des machines à mémoire partagée.

1.2.3 Les réseaux

Dans un ordinateur parallèle à mémoire distribuée, les noeuds de calcul communiquent entre eux par des échanges de messages à travers le réseau de communication. Les caractéristiques et les performances du réseau de communication constituent souvent un élément déterminant dans l'évaluation globale d'une machine parallèle. Le réseau doit être capable d'envoyer rapidement des petits messages, pour faire circuler des informations de contrôle, synchroniser des noeuds de calcul ou réaliser des opérations collectives (cf. 2.1.3). La latence de communication, temps de démarrage de la communication, doit être la plus faible possible. Le réseau doit aussi être capable de redistribuer les données sur les processeurs. Dans ce cas, les volumes de données échangées entre les noeuds sont très importants. La bande passante du réseau, quantité de données qui peuvent transiter sur un lien par unité de temps, doit être la plus élevée possible. Les ordinateurs parallèles actuels possèdent souvent un réseau spécifique. En effet, si les réseaux locaux d'aujourd'hui qui relient les stations de travail ont des débits très importants, la latence de communication de ces réseaux reste importante.

Les constructeurs ont essayé plusieurs topologies de réseau pour connecter les noeuds de calcul : des hypercubes (CM2, iPSC), des grilles (Paragon), des tores (T3E), des *fat-tree* (CM5), des réseaux multi-étages (SP1), ... La tendance actuelle est aux réseaux de degré constant et facilement extensible : tores, grilles ou réseau multi-étages. Un autre élément important des réseaux est l'algorithme d'acheminement d'un message dans le réseau, c'est-à-dire le routage. La plupart des ordinateurs parallèles à mémoire distribuée de la dernière génération utilisent des modes de routage dérivés du *wormhole*. Dans ce mode, acheminer un message consiste à établir un chemin dans le réseau, entre le noeud source et la destination. Le message est ensuite découpé en petits paquets qui sont envoyés les uns à la suite des autres sur le chemin fixé. Ce mode de routage rend les communications peu dépendantes de la topologie du réseau. C'est-à-dire que la distance qui sépare les deux noeuds dans la topologie du réseau a une influence négligeable sur la communication [126].

Toujours pour améliorer les performances de communication, certaines machines (T3D, T3E, et SP2 en partie) offrent la possibilité de recouvrir les calculs et la communication. C'est-à-dire qu'il est possible de réaliser simultanément des calculs et des communications sur des données différentes. Ceci permet de masquer le sur-coût des communications [26] [40]. Les noeuds de calcul des machines qui permettent le recouvrement possèdent un microprocesseur spécialisé dans les communications qui accède directement à la mémoire du noeud. Il est possible d'assimiler ces noeuds de calcul à des ordinateurs à mémoire partagée bi-processeurs, dont un processeur est dédié aux calculs et l'autre est dédié aux communications (cf. figure 1.2 (a)).

1.3 Méthodologie de parallélisation

Il existe aujourd'hui, en particulier pour les ordinateurs à mémoire partagée, des outils de parallélisation semi-automatiques ou des compilateurs qui proposent de construire automatiquement un programme parallèle à partir d'un programme séquentiel. Cette approche est très pratique, mais actuellement, les généralisations pour les ordinateurs à mémoire distribuée ne sont pas à la hauteur des espérances. La parallélisation automatique ou semi-automatique à partir d'un programme séquentiel demande peu de travail en comparaison d'une parallélisation explicite, aussi il est toujours intéressant d'essayer cette approche, et d'envisager une parallélisation explicite en fonction des résultats et des besoins. Pour la plupart des problèmes, la parallélisation explicite est souvent la seule solution envisageable, au moins à moyen terme, pour obtenir de bonnes performances. Il est souvent nécessaire de construire un algorithme parallèle spécifique très différent de l'algorithme séquentiel habituel. La résolution en parallèle d'un programme nécessite une vision globale du problème, les outils automatiques n'ont qu'une vision locale liée à un algorithme séquentiel particulier.

La parallélisation explicite comporte plusieurs étapes : il faut trouver le parallélisme, puis découper le problème en tâches. Il faut ensuite définir un ordre d'exécution et associer à chacune des tâches un processeur chargé de leur exécution. La parallélisation explicite d'un problème est un travail d'expertise, il n'existe pas d'approche type. Les paragraphes qui suivent résument les questions importantes à se poser et indiquent un état de l'art des approches de parallélisation les plus prometteuses pour chaque nature de problème.

1.3.1 Sources de parallélisme

Parallélisme de données

La première source de parallélisme est le parallélisme de données. Il existe quand une même opération est appliquée à chacun des éléments d'un ensemble de données. Cette source de parallélisme est par exemple très fréquente dans les problèmes d'algèbre linéaire dense. Des langages tels que HPF (High Performance FORTRAN) [3] [103] permettent aisément d'exprimer et d'exploiter cette forme de parallélisme. Dans ce mode d'expression du parallélisme, le travail des processeurs est guidé par la distribution des données. Les communications sont définies comme des phases de redistribution des données sur les processeurs. Le programme parallèle est une succession de phases de calculs et de phases de communications pour la redistribution des données. Cependant, le parallélisme de données ne permet pas de décrire efficacement toutes les formes de parallélisme. Il est souvent difficile de connaître *a priori* si le parallélisme de données est adapté à un problème. Par exemple, au cours d'une étude préliminaire à ce travail [6], nous avons montré que le parallélisme de données était adapté au calcul complet des forces non-liées, mais pas à l'algorithme du rayon de coupure (cf. chapitre 4).

[3] FORTRAN Haute performance.

Parallélisme de contrôle

Plus général que le parallélisme de données, le parallélisme de contrôle consiste à décrire un algorithme parallèle sous la forme d'un graphe orienté sans cycle. Les noeuds du graphe sont des suites d'opérations élémentaires exécutées en séquentiel, ce sont les tâches du graphe. Les arcs du graphe indiquent les contraintes de précédence entre les tâches. C'est-à-dire l'utilisation par une tâche d'une donnée calculée par une tâche précédente. Ce graphe, appelé graphe de précédence ou graphe de tâches, définit un ordre partiel sur les tâches. Les tâches non ordonnées par cet ordre partiel peuvent être exécutées en parallèle. Le travail des processeurs est ici guidé par les dépendances entre les tâches.

Le parallélisme de contrôle est plus général que le parallélisme de données. Il est ainsi possible de traiter le parallélisme de données comme un cas particulier de parallélisme de contrôle. Dans la suite de ce document, nous parlerons essentiellement de parallélisme de contrôle.

1.3.2 Choix de la granularité

Lors de la description d'un algorithme parallèle sous la forme d'un graphe de tâches, choisir la granularité consiste à définir plus précisément ce qu'est une tâche. Empiriquement, on peut définir la granularité comme le rapport entre le temps moyen d'exécution des tâches divisé par la largeur du graphe de tâches. La largeur du graphe de tâches est le nombre maximum de tâches potentiellement exécutables en parallèle. Elle est fortement liée au volume des communications, c'est pourquoi la granularité est également définie comme le rapport entre le temps de calcul divisé par le temps de communication. Dans la suite, pour rester indépendant de la machine-cible, nous garderons la première définition.

Ainsi pour diminuer la granularité, il faut découper chaque tâche en sous-tâches plus petites. On parle de grain plus fin. En pratique, diminuer la granularité augmente le parallélisme potentiel et permet d'utiliser plus de processeurs. Inversement, augmenter la granularité consiste à regrouper des tâches. On parle alors de grain plus gros. Augmenter la granularité permet de réduire le temps nécessaire à la gestion du parallélisme.

Le choix de la granularité dépend fortement de la machine-cible et de l'application. Il existe même une opposition importante entre le type des machines et la granularité qu'elles peuvent traiter efficacement. En effet, les ordinateurs parallèles à mémoire partagée peuvent gérer un parallélisme de grain fin avec un sur-coût faible. Mais le nombre de processeurs de ces machines est limité. Ainsi, le parallélisme potentiel important, induit par le grain fin, n'est pas exploité. Inversement, les ordinateurs à mémoire distribuée permettent d'exploiter le maximum de parallélisme. Mais un grain trop fin entraîne souvent trop de communications entre les processeurs. Et un grain trop gros entraîne des déséquilibres de charge entre les noeuds de calcul. Le choix de la granularité est un compromis entre le maximum de parallélisme potentiel, un sur-coût de gestion minimum, et un bon équilibre de charge entre les noeuds de calcul. C'est une étape primordiale, il est souvent utile de pouvoir modifier facilement le grain d'une application pour pouvoir affiner ce choix.

1.3.3 Caractérisation des algorithmes parallèles

Avant tout travail de parallélisation, il est important de connaître les caractéristiques des algorithmes. La stratégie à employer par la suite dépend fortement de ces caractéristiques.

Régularité / Irrégularité

La distinction entre un problème régulier et un problème irrégulier n'est pas tranchée. En règle générale, un problème est dit régulier si le graphe de tâche et la durée des tâches de ce graphe sont connus et s'ils ne changent pas d'une exécution à l'autre. Inversement, si le graphe de précédence n'est pas complètement déterminé le problème est irrégulier. Plus précisément, ce que l'on connaît du graphe de précédence et le moment où on le connaît définissent la nature régulière ou irrégulière d'un problème. Voici quelques-unes des caractéristiques des graphes de précédence :

- **statique ou dynamique** : le graphe est statique s'il est connu a priori. Inversement, il est dynamique s'il est connu seulement à l'exécution et s'il dépend des données ;
- **structuré ou non** : un graphe structuré possède une topologie régulière. Un graphe dynamique structuré peut être prévu au début de l'exécution, alors qu'un graphe non structuré est imprévisible ;
- **uniforme ou non** : un graphe est uniforme, si toutes les tâches du graphe ont le même temps d'exécution.

Ce que l'on connaît des tâches détermine aussi la nature du problème. Une tâche peut être :

- **statique** : le coût de calcul de la tâche est connu a priori avant l'exécution. Il est indépendant des données et il est constant d'une exécution à l'autre ;
- **dynamique prévisible** : le coût de la tâche dépend des données. Mais il existe une fonction simple qui, à partir des entrées de la tâche, permet d'estimer le coût de calcul de la tâche. Une fonction simple signifie ici que le coût de calcul de cette fonction est négligeable devant le coût de calcul de la tâche ;
- **dynamique inconnu** : le coût d'une tâche est considéré comme inconnu ou imprévisible, s'il n'existe pas de fonction simple permettant d'estimer le coût de calcul de la tâche en fonction de ces entrées. Typiquement, le coût des tâches est inconnu si estimer ce coût revient à exécuter la tâche.

Modèles de machine et complexité

Comme en séquentiel, pour comparer quantitativement les algorithmes parallèles, il faut déterminer leurs complexités parallèles. Le modèle PRAM (*Parallel Random Access Memory*) [4] est un modèle abstrait de machine parallèle très populaire pour évaluer et comparer les algorithmes parallèles [57]. Une PRAM est une machine synchrone composée

[4] Accès aléatoires et concurrents à la mémoire.

d'une infinité de processeurs reliés entre eux par une mémoire globale infinie. À chaque unité de temps, les processeurs lisent les données dans la mémoire globale, exécutent une opération, et écrivent les résultats dans la mémoire globale. Le modèle spécifie également comment les conflits pour les accès concurrents aux données sont résolus. Ainsi, le modèle le moins puissant EREW-PRAM (*Exclusive Read Exclusive Write*) autorise une seule lecture et une seule écriture à la fois dans une case mémoire. Le modèle le plus couramment utilisé, CREW-PRAM (*Concurrent Read Exclusive Write*) n'autorise qu'une écriture à la fois, mais il autorise les lectures multiples. Enfin le modèle le plus puissant CRCW-PRAM (*Concurrent Read Concurrent Write*) autorise les lectures et les écritures concurrentes. Pour ce dernier type de PRAM plusieurs modes permettent de préciser la façon de traiter les écritures concurrentes : soit les processeurs qui écrivent dans une même case mémoire écrivent la même valeur (*Common*) ; soit on utilise un opérateur arithmétique comme une somme pour combiner les écritures (*Combining*); soit on choisit arbitrairement (*Arbitrary*) ou avec un système de priorité (*Priority*) le processeur qui écrit. Ce modèle de machine permet d'évaluer, en fonction de la taille du problème, le parallélisme potentiel et le temps d'exécution parallèle d'un algorithme. Mais il ne permet pas d'évaluer le coût des communications. Il existe d'autres modèles de machines dérivés du modèle PRAM qui prennent en compte les coûts des communications, le lecteur intéressé par un état de l'art de ces modèles, peut consulter la thèse de Frédéric GUINAND [69].

Plus qu'un modèle de machine, BSP (*Bulk Synchronous Parallelism*) est un modèle de programmation et d'estimation des performances [127] [118]. Les programmes BSP sont composés d'un nombre fixé de processeurs virtuels ayant une mémoire locale infinie. Les processeurs virtuels sont placés aléatoirement sur les processeurs physiques, sans tenir compte de la localité entre les processeurs virtuels. Les programmes BSP comportent une suite de super-étapes (*supersteps*), chacune divisées en trois phases :

– une phase de calcul dans laquelle les processeurs utilisent uniquement des données de leur mémoire locale ;
– une phase de communication, éventuellement recouverte par les calculs, pour l'échange des données nécessaires à la super-étape suivante ;
– une synchronisation globale des processeurs pour attendre la fin des calculs et des communications de la super-étape courante.

Pour estimer les performances d'un programme BSP, on estime le coût de chacune des super-étapes. C'est-à-dire, pour une super-étape, le maximum des coûts de calcul de chacun des processeurs virtuels, plus le maximum des temps des communications, et plus le temps de la synchronisation. Pour qu'un programme BSP soit performant, il faut un minimum de super-étapes pour éviter les synchronisations, et dans une super-étape, il faut équilibrer le calcul et les communications entre les processeurs virtuels. Des améliorations du modèle permettent de tenir compte du recouvrement des calculs par les communications, ou de limiter les synchronisations à un sous-ensemble de processeurs virtuels.

En marge de ces modèles de machines pour évaluer les algorithmes parallèles, il est courant d'utiliser une évaluation empirique des performances d'un algorithme. Pour cela,

on donne l'ordre de grandeur du nombre d'opérations à effectuer sur un processeur, en fonction du nombre de processeurs, et en fonction de la taille du problème. De même pour les communications, on estime le volume des données que doit envoyer ou recevoir un processeur, toujours en fonction de la taille du problème et du nombre de processeurs.

1.3.4 Ordonnancement

L'ordonnancement des tâches d'un graphe de précédence consiste à attribuer une date de début d'exécution aux tâches du graphe. Ces dates doivent être compatibles avec le nombre de processeurs choisi et avec l'ordre partiel défini par le graphe de précédence. La stratégie d'ordonnancement à employer dépend des caractéristiques du graphe de tâches. Si le problème est régulier, le graphe de précédence est complètement déterminé, on utilise des stratégies d'ordonnancement statique. L'ordonnancement statique d'un graphe de précédence quelconque est un problème NP-complet. Mais il existe de bonnes heuristiques d'ordonnancement statique [39] [69].

Dans la majorité des problèmes, le graphe de tâches dépend des données en entrée du programme. Le problème est irrégulier et il n'est pas possible d'utiliser un ordonnancement statique. Dans ce cas, il est possible d'utiliser une stratégie d'ordonnancement en ligne (*on-line scheduling*). Cette méthode ordonne les tâches au fur et à mesure de leur création. L'ordonnancement en ligne nécessite un mécanisme de migration de tâches qui permet d'arrêter l'exécution d'une tâche et de la reprendre sur un autre processeur. Des résultats théoriques récents montrent que l'on peut obtenir de bonnes performances avec des algorithmes gloutons simples d'ordonnancement en ligne [116].

Il existe également des solutions intermédiaires entre les ordonnancements statiques qui nécessitent la connaissance complète du graphe de tâches et les ordonnancements en ligne qui utilisent peu d'information sur le graphe de tâches. En particulier, pour le problème itératif qui nous intéresse, nous avons développé une stratégie d'ordonnancement dynamique associée à un placement des tâches au début du programme (cf. 2.3).

1.3.5 Placement

Les étapes d'ordonnancement et de placement sont fortement liées, et souvent en pratique, ces deux étapes n'en forment qu'une seule. Nous avons tenu à les séparer car le rôle et les objectifs de ces deux étapes sont bien différents. Le placement des tâches d'un graphe de précédence consiste à associer un site d'exécution à chacune des tâches du graphe. Dans les ordinateurs parallèles à mémoire distribuée, l'accès aux données n'est pas uniforme. En effet, si deux tâches sont placées sur le même noeud de calcul, la communication des données entre ces deux tâches se fait par la mémoire du noeud, et le temps de communication est nul. Au contraire, si les deux tâches sont placées sur des noeuds de calcul différents, il faut communiquer les données à travers le réseau. Le placement doit donc chercher à minimiser les communications en plaçant sur le même processeur des tâches ayant beaucoup de données à échanger. Le placement doit également chercher à équilibrer au mieux la charge de calcul entre les processeurs. Le placement est la

recherche d'un compromis entre l'équilibre de charge et la minimisation des communications.

Ici aussi, la stratégie de placement dépend de ce que l'on connaît. Si le problème est régulier, on utilise un placement statique [69]. À partir du résultat de l'ordonnancement, on construit un graphe non orienté dont les sommets sont valués par les coûts de calcul et dont les arêtes sont valuées par les coûts de communication des données. Le placement statique est une optimisation combinatoire dont la fonction à minimiser rend compte du déséquilibre de charge et des communications d'un placement.

Si le problème est irrégulier, on utilise des algorithmes de placement dynamique et des mécanismes de régulation dynamique de la charge de calcul sur les processeurs.

Regroupement des communications

Toujours pour minimiser les communications, cette étape consiste, une fois le placement défini, à regrouper les communications de données identiques. Plus précisément, il s'agit d'identifier les données communiquées entre chaque processeur, et à regrouper les communications de données identiques en une seule communication. Il peut également être intéressant d'identifier des communications collectives pour améliorer leur efficacité.

Répartition dynamique de charge

Dans un problème irrégulier, où le coût des tâches est inconnu ou imparfaitement estimé, il faut utiliser un mécanisme de régulation dynamique de la charge de calcul sur les processeurs. C'est-à-dire qu'il faut placer les tâches sur les processeurs en cours d'exécution. Les méthodes de répartition dynamique comportent quatre mécanismes essentiels [130] :

– **évaluation de la charge.** Chacun des noeuds de calcul détermine sa charge ;
– **évaluation du déséquilibre de charge.** Dans cette phase, il faut déterminer la surcharge ou la sous-charge de chacun des noeuds ; pour savoir, sur chacun des processeurs, quelle charge il faut exporter vers les autres processeurs, ou respectivement, quelle charge il faut importer des autres processeurs ;
– **migration de tâches.** Ce mécanisme permet effectivement d'effectuer le déplacement d'une tâche de calcul d'un noeud vers un autre ;
– **sélection des tâches à migrer.** Cette étape consiste à sélectionner les tâches dont la migration est la plus efficace pour l'équilibre de charge.

Les mécanismes de répartition dynamique de la charge ont tendance à augmenter le volume des communications. Aussi un bon algorithme de répartition dynamique doit tenir compte des communications, et doit également consommer un minimum de temps de calcul pour ne pas ralentir l'application. Il est souvent difficile de définir précisément ce qu'est la charge d'un processeur, cela dépend de la sémantique du programme. C'est pourquoi, dans beaucoup d'applications, les mécanismes de régulation sont intégrés à l'application et sont spécialement adaptés au problème.

Localité

La localité est la propriété de placer les tâches utilisant les mêmes données sur le même processeur. Un placement qui possède des bonnes propriétés de localité permet de réduire intrinsèquement les coûts de communications entre les processeurs. C'est-à-dire qu'il permet de réduire les coûts de communications sans les connaître précisément.

Le placement d'un graphe quelconque sur les processeurs est un problème d'optimisation combinatoire NP-complet. Pour diminuer l'aspect combinatoire du problème, il est possible de construire des heuristiques de placement spécifiques à un type de graphe en imposant des contraintes de localité. Il s'agit, en fonction de la structure du graphe de tâches de lier le placement de certaines tâches. Par exemple, on peut imposer le placement sur un même processeur de tâches ayant des fortes dépendances de données. Ou plus généralement, on peut contraindre le placement de certaines tâches à un sous-ensemble de processeurs fonction du placement d'autres tâches. Il convient cependant de vérifier que les contraintes ne nuisent pas à l'obtention d'un bon équilibre de charge. En particulier nous montrerons l'intérêt de cette approche pour notre application dans le chapitre 5.

1.4 Évaluation pratique des programmes parallèles

Pour comparer les algorithmes parallèles, et pour connaître leur efficacité par rapport aux algorithmes séquentiels, il faut faire des mesures de performances sur ces algorithmes. La plupart des ordinateurs parallèles à mémoire distribuée actuels possèdent sur chacun des noeuds un système UNIX qui permet de partager les noeuds de calcul entre plusieurs programmes. Mais pour obtenir des mesures correctes, il convient de réserver la machine à la seule application mesurée. Ceci afin de ne pas être perturbé par les autres utilisateurs.

Pour évaluer précisément les programmes parallèles, il est important de connaître les sources d'inefficacité. Pour cela nous distinguerons trois états possibles des noeuds de calcul. Et nous mesurerons les temps passés par chacun des noeuds dans ces états. Ainsi nous mesurerons :

- **le temps de calcul** ($t_{\textbf{calc}}$) : ce temps représente le temps de calcul directement utile pour l'application ;
- **le temps de communication** ($t_{\textbf{comm}}$) : le temps de communication est ici le temps de calcul du processeur utilisé pour réaliser et gérer les communications. Ce temps est proportionnel au volume des communications. Il doit être minimum dans un programme parallèle ;
- **le temps d'attente** ($t_{\textbf{att}}$) : il s'agit ici du temps perdu par le processeur sur l'attente d'un événement, attente d'une communication ou attente d'une synchronisation. Plus précisément c'est le temps d'inactivité du processeur. Il est caractéristique soit d'un déséquilibre de charge entre les noeuds de calcul de l'application, soit d'un mauvais entrelacement des calculs et des communications. Il doit également être le plus faible possible.

En pratique, il est souvent difficile de distinguer très précisément les états d'un noeud. Cela dépend beaucoup de la machine, et de la bibliothèque de passage de messages que

l'on utilise. Nous devrons nous contenter de mesures approchées des temps passés par chacun des noeuds dans les différents états.

1.4.1 Accélération et efficacité

L'accélération mesurée d'un algorithme sur p processeurs (A_p) est définie comme le rapport du temps d'exécution séquentiel de l'algorithme T_{seq} divisé par le temps d'exécution parallèle de l'algorithme sur p processeurs (T_p) :

$$A_p = \frac{T_{\mathrm{seq}}}{T_p}$$

Plus précisément, T_p est le maximum des temps d'exécutions mesuré sur chacun des processeurs.

L'efficacité d'un algorithme (E_p) est définie comme le rapport de l'accélération de l'algorithme sur p processeurs, divisée par le nombre de processeurs. Il est souvent exprimé par un pourcentage qui représente l'utilisation moyenne des processeurs par rapport à une parallélisation parfaite. Une parallélisation parfaite signifie que tous les processeurs sont en permanence utilisés pour effectuer des opérations utiles pour l'application.

$$E_p = \frac{A_p}{p} = \frac{T_{\mathrm{seq}}}{p \times T_p}$$

1.5 Conclusion

Les ordinateurs parallèles à mémoire distribuée sont actuellement les seules machines capables de simuler des problèmes de grande taille à des coûts raisonnables. Pour obtenir de bonnes performances, il faut explicitement construire les programmes parallèles. Le simple portage d'une application séquentielle est souvent impossible. Il existe aujourd'hui des techniques et des outils qui permettent de construire des applications portables, mais la programmation des ordinateurs parallèles à mémoire distribuée reste difficile. Construire des algorithmes parallèles extensibles requiert une bonne vision globale du problème. Il faut souvent faire plusieurs essais avant de trouver la meilleure granularité et le meilleur compromis entre un minimum de communications et un bon équilibre de charge entre les processeurs.

Chapitre 2

Expression du parallélisme par processus communicants

Dans ce chapitre, nous présenterons le très populaire paradigme de programmation par processus communicants. Ensuite nous introduirons le concept de multiprogrammation en insistant sur les difficultés de performances que cette approche peu résoudre.

2.1 Programmation par échange de messages

Un des aspects du calcul parallèle, indépendamment de la machine-cible, consiste à exploiter au mieux le parallélisme d'une application. Pour répondre à ce problème, plusieurs environnements de programmation parallèles s'inspirent du paradigme de programmation par processus communicants. Nous allons le décrire brièvement, puis nous donnerons les caractéristiques des deux environnements de programmation par processus communicants les plus populaires : PVM [60] et MPI [47].

2.1.1 Processus communicants

Dans le paradigme de programmation par processus communicants, un programme parallèle est un ensemble de processus qui concourent à la résolution d'un problème unique. Chacun des processus possède une zone mémoire privée et la coopération s'exprime par des communications explicites entre processus. Mais la notion de processus est distincte de la notion de processeur, il n'y a pas forcément un seul processus par processeur. Ainsi, tout les processus placés sur le même processeur partagent les ressources du noeud de calcul. En particulier, ils partagent le temps d'utilisation du processeur et l'espace mémoire. Mais, dans ce paradigme, il n'y a pas de partage de données entre les processus, même si elles sont dans la même mémoire physique. L'échange des informations n'est possible que par des échanges de messages explicites.

Le paradigme de programmation par processus communicant est très populaire car il permet une programmation très proche de ce qui se passe réellement dans un ordinateur

à mémoire distribuée. D'ailleurs on désigne quelquefois les processus sous le nom de processeurs virtuels. Dans la littérature, les processus sont également appelés tâches, mais pour éviter la confusion, nous réserverons le terme de tâches à la notion abstraite de groupes d'instructions indivisibles définie dans le chapitre précédent (cf. 1.3.1).

En pratique, pour faciliter la programmation des communications, il existe beaucoup de bibliothèques de fonction d'échanges de messages qui utilisent ce paradigme [27]. Dans la suite, nous décrirons brièvement PVM [60] et MPI [47], les deux bibliothèques les plus utilisées actuellement. Elles permettent d'écrire des applications portables, et même d'utiliser des machines et des réseaux hétérogènes. C'est ce qui fait leur succès.

2.1.2 Communication point-à-point

La communication point-à-point est l'envoi d'un message d'un processus à un autre. C'est la fonction de communication la plus élémentaire, la brique de base qui sert à la construction des autres opérateurs de communications. La communication point-à-point comprend deux primitives :

– **une primitive d'envoi** qui en plus de l'adresse des données à envoyer demande l'identité du destinataire, et une identité de message. L'identité du message permet de distinguer les messages allant vers la même destination.

```
envoyer( destination, id_du_message, contenu )
```

– **une primitive de réception** qui nécessite l'identité du processus émetteur, l'identité du message, et l'adresse de la zone mémoire qui reçoit les données. L'identité du message, plus l'identité du processus émetteur permettent d'apparier la primitive de réception avec la primitive d'envoi correspondante.

```
recevoir( émetteur, id_du_message, contenu )
```

Il existe essentiellement deux protocoles pour acheminer un message de sa source vers sa destination :

– les communications sont **synchrones** si l'émetteur attend que le processus récepteur soit prêt à recevoir les données pour les envoyer. Ensuite il attend la fin de la réception des données. La communication synchronise les deux processus. On parle aussi de communication par rendez-vous ;

– les communications sont **asynchrones** si l'émetteur envoie directement les données, sans vérifier la réception. C'est plus rapide qu'une communication par rendez-vous. Les deux processus n'ont pas besoin de se synchroniser. Mais la communication n'est pas fiable, et il n'y a plus de contrôle de flux. En effet, que se passe-t-il si le récepteur n'est pas capable de recevoir les données ? C'est à l'utilisateur de garantir que le récepteur est prêt. Concrètement, deux situations peuvent se présenter : soit le récepteur place les données dans un tampon et l'utilisateur doit garantir que le tampon n'est pas plein ; soit l'émetteur suppose que la requête de réception correspondante est déjà postée, c'est-à-dire qu'un espace mémoire est déjà réservé pour recevoir les données du message.

Les primitives de communications peuvent être bloquantes ou non-bloquantes. En effet, on parle d'envoi non-bloquant si le processus émetteur continue l'exécution sans attendre que le message soit effectivement envoyé. De même, on parle de réception non-bloquante si le processus récepteur continue sans attendre la réception effective du message. En attendant la réception effective des données, le processus récepteur peut exécuter un calcul qui ne nécessite pas les données à recevoir. De même, le processus émetteur peut continuer à calculer à condition de ne pas modifier les données qu'il envoie. On utilise deux primitives supplémentaires pour contrôler la fin effective des communications non-bloquantes :

- une primitive qui teste si l'envoi ou la réception sont effectivement terminés. Elle permet, dans le cas d'une réception non-bloquante, de savoir s'il est possible d'utiliser les données reçues ou s'il faut encore attendre. Dans le cas d'un envoi non-bloquant, cette primitive permet de savoir si les données sont effectivement envoyées et si on peut les modifier sans risque d'incohérences ;
- une primitive qui attend la fin effective de l'envoi ou de la réception des données. Elle suspend l'exécution du processus en attendant.

Le caractère asynchrone des communications non-bloquantes permet d'introduire plus de souplesse dans les programmes parallèles, en particulier, lors de communications par rendez-vous. Elles permettent également d'entrelacer les calculs et les communications pour masquer les temps des communications.

Elles simplifient les communications croisées. En effet, si les communications sont synchrones, l'ordre des primitives de communications entre les processus est très important, une erreur entraîne un inter-blocage. Par exemple si deux processus désirent échanger des données, l'utilisateur doit imposer un ordre pour savoir qui commence par envoyer et qui commence par recevoir. Si les communications sont asynchrones ou non-bloquantes, l'ordre n'est pas important et l'utilisateur peut programmer les communications de l'échange de la même façon pour les deux processus. En pratique cela permet d'utiliser le même programme pour chacun des processus. Seul le numéro du processus et les données de départ permettent de distinguer les différentes instances d'exécution. Ce mode de programmation s'appelle SPMD[1].

2.1.3 Communication collective

En plus des communications point-à-point, les bibliothèques de passage de messages offrent des primitives pour les communications collectives. Ces primitives permettent d'exprimer facilement et directement des communications qui impliquent l'ensemble des processus du programme parallèle. Parmi les communications collectives les plus courantes, on trouve :

- **la diffusion** (*broadcast*) est l'envoi d'une donnée d'un processus vers tous les autres. Si la donnée à envoyer dépend du processus destinataire, on parle de distribution ;

[1] SPMD = Single Program Multiple Data, programme unique données multiples

– **le regroupement** (*gathering*) est la réception de données provenant de tous les processus sur un seul processus. C'est l'opération inverse de la distribution. Une variante possible de cette communication collective est le produit itéré. On utilise un opérateur binaire associatif (somme, produit maximum, ...) pour combiner deux à deux les données reçues. On appelle cette opération un regroupement avec recombinaisons ;
– **l'échange total** (*all to all*) est la diffusion simultanée de données à partir de chacun des processus. Chacun des processus communique avec tous les autres. Si, à la place des diffusions, les processus effectuent des distributions (c'est-à-dire que les données envoyées dépendent de la destination), on parle d'échange total personnalisé.

Les bibliothèques de passage de messages permettent de créer des groupes de processus. Ainsi, elles permettent d'utiliser les primitives de communications collectives seulement sur les processus d'un groupe.

Les communications collectives sont très utiles pour écrire des algorithmes numériques réguliers. Elles permettent entre autres, de faire abstraction de la topologie de la machine et de rester efficace après le portage sur une autre machine. En effet ainsi, l'application numérique utilise des primitives de communications bien identifiées, et bien optimisées pour la machine-cible lors du développement de la bibliothèque de passage de messages.

Mais les opérations de communication globales disponibles actuellement sont bloquantes et synchrones. Elles synchronisent les processus impliqués. Pour cette raison, elles ne sont pas toujours adaptées aux applications irrégulières. En effet, pour ces applications, les synchronisations pénalisent souvent les performances car il est difficile d'équilibrer la charge de calcul des processeurs pour que chacun commence l'opération de communication au même moment. De plus, les opérations de distributions permettent d'envoyer des données différentes en fonction de la destination, mais la taille de ces données doit être identique. Plus difficile encore, nous verrons qu'il existe des problèmes irréguliers où une phase de communication collective est bien identifiée, mais le schéma est du type *tous-pour-plusieurs* (chacun des processeurs communique avec un sous-ensemble des autres) sans topologie de communication régulière (comme un anneau, une grille, ...). C'est pourquoi, on préférera utiliser des communications point-à-point non-bloquantes, plus souples pour traiter les communications de ces applications.

2.1.4 La bibliothèque de passage de messages PVM

PVM (*Parallel virtual Machine*) est un environnement de programmation parallèle, par processus communicants, très populaire. Développé et distribué par l'université du Tenessee et par le Oak Ridge National Laboratory, il est devenu un standard de fait. Développé à l'origine pour utiliser des réseaux de stations de travail homogènes ou hétérogènes comme des machines parallèles virtuelles, il est également disponible sur la plupart des machines parallèles dédiées.

La portabilité est un atout majeur de PVM. Elle permet notamment de développer des

programmes parallèles sur un réseau de stations de travail, en utilisant les nombreux outils disponibles, et de porter ensuite facilement le code sur une machine parallèle spécifique.

L'environnement de programmation par processus communicants PVM est composé :

- d'une bibliothèque de primitives de communications et de gestion de la machine virtuelle. Ces primitives sont accessibles à l'utilisateur à partir d'un programme écrit en langage C, C++ ou FORTRAN ;
- d'une console, interface qui permet d'initialiser la machine parallèle virtuelle, et de gérer inter-activement les processus d'un programme parallèle ;
- d'un démon, placé sur chacun des noeuds de la machine. C'est un processus spécialement chargé des communications et de la gestion des processus placés sur son noeud de calcul. Plus précisément, il est chargé d'acheminer les messages des processus placés sur son noeud, vers les démons des autres noeuds de la machine. Bien sûr, il est aussi chargé de recevoir les messages destinés à son processeur. Il les redistribue ensuite au processus récepteur et à la demande de celui-ci. En particulier, les démons PVM utilisent des tampons de communication et offrent un mécanisme de communication asynchrone très pratique.

La figure 2.1 montre l'organisation de PVM sur une machine parallèle virtuelle.

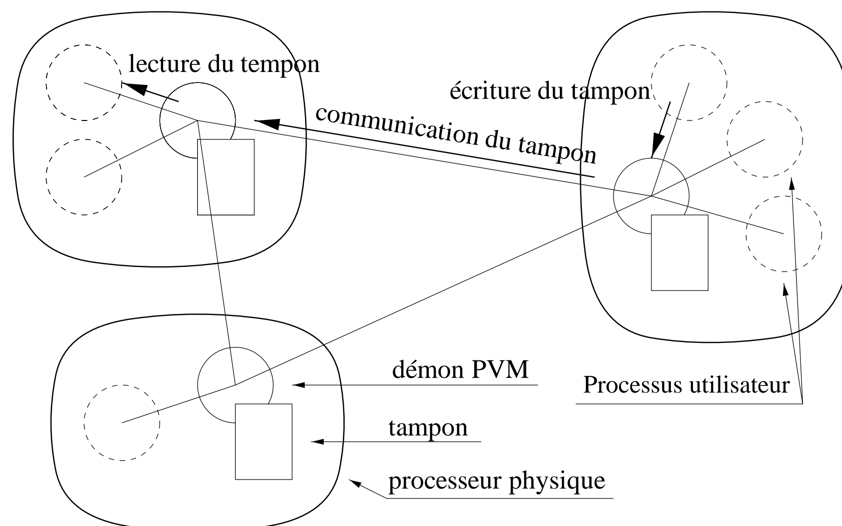


FIG. 2.1: Organisation des processus dans une machine PVM, et chemin des communications entre deux processus utilisateurs.

En plus des primitives de communications point-à-point et des primitives de communications collectives classiques que nous avons décrites dans le paragraphe précédent, PVM offre des primitives pour la gestion de la machine virtuelle. On peut ainsi dynamiquement pendant l'exécution d'une application, ajouter ou supprimer des noeuds de calcul à la machine virtuelle (les démons). On peut aussi naturellement lancer de nouveaux processus comme autant de processeurs virtuels. PVM offre également un mécanisme de tolérance aux pannes.

2.1.5 La spécification MPI

MPI (*Message-Passing Interface*) est la spécification d'un ensemble de fonctions pour la communication par échange de messages. Elle a été développée en regroupant les meilleurs aspects des bibliothèques de communications existantes. Le groupement de chercheurs et d'industriels à l'origine de cette spécification avait pour but de construire une interface standard et portable pour l'expression du parallélisme par processus communicants. Il existe plusieurs implémentations de cette interface, les constructeurs telle que IBM ou CRAY ont développé des versions spécifiques pour leurs machines respectives. Il existe également des versions pour réseau de stations de travail comme LAM/MPI[2] ou MPICH[3].

MPI possède beaucoup de fonctionnalités parmi lesquelles on trouve :

- un ensemble très complet de primitives de communication point-à-point, permettant de choisir le protocole de communication : synchrone, ou asynchrone. Ces primitives existent en version bloquante et non-bloquante ;
- des primitives de communications collectives permettant de réaliser les communications globales usuelles. Ainsi que des primitives pour les opérations globales de réduction ou de calcul de préfixe ;
- des fonctions pour la description des types de données complexes ;
- des primitives pour la gestion des groupes de processeurs et des contextes de communications. En particulier, les contextes de communications permettent aisément de développer des bibliothèques de fonctions parallèles, sans risque de conflit lors de leurs utilisations conjointes dans un même programme ;
- et un ensemble de fonctions pour définir des topologies de communications.

En revanche, contrairement à PVM, MPI ne possède pas de fonctions pour la gestion de la machine virtuelle. Il n'est pas possible d'ajouter ou d'enlever des processeurs en cours d'exécution. Il n'existe pas de mécanismes de tolérance aux pannes. Il manque également des spécifications pour les entrées/sorties en parallèle. Mais MPI2 la version future de MPI, en cours de spécification, devrait répondre à ces limitations.

En conclusion, MPI est une interface de programmation parallèle, par processus communicants, très complète. Elle permet une description très précise des communications, tout en restant portable et efficace.

[2] LAM/MPI est développé à l'Ohio Supercomputer Center

[3] MPICH est développé au Argonne National Laboratory

2.2 Processus léger et échange de messages

Construire des programmes parallèles efficaces nécessite d'utiliser au mieux toutes les ressources de la machine, pour faire des opérations directement utiles au problème traité. En particulier pour avoir de bonnes accélérations, l'utilisation maximale de la disponibilité du processeur est primordiale. Dans le cas de problèmes réguliers, la parallélisation conduit souvent à des algorithmes globalement synchrones qui sont des enchaînements de phases de calcul équilibrées et de phases de communications pour la redistribution des données [26]. Une programmation analogue pour les problèmes irréguliers est impossible, car l'équilibre de charge des phases de calcul est souvent très difficile à obtenir. Pour avoir des bonnes performances, il faut impérativement tenir compte de leur caractère imprévisible.

2.2.1 Parallélisme et multiprogrammation

La multiprogrammation où utilisation partagée du processeur par plusieurs flots d'instructions est employée depuis longtemps par les systèmes d'exploitation comme UNIX. Elle leur permet de partager le processeur entre plusieurs applications. Mais surtout, elle permet d'utiliser au mieux le processeur en entrelaçant les flots d'instructions. Ainsi, si un flot d'instructions se bloque pour attendre la fin de l'accès à une ressource, un autre flot d'instructions peut prendre la main et utiliser le processeur.

Dès lors, l'idée d'utiliser la multiprogrammation dans une application parallèle semble assez naturelle. L'utilisation concurrente de plusieurs flots d'instructions (ou fils d'exécution) par noeud doit permettre d'utiliser en parallèle les ressources du noeud de calcul et de garder les processeurs actifs le plus longtemps possible. Essentiellement, cela signifie qu'il faut entrelacer l'utilisation des processeurs et du réseau. Par exemple, quand un fil d'exécution se bloque dans l'attente d'une communication, un autre fil d'exécution doit pouvoir utiliser le processeur pour traiter d'autres calculs utiles pour l'application.

Utiliser la multiprogrammation dans une application signifie concrètement qu'il faut augmenter le parallélisme virtuel et créer des fils d'exécution concurrents sur les noeuds de la machine. Dans le paradigme de programmation par processus communicants, une application est déjà décrite sous la forme de fils d'exécution multiples. Une façon simple d'exploiter la concurrence sur les noeuds consiste à exécuter l'application sur des processeurs virtuels plus nombreux que les processeurs physiques. Les processeurs physiques exécutent ensuite plusieurs processeurs virtuels en concurrence. Mais ce n'est pas toujours suffisant pour exploiter la concurrence. En fait, sur chacun des noeuds, il faut trouver un ordonnancement des fils d'exécution qui entrelace l'utilisation du réseau et du processeur, pour garder ce dernier actif. En effet, prenons un problème dans lequel une phase de communication, de type échange total, suit une phase de calcul mal équilibrée. La phase de communication va synchroniser les processeurs et le mauvais équilibre de charge va obliger les processeurs les moins chargés à attendre. Augmenter le nombre de processeurs virtuels ne change rien. Pour garder les processeurs actifs, il faut chercher un autre ordonnancement des tâches. Par exemple, il faut chercher à exécuter des tâches

ne nécessitant que des données locales, en même temps que la phase de communication. Elles pourront ainsi recouvrir les temps d'attente des processeurs les moins chargés. Notons que si la machine le permet, cet ordonnancement permet également de recouvrir les temps de communications par des calculs. Nous donnerons un exemple plus précis dans le paragraphe suivant (cf. 2.3).

L'entrelacement des calculs et des communications n'est pas la seule possibilité offerte par la multiprogrammation des noeuds pour faciliter l'implémentation efficace des algorithmes parallèles. La multiprogrammation peut également :

- entrelacer l'utilisation des entrées/sorties, si le système d'exploitation le permet ;
- aider à la construction d'un mécanisme d'accès à la mémoire distante, en utilisant des primitives de lectures/écritures à distance ;
- permettre la circulation de l'information de charge des processeurs pour un régulateur dynamique ;
- équilibrer la charge de calcul en implémentant un mécanisme de migration des fils d'exécution [100].

Dans la suite, nous utiliserons la multiprogrammation essentiellement pour entrelacer les calculs et les communications.

2.2.2 Description de la multiprogrammation

Les fonctions de communications asynchrones, présentes dans la plupart des bibliothèques de programmation par processus communicants, constituent déjà un moyen de décrire l'utilisation entrelacée des calculs et des communications. Elles sont suffisamment générales pour décrire tout les entrelacements calculs/communications possibles. De ce fait, les communications asynchrones sont largement utilisées dans les algorithmes réguliers pour le recouvrement des communications [26]. Leurs utilisations pour les problèmes irréguliers est plus difficile. En effet, le caractère imprévisible de ces problèmes nécessite un entrelacement dynamique des calculs et des communications. Cela conduit à une étude de cas et à la construction d'un automate dont les états indiquent les tâches à exécuter et dont les transitions dépendent de l'avancement des communications.

Une autre solution, consiste à utiliser un mécanisme d'exécution concurrente, et à séparer ainsi la description de la concurrence de l'entrelacement dynamique des calculs et des communications. Ce mécanisme doit pouvoir :

- **créer** ou **détruire** un fil d'exécution ;
- **donner la main** à un autre fil d'exécution (changer le contexte d'exécution), si le fil courant se bloque sur l'attente d'un événement.

Les processus UNIX, ou processus lourds, permettent de réaliser ces opérations. De plus, comme nous l'avons vu précédemment, des environnements de programmation comme PVM permettent de gérer plusieurs processus par noeud de calcul. Mais les opérations de création, destruction et changement de contexte sont coûteuses. En effet, les processus lourds possèdent, en plus de leur contexte d'exécution (registres et pile), un espace

d'adressage et de protection et des descripteurs pour les opérations d'entrée/sortie. Les opérations sur les processus lourds demandent l'intervention du système qui doit gérer les espaces d'adressage et de protection.

Processus légers

Les processus légers (*thread*) possèdent seulement un contexte d'exécution (registres et pile), ils partagent un espace d'adressage et de protection unique. Ils partagent également les descripteurs d'entrée/sortie du processus lourd auquel ils appartiennent. La gestion des processus légers nécessite seulement la manipulation du contexte d'exécution, elle est donc moins coûteuse que la gestion des processus lourds. Par exemple, un changement de contexte est 10 à 1 000 fois plus rapide.

Les noyaux de processus légers les plus connus sont les processus légers de la norme POSIX [98]. Le lecteur intéressé peut trouver une comparaison détaillée des noyaux de processus légers dans la thèse de Michel CHRISTALLER [33]. Ces noyaux se caractérisent par leur politique d'auto-ordonnancement, c'est-à-dire par la façon dont est gérée l'exécution concurrente des processus légers. En effet, il faut déterminer la façon de partager le processeur et l'ordre dans lequel on exécute les processus légers. Ainsi, pour aider l'auto-ordonnanceur à choisir l'ordre d'exécution des fils, l'utilisateur peut leur affecter des priorités. L'utilisation de ces priorités par le noyau dépend de la politique de partage du processeur. Il existe essentiellement deux politiques d'auto-ordonnancement (préemption) des processus légers :

- **la politique FIFO**[4]. Les processus sont actifs jusqu'à ce qu'ils se bloquent ou se terminent. Ils ne sont pas interrompus, les changements de contexte ont lieu seulement dans un petit nombre de primitives du noyau de processus légers. Les changements de contexte sont réduits au minimum. Mais cette politique pose des problèmes de réactivité. En effet, même si on utilise des priorités, les fils d'exécution ne sont pas interrompus. Par exemple, si un fil d'exécution de plus haute priorité se débloque, le fil courant continue son exécution jusqu'à ce qu'il se bloque à son tour. Les fils d'une priorité donnée ne peuvent s'exécuter que si tous les fils de priorités supérieures sont bloqués. À un même niveau de priorité, les fils sont exécutés les uns à la suite des autres. Pour améliorer la réactivité, l'utilisateur peut introduire des changements de contexte explicites. C'est-à-dire qu'un fil d'exécution peut explicitement demander à l'auto-ordonnanceur d'activer un autre processus prêt. Cette primitive permet, au niveau de l'utilisateur, de programmer un partage coopératif du processeur. Le choix de la place des primitives de changement de contexte est un compromis entre la réactivité et un minimum de changements de contexte ;
- **la politique de temps partagé**. Le processus léger s'exécute au maximum pendant un quantum de temps, puis il est préempté par le noyau au profit du suivant. Les priorités déterminent la fréquence de préemption de chacun des fils d'exécution. Cette politique résout le problème de réactivité, mais le nombre de changements de contexte peut être important. De plus, les fils d'exécution ne maîtrisent pas le

[4]FIFO = firt in first out, premier entré, premier sorti.

moment où ils sont interrompus, cela complique la programmation car il faut qu'ils soient réentrants[5].

En plus de faciliter l'utilisation entrelacée des ressources, les noyaux de processus légers permettent une programmation efficace grâce à un partage de l'espace d'adressage. En effet, les fils d'exécution peuvent partager des données sans avoir à les copier. Mais l'absence de protection entre les fils d'exécution introduit des problèmes de programmation liée à la concurrence. À savoir, des problèmes de :

– **conflits d'accès aux données**. Un conflit intervient quand un processus léger écrit une donnée alors qu'un autre lit la même donnée. Il faut verrouiller l'accès aux données partagées pour garantir la cohérence des accès. Les noyaux de processus légers offrent un mécanisme de sémaphores, pour synchroniser les processus légers et assurer l'unicité des accès aux variables partagées ;

– **inter-blocage**. Le verrouillage des données peut créer des situations où les processus se bloquent mutuellement. Par exemple, un inter-blocage survient si deux processus légers verrouillent une donnée et si, ensuite, chacun essaye d'accéder à la donnée de l'autre ;

– **famine**. La famine intervient quand un processus léger n'arrive jamais à accéder à une donnée, car elle est toujours verrouillée par un autre.

Les problèmes de concurrence dépendent de la politique d'auto-ordonnancement des processus légers. En particulier, l'auto-ordonnancement en temps partagé impose la protection rigoureuse de toutes les variables partagées. Si une portion de code satisfait à la protection rigoureuse de ces variables partagées, elle est dite réentrante, c'est-à-dire stable aux changements de contexte aléatoires. Ceci peut être très coûteux. Par exemple, l'accumulation de valeurs dans une variable par des processus concurrents, nécessite des opérations de lectures/écritures très rapprochées. Un changement de contexte entre la lecture et l'écriture peut créer des incohérences. La protection de la variable d'accumulation est nécessaire, mais si les accumulations sont fréquentes, la protection de la variable entraîne un sur-coût important. Une protection aussi stricte des variables partagées n'est pas nécessaire avec un auto-ordonnancement FIFO car les changements de contexte sont isolés dans les primitives bien identifiées.

Une autre difficulté, liée au temps partagé, est la mesure des performances. En effet, comment mesurer le temps de calcul d'une portion de code si l'exécution de ce code peut être interrompue à tout les moments pour réaliser un autre calcul ? C'est impossible sans tenir compte des changements de contexte, et sans introduire un sur-coût très important.

En fait, le choix d'une politique d'auto-ordonnancement dépend de l'application. Si les processus légers partagent peu de données, on préférera favoriser l'équité en utilisant le temps partagé. Mais si les processus utilisent beaucoup de données communes, on préférera limiter la protection des données en utilisant une politique FIFO.

[5]Nous allons définir ce terme par la suite.

2.2.3 La librairie ATHAPASCAN-0A

Un des axes de recherche du projet APACHE[6] [14], dans le cadre duquel ce travail a été fait, consiste à réaliser une plate-forme d'exécution pour les applications irrégulières qui utilise la multiprogrammation des noeuds. Plus précisément, il s'agit de combiner un noyau de processus légers, et une librairie de communication.

Fondements

La librairie ATHAPASCAN-0A[7] est la première maquette réalisée de cette plate-forme d'exécution. Au départ, elle a été conçue pour la parallélisation d'applications ayant des graphes de tâches série/parallèle (cf. figure 2.2) [69]. Ces graphes sont caractéristiques d'une approche de construction d'algorithmes parallèles *diviser pour paralléliser*. Un algorithme de ce type comporte trois phases :

1. une phase de **découpe** du problème, qui divise le problème initial en plusieurs sous-problèmes indépendants ;
2. une phase de **résolution**, dans laquelle on résout récursivement en parallèle chacun des sous-problèmes ;
3. une phase de **fusion**, pour regrouper les résultats de chacun des sous-problèmes et construire ainsi la solution du problème initial.

Ensuite, le concept consiste à lier la notion abstraite de tâche à la notion de fils d'exécution. Ainsi un fil d'exécution commence par découper le problème. Puis il crée plusieurs fils d'exécution pour résoudre les sous-problèmes en parallèle. Enfin il attend la fin de l'exécution de tout les fils créés pour fusionner les résultats. La figure 2.2 montre sur la droite la décomposition en fils d'exécution correspondant au graphe de tâches de la partie gauche.

Voici les autres objectifs qui ont guidé la réalisation de la librairie ATHAPASCAN-0A :

- **portabilité**. L'évolution des machines parallèles étant souvent plus rapide que le développement d'applications parallèles, la portabilité est impérative ;
- **efficacité et extensibilité**. La plate-forme d'exécution doit pouvoir s'exécuter sur une machine massivement parallèle, sans sur-coût de gestion excessif ;
- **structurant**. L'objectif initial était également de construire un langage structuré pour l'expression du parallélisme de contrôle.

Description

ATHAPASCAN-0A offre un ensemble de primitives écrites en C qui permettent une expression procédurale du parallélisme de contrôle d'une application. Plus précisément, le parallélisme s'exprime par des appels asynchrones de procédures distantes. Contrairement à une procédure classique, la procédure distante est calculée par un autre processeur que le processeur appelant. L'appel de la procédure est asynchrone car ses résultats ne

[6] APACHE signifie : Algorithmique Parallèle et PArtage de CHargE.
[7] Athapascan est la langue des Apaches.

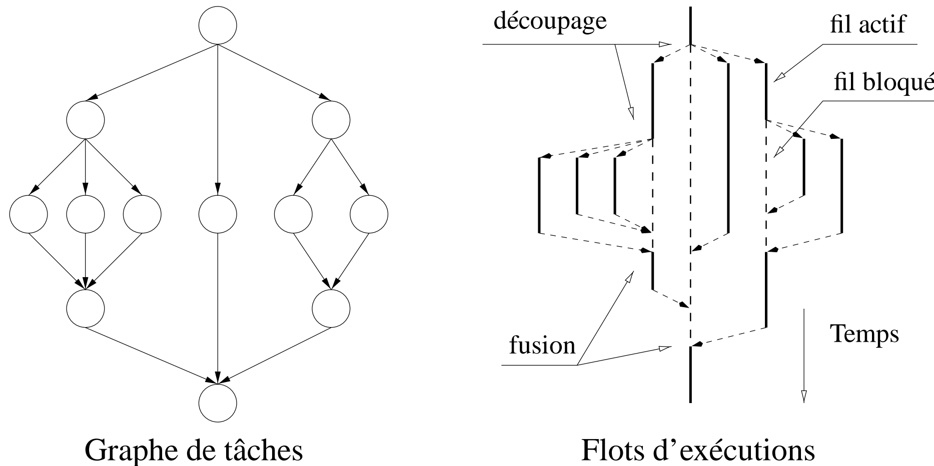


FIG. 2.2: Exemple de graphe de tâches série/parallèle, et décomposition en flots d'exécution correspondante pour ATHAPASCAN-0A.

sont pas disponibles juste après son appel, mais seulement après l'appel d'une primitive spéciale, qui bloque l'appelant en attendant les résultats (c'est analogue à la primitive d'attente d'une communication asynchrone). L'exécutif ATHAPASCAN-0A réalise un modèle Client/Serveur basé sur l'appel de procédure à distance. C'est un ensemble de serveurs capables de répondre à des appels de procédures à distance de façon synchrone ou asynchrone. Chacun de ces appels entraîne la création d'un processus léger sur le noeud distant pour exécuter la procédure. Les communications sont dissimulées dans le mécanisme de passage des paramètres et des résultats aux procédures distantes [31] [34]. ATHAPASCAN-0A est construit à partir d'un noyau de processus légers FIFO sans priorité et à partir de la bibliothèque de communication PVM.

Plus concrètement, le programmeur commence par définir des **modèles de services**, ce sont des procédures comportant obligatoirement deux paramètres : un descripteur pour les données, et un autre pour les résultats. Ces modèles de services comportent naturellement trois parties :

1. une partie pour récupérer les arguments du service, à partir du descripteur de données, c'est équivalent à une réception des données ;
2. la partie de traitement, comme une fonction ordinaire ;
3. enfin, la partie qui place les résultats dans le descripteur de résultat, c'est équivalent à un envoi.

La figure 2.3 donne le squelette d'un appel de service.

Ensuite, le programmeur déclare ces modèles de services, au début du programme. Les processus du programme parallèle comportent ainsi un ensemble de services auxquels ils peuvent répondre suite à un appel distant.

Dans le code, l'utilisation d'un service comporte trois phases :

1. on utilise la primitive qui crée une instance de service. Cette primitive demande : le numéro du service à appeler, le numéro du processeur qui va exécuter le service,

```
EP_MODEL_MACRO( nom_du_service, desc_arg, desc_res )
BEGIN_EP
  déballage des arguments (1) :
  UnPack( desc_arg, types des arguments , arg1, arg2, ... );
  ... traitement, comme une fonction ordinaire (2) ...
  emballage des résultats (3)
  Pack( desc_res, types des résultats , res1, res2, ... );
END_EP
```

FIG. 2.3: Squelette d'un modèle de service en ATHAPASCAN-0A.

et les arguments du service. Elle retourne une variable de synchronisation pour les résultats du service ;

2. l'appel d'un service est asynchrone, l'appelant peut donc continuer son exécution ;
3. on utilise ensuite la primitive qui bloque le processus courant en attendant la fin du service. Elle utilise la variable de synchronisation et elle retourne les résultats du service.

La figure 2.4 montre l'appel d'un service.

```
... appel du service (1) :
Spawn( ..., types des arguments , arg1, arg2, ..., argn, &sync );
... traitement (2) ...
récupération des résultats (3) :
WaitSpawnRes( sync, types des résultats , res1, res2, ..., resn );
```

FIG. 2.4: Squelette d'un appel de service en ATHAPASCAN-0A.

Les graphes de tâches série/parallèle peuvent être décrits directement avec les primitives d'appel de procédures à distance sans effet de bord. C'est-à-dire des procédures qui ne modifient pas d'autres données que les données locales au processus léger qui les exécute. Nous l'avons déjà écrit au paragraphe précédent, il suffit de lier la notion abstraite de tâche à la notion de service à distance. Ainsi, la création des tâches enfants d'une tâche parente se fait directement en appelant de façon asynchrone le modèle de service correspondant aux tâches enfants. Les dépendances de données entre la tâche parente et ses tâches enfants sont incluses dans l'envoi des paramètres des services et dans la réception des résultats des services. Aucune autre communication de données entre les services n'est nécessaire pour les graphes de tâches de ce type. Autre avantage de cette programmation structurée, elle évite les pièges de la programmation concurrente. Au niveau utilisateur, il n'y a pas de problème de conflits d'accès aux données car elles sont toutes locales au processus léger qui exécute le service. Ceci élimine également les problèmes d'inter-blocage inhérents à l'accès des ressources partagées.

Mais tous les graphes de tâches ne sont pas du type série/parallèle. En particulier, les graphes de tâches issus d'une parallélisation par décomposition du domaine physique (cf.

exemple 2.3) ne sont pas de ce type. Pour ne pas se limiter à la programmation d'applications ayant des graphes de tâches série/parallèle, la communication de données entre les services ne peut pas se faire uniquement de façon structurée par le passage des paramètres et le retour des résultats aux procédures distantes. Elle se fait également de façon non structurée, par le partage de la mémoire d'un processeur entre les services de ce processeur. Des primitives pour synchroniser les services d'un même processeur ont été ajoutées. Elles permettent le partage cohérent de données entre les services d'un même processeur. Évidement, si l'utilisateur choisit de partager des données entre les services, il doit garantir la cohérence des accès et l'absence d'inter-blocage.

Nous avons donné seulement les primitives de base de ATHAPASCAN-0A, le lecteur intéressé par plus de détails est invité à consulter la thèse de Michel CHRISTALLER [33].

Conclusions sur ATHAPASCAN-0A

ATHAPASCAN-0A étant la première plate-forme du projet combinant un noyau de communications et un noyau de processus légers, nous l'avons utilisée pour une partie de notre travail (cf. chapitre 5). Le petit nombre de primitives a permis un apprentissage rapide.

Mais ne disposant pas d'un graphe de tâches série/parallèle (cf. 5.2), nous avons dû utiliser la mémoire pour partager des données entre les services. Il faut alors ajouter des synchronisations entre les services d'un même processeur, pour garantir la cohérence des accès aux données partagées. On perd ainsi l'avantage de la programmation structurée qu'offre normalement le paradigme de programmation d'ATHAPASCAN-0A. Cela conduit à une programmation peu homogène, qui mélange des concepts de haut niveau comme les appels de procédures distantes avec des concepts de plus bas niveau de protection des variables partagées.

Parmi les environnements de programmation parallèle qui utilisent des processus légers, seul PM^2 [101] [100] offre un paradigme de programmation structuré par des appels de procédures distantes. Comme celui d'ATHAPASCAN-0A, il se limite au traitement des graphes du type série/parallèle. Les autres environnements de programmation n'imposent pas de paradigme de programmation structurée. En effet, dans NEXUS [59], les communications entre les processeurs se font par des appels de service à distance (RSR), ce sont des appels de procédures à distance asynchrones sans retour de résultats. NEXUS possède également des primitives de synchronisation pour partager les données entre les services d'un même processeur. Inversement TPVM [54] interdit le partage de données entre processus légers, l'échange de données se fait uniquement par des communications point-à-point explicites entre les processus légers.

Enfin, signalons le travail de Michel RIVIERE qui a essayé de conserver une programmation structurée indépendamment du graphe de tâches. L'idée consiste à généraliser le concept d'appels de procédures distantes en utilisant des multi-procédures. Ce sont des procédures distantes acceptant des paramètres provenant de plusieurs appelants et retournant des paramètres à plusieurs appelants [112].

Notons, à l'avantage de cette première version, les outils complémentaires développés pour cet environnement ; des outils pour l'analyse de performances [123] et la recherche

de bogues. En particulier citons un travail sur la réexécution déterministe des programmes parallèles, destiné à trouver des bogues non-déterministes liés à l'exécution concurrente [53]. Le mécanisme comporte une partie enregistrant le déroulement de l'application, et une seconde partie permettant de rejouer l'exécution dans le même ordre.

2.2.4 La librairie ATHAPASCAN-0B

ATHAPASCAN-0B est la deuxième version du support d'exécution pour applications irrégulières réalisé dans le cadre du projet. De conception plus pragmatique, il n'impose pas de paradigme de programmation parallèle structurée. Au contraire, il est conçu pour être multi-paradigme, et il offre un ensemble de primitives très complet pour l'expression du parallélisme et de la multiprogrammation. Il laisse l'utilisateur choisir la façon de programmer la mieux adaptée à son problème. ATHAPASCAN-0B est construit, pour être portable et efficace, à partir d'un noyau de processus légers à la norme POSIX [98] et du noyau de communication MPI. La bibliothèque MPI a été préférée à PVM car elle est réentrante. De plus, elle exploite souvent mieux les réseaux de communication spécialisés des machines parallèles. En particulier sur le IBM-SP1, MPI est plus performant que PVM.

Fonctionnalités

En pratique, ATHAPASCAN-0B offre un ensemble de primitives écrites en C pour la gestion des processus légers et pour les communications entre les processus légers. Pour la gestion des processus légers, les primitives permettent :

- la création de processus légers localement à un processeur ;
- la création de processus légers à distance, sur un autre processeur ;
- et la protection des variables partagées entre processus légers d'un même processeur, grâce à des sémaphores.

Les primitives de communication entre les processus légers sont très proches des primitives de communication point-à-point synchrone de MPI. La bibliothèque ATHAPASCAN-0B comprend donc, des primitives d'envois et de réceptions bloquantes et non-bloquantes, et des primitives pour attendre ou tester la fin des communications non-bloquantes.

Les primitives de communication utilisent un mécanisme de ports de communications pour établir un lien entre les deux processus légers impliqués dans une communication. Ainsi l'émetteur envoie des données sur un port et à destination d'un processeur donné. Le récepteur lit les données en provenance d'un processeur et d'un port. Ce mécanisme de ports de communication évite de nommer les processus légers pour les identifier.

Une couche logicielle écrite au-dessus de ATHAPASCAN-0B offre un ensemble de primitives d'accès à la mémoire distante (*Remote Memory Access*). Elle permet aussi d'utiliser des tubes de communications. Le processus léger émetteur place des données dans le tube et le processus léger récepteur les lit au fur et à mesure. En fait, côté émetteur, les données sont placées dans un tampon de taille fixe. Quand le tampon est plein, il est envoyé de façon bloquante, puis réutilisé pour placer la suite des données. Du côté

du récepteur, les données sont lues du tampon, quand le tampon est vide une requête de réception bloquante est émise pour recevoir la suite des données. La communication bloquante des tampons exerce un contrôle de flux entre l'émetteur et le récepteur. Les tubes de communications sont très pratiques pour envoyer des messages dont on ne connaît pas la taille au départ. Ils sont également très pratiques pour la distribution initiale de données sur les processeurs. En effet, le processeur maître lit les données d'un fichier et les distribue au fur et à mesure aux processeurs esclaves. De même pour le regroupement de résultats, le processeur maître combine les données pour les écrire dans un fichier, il les reçoit au fur et à mesure des autres processeurs. Ici le contrôle de flux évite l'engorgement du processeur maître.

Ici aussi, nous nous sommes limités à une description succincte des primitives de base de ATHAPASCAN-0B, le lecteur intéressé par plus de détails peut consulter la thèse de Ilan GUINZBURG [70].

Conclusion sur ATHAPASCAN-0B

Cette version du support d'exécution pour applications parallèles irrégulières est plus homogène et plus souple à utiliser que la version précédente. L'utilisateur peut utiliser suivant ses besoins :

- une programmation par appels de procédures distantes, grâce au mécanisme de création de processus légers à distance ;
- une programmation par échanges explicites de messages entre les processus légers ;
- le partage de données entre les processus légers d'un même processeur grâce à des primitives de synchronisation ;
- l'accès aux données à distance grâce à un mécanisme de lecture/écriture à distance.

MPI rend aussi cette version plus rapide. C'est ce gain de performance et la souplesse accrue qui nous a incité à l'utiliser. En effet, comme nous le verrons par la suite dans le chapitre 6, nous n'avons pas lié la notion de tâche et de processus légers. Nous avons regroupé les tâches en fonction des ressources qu'elles utilisent puis nous avons créé des processus légers pour exécuter ces groupes de tâches.

2.3 Programmation synchrone, asynchrone avec ou sans processus légers : Un exemple simple

Une stratégie très utilisée pour paralléliser un algorithme est la décomposition du domaine de simulation en sous-domaines. C'est caractéristique des problèmes modélisant des phénomènes sur un espace géométrique. Que se soit les noeuds d'un maillage ou des particules, les éléments sont distribués dans un espace de simulation. Des équations locales, c'est-à-dire ne faisant intervenir que des éléments voisins, permettent de résoudre itérativement les grandeurs physiques associées à chacun des éléments. La parallélisation consiste à découper l'espace de simulation en autant de sous-espaces que de processeurs. Nous allons étudier les différentes stratégies d'implémentation pour ce type de problèmes.

Pour simplifier, nous utilisons seulement deux sous-domaines A et B, et donc seulement deux processeurs. Le graphe de tâches associé au schéma de calcul comporte les tâches suivantes :

– T_d^A (resp. T_d^B), elle **découpe** le sous-domaine A (resp. B) en un **domaine intérieur** A_i (resp. B_i) dont le calcul des valeurs ne nécessite que des données locales au processeur, et un **domaine frontière** A_f (resp. B_f) dont le calcul nécessite les données des processeurs voisins ;
– T_i^A (resp. T_i^B), elle **calcule** les nouvelles valeurs des éléments du **domaine intérieur** A_i (resp. B_i) ;
– T_f^A (resp. T_f^B), elle **calcule** les nouvelles valeurs des éléments du **domaine frontière** A_f (resp. B_f) ;
– T_r^A (resp. T_r^B), elle **regroupe** les calculs du **domaine intérieur** et du **domaine frontière**, pour reconstruire le nouveau sous-domaine A (resp. B).

La figure 2.5 montre les différents sous-domaines. Elle montre également les dépendances entre les tâches, et la façon dont elles sont placées sur les deux processeurs.

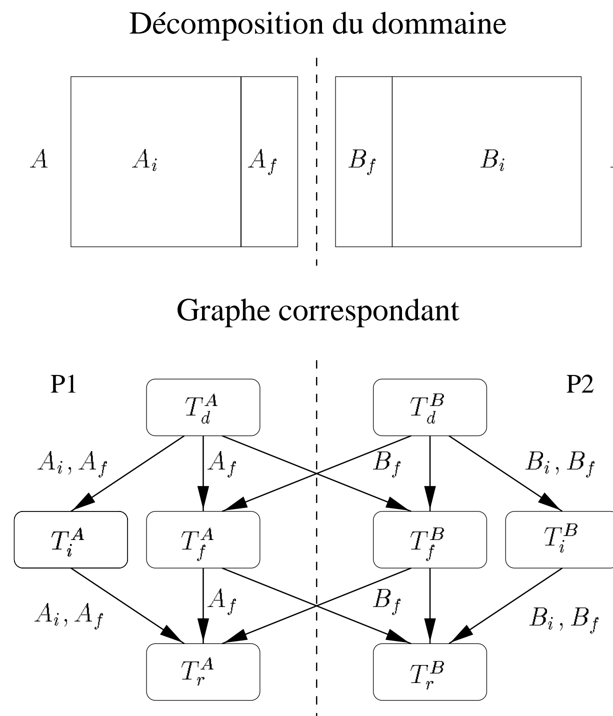


FIG. 2.5: Décomposition en deux sous-domaines et graphe de tâches correspondant.

Supposons que notre exemple soit un problème irrégulier, avec une incertitude sur le coût de calcul des tâches. Ainsi le coût de calcul d'un type de tâche dépend du domaine auquel il s'applique. Concrètement, par exemple pour les tâches qui découpent les sous-domaines, le temps de calcul de T_d^A est différent du temps de calcul de T_d^B. La figure 2.6 montre le diagramme espace-temps de chacune des façons de programmer :

1. **Processus communicants + communications synchrones** (PC+CS). On n'utilise pas de multiprogrammation, mais seulement des processus simples. On peut facilement identifier deux phases de communications : une pour l'échange des frontières avant le calcul de T_f^A (resp. T_f^B), et un autre pour l'échange des frontières après le calcul de T_f^A (resp. T_f^B). Pour ces deux phases de communication on utilise des primitives de communications collectives synchrones. De plus on impose un ordonnancement statique sur chacun des processeurs avec l'ordre suivant $T_d^A < T_f^A < T_i^A < T_r^A$ (resp. $T_d^B < T_f^B < T_i^B < T_r^B$). Comme on peut le voir sur la figure, la synchronisation des communications est pénalisante ;
2. **Processus communicants + communications asynchrones** (PC+AS). On impose toujours le même ordonnancement des tâches mais pour essayer d'entrelacer les calculs et les communications, on utilise des communications point-à-point asynchrones. Ainsi, comme le montre la figure l'absence de synchronisation permet de recouvrir les temps de communications par du calcul ;
3. **Processus communicants + communications asynchrones + ordonnancement par événement** (PC+AS+OE). On utilise toujours des communications asynchrones mais pour améliorer l'entrelacement des ressources, on utilise un ordonnancement par événement. C'est-à-dire que l'ordonnancement des tâches dépend de l'avancement des communications. Ainsi pour notre exemple, on commence par exécuter la tâche de type T_d. Puis, si les données sont prêtes on calcule T_f suivie de T_l, et sinon on calcule T_l suivie de T_f. On termine par le calcul de la tâche de type T_r. La figure montre que l'ordonnancement n'est pas identique sur les deux processeurs. Le recouvrement s'est amélioré, mais il n'est pas parfait ;
4. **processus légers** (PL). On utilise la multiprogrammation des noeuds en affectant un processus léger à chacun des types de tâches. On affecte une priorité plus élevée aux processus légers qui font des communications. Ainsi comme le montre la figure, les tâches de calcul des frontières sont plus prioritaires, elles sont exécutées au plus tôt, dès que les données sont prêtes. Le calcul des tâches T_l permet de recouvrir les temps de communications.

Cet exemple est un peu abstrait mais nous verrons dans le chapitre 6 qu'il est très proche de notre problème réel.

2.4 Conclusion

Le caractère imprévisible des applications irrégulières rend difficile l'utilisation efficace des primitives de communication synchrone traditionnelles. Pour améliorer les performances de ces applications il faut trouver un ordonnancement compatible avec une utilisation entrelacée des ressources de calcul et de communications. Ensuite, il faut exploiter au mieux cet entrelacement. La multiprogrammation des noeuds de calculs est une alternative pratique à la programmation de communication asynchrone pour entrelacer automatiquement l'utilisation des ressources. Nous verrons dans le chapitre 6 concrètement ce que peut apporter la multiprogrammation pour une application irrégulière.

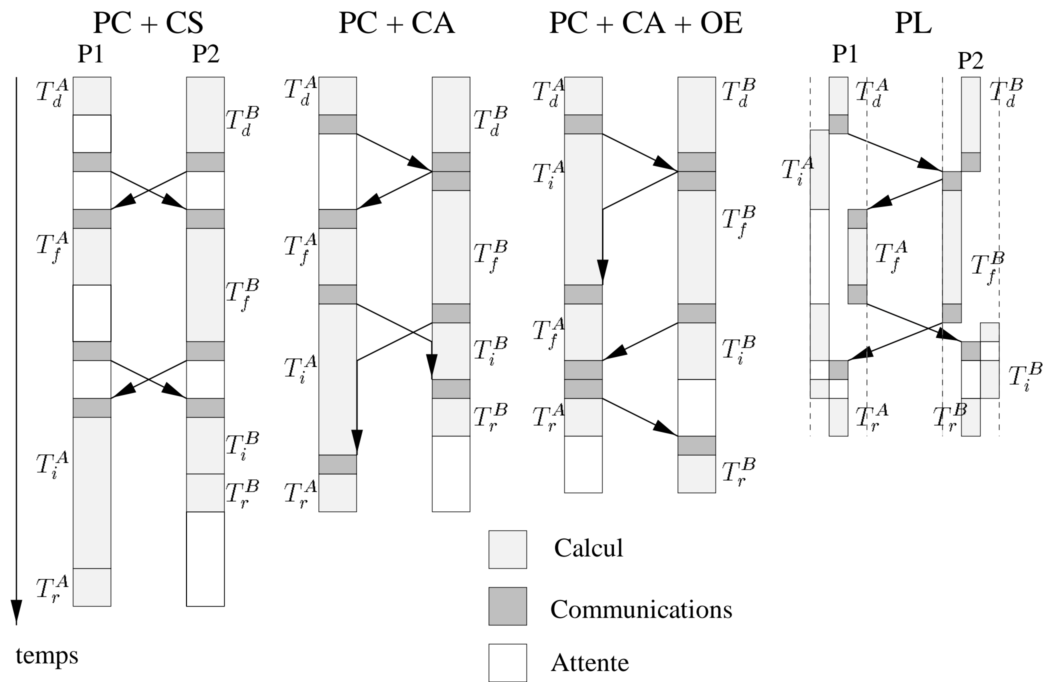


FIG. 2.6: Diagramme espace-temps de chacune des façons de programmer.

Deuxième partie

Modèles et Approximations

Chapitre 3

Modèle classique de dynamique moléculaire

Beaucoup de problèmes physiques très différents sont traités par dynamique moléculaire. Les modèles utilisés pour chacun de ces phénomènes sont souvent très différents. Dans ce chapitre, nous avons isolé les équations utiles pour un modèle de dynamique moléculaire opérationnel pour la biologie. C'est le modèle utilisé ensuite dans notre application. Nous présenterons les principales différences avec les autres modèles de dynamique moléculaire.

3.1 Introduction

La dynamique moléculaire classique est une simulation du mouvement des atomes et des molécules d'un système donné. Cette technique est largement utilisée pour simuler les propriétés des solides, des liquides et des gaz. Elle est utilisée en micro-éléctronique, pour étudier la diffusion d'éléments dopant dans un substrat. En biologie, elle est employée pour étudier les conformations des macromolécules, et pour la compréhension des mécanismes réactionnels des protéines dans les structures biologiques.

Pour simuler le mouvement des atomes, on utilise les équations de la mécanique classique. En effet, pour beaucoup de phénomènes, la résolution des équations de la mécanique quantique, plus proche de la réalité physique, nécessite un volume de calcul trop important, même pour un ordinateur parallèle. En fait, les équations de la mécanique quantique sont surtout utilisées pour étudier des réactions chimiques faisant intervenir un petit nombre d'atomes (de l'ordre de 100 atomes). Nous nous intéressons essentiellement à l'étude du mouvement des atomes des protéines. Pour cela, les modèles classiques sont suffisamment précis. En effet, on cherche à étudier des phénomènes sans réaction chimique entre les molécules. C'est-à-dire qu'il n'y a pas de modification dans les liaisons entre les atomes du système. En biologie, les modèles classiques sont utilisés pour étudier les interactions mécaniques entre les protéines d'assemblages complexes. Le principal besoin est de pouvoir simuler des systèmes moléculaires de grande taille (10 000 voire 100 000 atomes). On souhaite par exemple, modéliser et simuler les protéines d'une

membrane cellulaire dans leur environnement, pour comprendre leur fonctionnement. Il est également nécessaire de réaliser des simulations longues de ces systèmes. Long à l'échelle atomique signifie 1 à 100 nano-seconde. En effet, une partie importante du temps de dynamique est utilisée pour affiner la construction du système et ceci avant même son étude. L'assemblage initial est souvent grossier, et il faut, pour stabiliser le système, simuler son mouvement en le chauffant progressivement jusqu'à la température de simulation désirée. Ensuite seulement, l'étude du système peut commencer.

Dans les paragraphes qui suivent, nous décrirons le modèle de dynamique moléculaire utilisé dans notre application. C'est un modèle opérationnel pour l'étude de structures biologiques complexes, il est comparable aux modèles existant dans les programmes traditionnellement utilisés en biologie. Nous nous sommes d'ailleurs largement inspirés des modèles utilisés dans CHARMM [16], XPLOR [23] et EGO [67] [3] pour extraire les fonctionnalités du modèle indispensable à l'étude du mouvement des protéines.

3.2 Principe

La dynamique moléculaire simule le mouvement des atomes en calculant la suite, au cours du temps, des positions des atomes d'un système donné. Dans ce modèle, les atomes sont considérés comme des points matériels assortis de données statiques : une masse, une charge électrique et un type (azote, carbone, hydrogène, oxygène, . . .). Et dépendant du temps, on associe également à l'atome une position dans l'espace et un vecteur vitesse. Le mouvement de ces atomes est régi par l'équation du mouvement de NEWTON ou par l'équation du mouvement de LANGEVIN (cf. 3.6) suivant le type de simulation que l'on désire.

Le calcul du mouvement d'un ensemble de particules est analogue à l'étude du mouvement d'un ensemble d'étoiles ou de planètes. C'est un problème à N-corps. Dans un tel problème, dès qu'il y a plus de deux particules dans le système, on ne connaît pas de solution explicite pour décrire le mouvement de ces particules. Pour calculer le mouvement des atomes d'un système, on effectue itérativement une intégration numérique des équations du mouvement. Et ainsi, de proche en proche, à partir des positions et des vitesses initiales des atomes, on calcule la suite au cours du temps des positions et des vitesses des atomes du système. Les positions et les vitesses initiales des atomes sont des données du problème.

Pour intégrer l'équation du mouvement, il faut, à chaque pas d'intégration, calculer les forces qui s'exercent entre les atomes du système. Dans le modèle classique utilisé traditionnellement en biologie, on considère trois types de forces :

- **les forces des interactions non-liées classiques.** Il s'agit des forces électrostatiques de COULOMB et des forces de VAN DER WAALS. Ces forces s'exercent sur chacune des paires d'atomes du système ;
- **les forces des interactions géométriques.** Dérivées de plusieurs fonctions d'énergie empiriques, ces forces s'exercent entre les atomes d'une même molécule. Elles rendent compte des déformations géométriques entre ses atomes ;

– **les forces de contraintes (*restraint*).** Elles sont principalement utilisées pour maintenir les atomes dans l'espace de simulation et éventuellement simuler l'influence du milieu extérieur sur le système étudié.

En résumé, l'algorithme principal de dynamique moléculaire est constitué d'une grande itération représentant les pas d'intégrations dans le temps (cf. figure 3.1). A chaque itération, on calcule l'ensemble des forces qui s'exercent entre les atomes du système. Puis on intègre l'équation du mouvement pour trouver les nouvelles positions des atomes [16] [90] [23].

```
pour i ← 1 à nbIter faire
   calcul des forces non-liées ;
   calcul des forces géométriques ;
   calcul des forces de contraintes ;
   intégration du mouvement ;
fin pour
```

FIG. 3.1: Schéma de l'itération fondamentale de dynamique moléculaire.

Dans la suite de ce chapitre, on va donner plus précisément les forces inter-atomiques et les équations du mouvement utilisées en dynamique moléculaire classique. On discutera aussi de la position initiale des atomes et des contraintes au bord du système, nécessaire pour la simulation d'un ensemble fini d'atomes.

3.3 Forces d'interactions non-liées

Les forces d'interactions non liées, contrairement aux forces géométriques s'exercent entre chaque paire d'atomes du système. Il existe deux types de forces d'interactions non liées : les forces d'interactions de VAN DER WAALS, dérivées d'un potentiel de LENNARD JONES ; et les forces d'interactions électrostatiques dérivées d'un potentiel de COULOMB.

3.3.1 Énergie de VAN DER WAALS

L'énergie de VAN DER WAALS, représentée par un potentiel de LENNARD JONES, se compose d'un potentiel d'attraction et d'un potentiel de répulsion. Voici la forme de l'énergie de VAN DER WAALS entre deux atomes i et j :

$$E_{\mathrm{VdW}(i,j)} = \left(\frac{A_{(i,j)}}{r_{(i,j)}^{12}} \right) - \left(\frac{B_{(i,j)}}{r_{(i,j)}^{6}} \right) \tag{3.1}$$

$r_{(i,j)}$ est la distance entre les deux atomes. Les constantes $A_{(i,j)}$ et $B_{(i,j)}$ dépendent des types des deux atomes de l'interaction.

Voici les forces qui s'exercent sur les atomes correspondants :

$$\overrightarrow{F_i} = -\overrightarrow{\nabla_i} E_{\mathrm{VdW}(i,j)} = \left(\frac{12A_{(i,j)}}{r^{14}_{(i,j)}} - \frac{6B_{(i,j)}}{r^{8}_{(i,j)}} \right) \begin{bmatrix} x_i - x_j \\ y_i - y_j \\ z_i - z_j \end{bmatrix}$$

$$\overrightarrow{F_j} = -\overrightarrow{\nabla_j} E_{\mathrm{VdW}(i,j)} = \left(\frac{12A_{(i,j)}}{r^{14}_{(i,j)}} - \frac{6B_{(i,j)}}{r^{8}_{(i,j)}} \right) \begin{bmatrix} x_j - x_i \\ y_j - y_i \\ z_j - z_i \end{bmatrix}$$

Paramètres de VAN DER WAALS

Les paramètres de VAN DER WAALS ($A_{(i,j)}$ et $B_{(i,j)}$) sont construits de la façon suivante :

1. pour chacun des types d'atomes I, une table donne les paramètres de VAN DER WAALS ϵ_I et $\sigma_I = R_{\mathrm{VdW}(I)} * 2^{-1/6}$. Ici $R_{\mathrm{VdW}(I)}$ est le rayon de VAN DER WAALS de l'atome de type I ;
2. puis en utilisant les équations de combinaison de LORENTZ-BERTHELOT, pour chaque paire de types d'atomes (I, J) on construit $\epsilon_{(I,J)}$ et $\sigma_{(I,J)}$:

$$\epsilon_{(I,J)} = \sqrt{\epsilon_I \epsilon_J} \quad , \qquad \sigma_{(I,J)} = (\sigma_I + \sigma_J)/2$$

3. puis pour chaque paire de types d'atomes (I, J), on construit les paramètres de VAN DER WAALS $A_{(I,J)}$ et $B_{(I,J)}$:

$$A_{(I,J)} = 4\,\sigma^{12}_{(I,J)}\,\epsilon_{(I,J)} \quad , \qquad B_{(I,J)} = 4\,\sigma^{6}_{(I,J)}\,\epsilon_{(I,J)}$$

L'utilisateur peut également donner explicitement les valeurs de certains des paramètres $A_{(I,J)}$ et $B_{(I,J)}$. Cette possibilité de définir explicitement les paramètres de VAN DER WAALS entre certains types d'atomes est principalement utilisée pour les interactions entre les atomes d'hydrogène et d'oxygène appartenant aux molécules d'eau.

3.3.2 Énergie de COULOMB

L'énergie électrostatique de COULOMB et les forces dérivées de cette énergie dépendent des charges Q_i et Q_j des atomes i et j respectivement. Voici la forme de l'énergie de COULOMB entre deux atomes i et j :

$$E_{\mathrm{Coul}(i,j)} = \left(\frac{Q_i Q_j}{4\pi\epsilon_0 r_{(i,j)}} \right) \tag{3.2}$$

$r_{(i,j)}$ est la distance entre les deux atomes.

Voici les forces qui s'exercent sur les atomes correspondants :

$$\overrightarrow{F_i} = -\overrightarrow{\nabla_i} E_{\mathrm{Coul}(i,j)} = \left(\frac{Q_i Q_j}{4\pi\epsilon_0 r^{3}_{(i,j)}} \right) \begin{bmatrix} x_i - x_j \\ y_i - y_j \\ z_i - z_j \end{bmatrix}$$

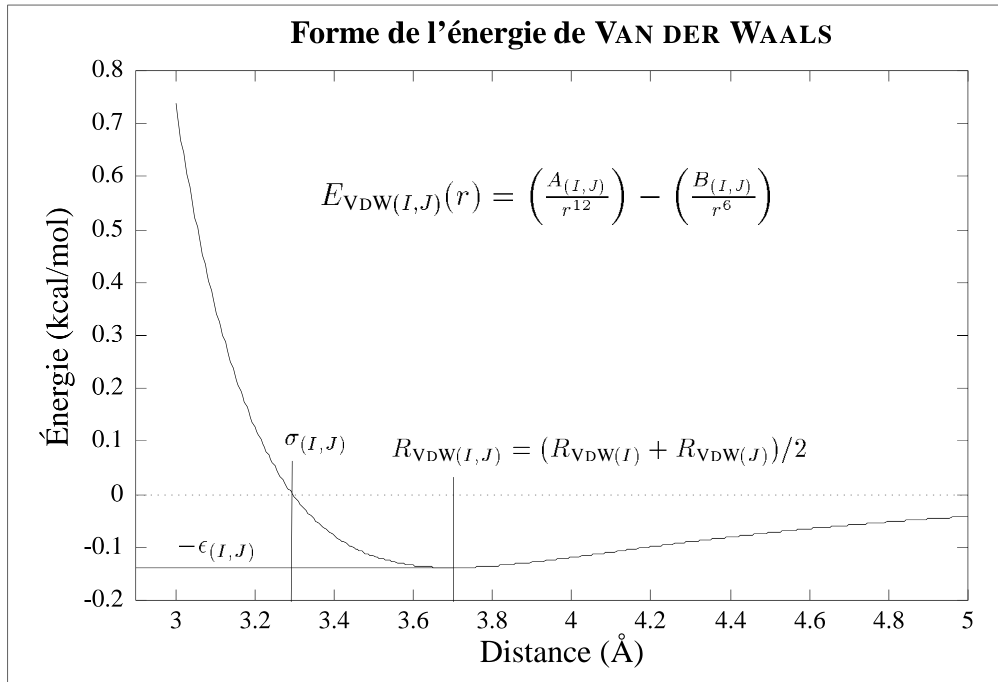


FIG. 3.2: Forme de l'énergie de Van der Waals entre deux atomes de type I et J.

$$\overrightarrow{F_j} = -\overrightarrow{\nabla_j} E_{\mathrm{Coul}(i,j)} = \left(\frac{Q_i Q_j}{4\pi\epsilon_0 r_{(i,j)}^3}\right) \begin{bmatrix} x_j - x_i \\ y_j - y_i \\ z_j - z_i \end{bmatrix}$$

3.4 Forces d'interactions géométriques

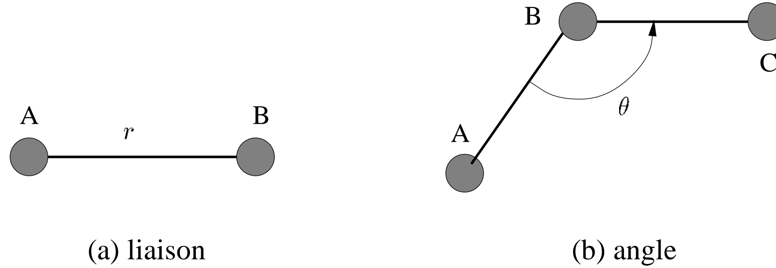
Les forces d'interactions géométriques décrivent les propriétés géométriques des molécules. Elles rendent compte de la déformation des liaisons, des angles et des angles dièdres entre les atomes d'une même molécule. Il s'agit respectivement de termes à deux, trois et quatre corps, qui dérivent de potentiels d'énergie empirique. Les constantes de ces potentiels d'énergie dépendent des types des atomes des interactions. Ce sont des données du problème. En effet, une table fournit ces paramètres pour chacun des types de liaisons, d'angles et d'angles dièdres possibles.

3.4.1 Énergie des liaisons

L'énergie de liaison inter-atomique est approchée par un potentiel harmonique qui s'exerce entre les deux atomes d'une liaison covalente. S'il existe une liaison covalente entre les atomes A et B, voici la forme du potentiel d'énergie de cette liaison :

$$E_{\mathrm{L}(A,B)} = K_r \left(r_{(A,B)} - r_0\right)^2$$

Dans cette équation, $r_{(A,B)}$ est la distance entre les atomes A et B. K_r est une constante d'énergie. Et r_0 est la distance moyenne de la liaison covalente entre les deux atomes (cf. figure 3.3 (a)).



(a) liaison (b) angle

FIG. 3.3: (a) Liaison covalente. (b) Angle entre deux liaisons covalentes.

3.4.2 Énergie des angles

L'énergie d'angle est approchée par un potentiel harmonique qui prend en compte les déformations angulaires entre deux liaisons covalentes issues d'un même atome. Par exemple, s'il existe une liaison entre les atomes A et B, et s'il existe une autre liaison covalente entre les atomes B et C, la forme du potentiel d'énergie angulaire qui s'exerce sur les atomes de l'angle $\widehat{ABC}$ est :

$$E_{\mathrm{A}(A,B,C)} = K_\theta \left(\theta_{(A,B,C)} - \theta_0\right)^2$$

$\theta_{(A,B,C)}$ est l'angle entre les atomes A, B et C. K_θ est une constante d'énergie. Et θ_0 est l'angle moyen entre les deux liaisons covalentes (cf. figure 3.3 (b)).

3.4.3 Énergie des angles dièdres

Il existe deux types de potentiels, suivant que la rotation autour de l'angle dièdre est possible ou non. Pour ces deux potentiels, il est nécessaire de calculer l'angle dièdre $\varphi_{(A,B,C,D)}$ entre les atomes A, B, C et D (cf. figure 3.4 et 3.5). Par la suite, pour simplifier, on note $\varphi = \varphi_{(A,B,C,D)}$. Ainsi, on calcule l'angle φ suivant $\overrightarrow{BC}$ en posant :

$$\begin{aligned}
\vec{u} &= \overrightarrow{BA} \wedge \overrightarrow{CB} \\
\vec{v} &= \overrightarrow{CB} \wedge \overrightarrow{DC} \\
\vec{w} &= \overrightarrow{CB} \wedge \vec{u} = \overrightarrow{CB} \wedge \left(\overrightarrow{BA} \wedge \overrightarrow{CB}\right)
\end{aligned}$$

Ici $\overrightarrow{U} \wedge \overrightarrow{V}$ désigne le produit vectoriel des deux vecteurs $\overrightarrow{U}$ et $\overrightarrow{V}$. Ainsi :

$$\cos(\varphi) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \cdot \|\vec{v}\|} \qquad \text{et} \qquad \sin(\varphi) = -\frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}$$

On calcule l'angle dièdre avec la meilleure précision possible :

$$\varphi = \left\{ \begin{array}{ll} \left\{ \begin{array}{lll} \pi - \arcsin(\sin(\varphi)) & \text{si} & \cos(\varphi) \leq 0 \\ \arcsin(\sin(\varphi)) & \text{si} & \cos(\varphi) > 0 \end{array} \right\} & \text{si} \quad |\sin(\varphi)| \leq 0.7 \\ \left\{ \begin{array}{lll} -\arccos(\cos(\varphi)) & \text{si} & \sin(\varphi) \leq 0 \\ \arccos(\cos(\varphi)) & \text{si} & \sin(\varphi) > 0 \end{array} \right\} & \text{si} \quad |\sin(\varphi)| > 0.7 \end{array} \right.$$

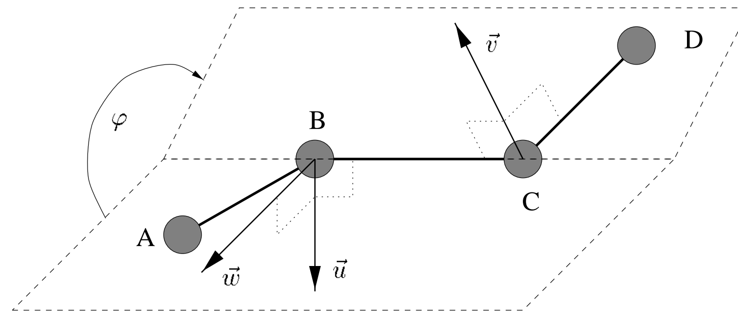


FIG. 3.4: Angle dièdre propre.

Énergie des angles dièdres propres

L'énergie d'angle dièdre propre est une rotation possible autour de l'angle dièdre. C'est une somme de potentiels de torsion qui s'exerce sur les quatre atomes A, B, C et D de l'angle dièdre $\varphi_{(A,B,C,D)}$.

$$E_{\mathrm{Dp}(A,B,C,D)} = \sum_i K_{\varphi_i} \left(1 + \cos\left(n_{\varphi_i}\varphi_{(A,B,C,D)} + \delta_{\varphi_i}\right)\right)$$

K_{φ_i} est une constante d'énergie, n_{φ_i} est la période de rotation, et δ_{φ_i} est un décalage de phase initial.

Énergie des angles dièdres impropres

Pour les angles dièdres impropres, la rotation autour de l'axe (BC) n'est pas autorisée. Ce sont par exemple les angles dièdres appartenant à un cycle d'atomes dans une molécule. On utilise alors un potentiel d'oscillation harmonique autour d'une position d'équilibre. Il s'exerce sur les quatre atomes A, B, C et D de l'angle dièdre $\varphi_{(A,B,C,D)}$.

$$E_{\mathrm{Di}(A,B,C,D)} = K_{\varphi} \left(\varphi_{(A,B,C,D)} - \varphi_0\right)^2$$

K_{φ} est une constante d'énergie, et φ_0 l'angle dièdre impropre moyen.

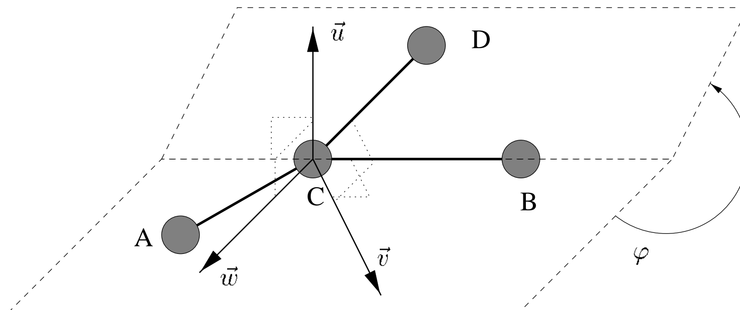


FIG. 3.5: Angle dièdre impropre.

3.4.4 Listes d'exclusion

Pour affiner le modèle il est nécessaire de supprimer le calcul des forces de COULOMB et de VAN DER WAALS entre les paires d'atomes appartenant à une même interaction géométrique. On appelle l'ensemble des couples d'atomes, pour lesquels on ne calcule pas les interactions non-liées, la liste d'exclusion.

En effet, on peut considérer le système étudié comme un graphe. Les atomes sont les sommets du graphe et les liaisons covalentes les arêtes du graphe. De cette façon, chaque composante connexe du graphe représente une molécule. On élimine le calcul des interactions de COULOMB et de VAN DER WAALS entre les atomes pour lesquels il existe un chemin dans le graphe inférieur à une longueur donnée. En général, on élimine complètement les interactions non-liées pour lesquelles il existe un chemin de longueur inférieure ou égale à deux.

Une option supplémentaire permet une exclusion partielle des interactions non-liées entre les atomes pour lesquels il existe un chemin de longueur 3 (exclusion de longueur 3). En effet, pour ces interactions, on utilise des paramètres de VAN DER WAALS ($A_{(i,j)}$ et $B_{(i,j)}$) spéciaux, et on ne tient compte que d'une partie des énergies de COULOMB (généralement 40 % [23]).

En plus des exclusions obtenues d'après les liaisons covalentes entre les atomes, il peut être nécessaire d'exclure des paires d'interactions particulières. Par exemple exclure le calcul des forces de COULOMB et de VAN DER WAALS entre atomes d'un cycle atomique. Ces exclusions sont explicitement données par l'utilisateur.

3.5 Potentiel des forces de contraintes (*restraint*)

3.5.1 Potentiel des contraintes harmoniques

Ces contraintes permettent de fixer la position de certains atomes en limitant leurs mouvements autour d'une position fixe donnée. On note (x_i^0, y_i^0, z_i^0) cette position pour l'atome i. Et on note (x_i, y_i, z_i) la position courante de l'atome i. On définit ainsi l'énergie

de contrainte harmonique par :

$$E_{\mathrm{H}(i)} = K_{\mathrm{H}(i)}((x_i - x_i^0)^2 + (y_i - y_i^0)^2 + (z_i - z_i^0)^2)$$

$K_{\mathrm{H}(i)}$ est une constante de contrainte harmonique qui dépend de l'atome i.

3.5.2 Contraintes aux bords du système

Pour maintenir les atomes dans la zone de simulation, et éviter par exemple l'évaporation des molécules d'eau, on ajoute un potentiel spécial qui s'exerce sur les atomes au bord du système :

$$E_{\mathrm{CB}(i)} = K_{\mathrm{CB}} d^2 (d^2 - P_{\mathrm{CB}})$$

Ici, K_{CB} et P_{CB} sont les paramètres du potentiel de contraintes aux bords. Généralement [17] on prend $K_{\mathrm{CB}} = 0,2\ kcal/mol/\text{Å}^4$ et $P_{\mathrm{CB}} = 2,25\ \text{Å}^2$. Les systèmes moléculaires étudiés peuvent avoir plusieurs formes. Ainsi, par exemple, pour un système sphérique centré de rayon R, on prend $d = \|\vec{r}_i\| - R + \sqrt{P_{\mathrm{CB}}/2}$. Ici $\|\vec{r}_i\| = \sqrt{x_i^2 + y_i^2 + z_i^2}$ est le module du vecteur position de l'atome i.

3.6 Équations du mouvement

Deux équations du mouvement sont utilisées en dynamique moléculaire classique. La dynamique de NEWTON utilise l'équation classique du mouvement. La dynamique de LANGEVIN utilise une version modifiée de l'équation du mouvement de NEWTON. En particulier, la dynamique de LANGEVIN permet de simuler l'influence du milieu extérieur au système. Pour intégrer les équations du mouvement, nous avons utilisé les schémas d'intégration traditionnels des programmes de dynamique moléculaire [90] [16]. Le choix du pas d'intégration dépend de la fréquence de vibration de l'atome le plus léger dans le système. S'il existe des atomes d'hydrogène, le pas d'intégration est très petit (environ $0,25 \cdot 10^{-15}\ s$ [16] [23]). Pour remédier à cet inconvénient, plusieurs méthodes existent. Il est par exemple possible d'ajouter des contraintes sur le mouvement des atomes les plus légers pour augmenter le pas d'intégration [129]. Une autre solution, consiste à modifier le schéma d'intégration en utilisant des termes plus précis pour les atomes d'hydrogène [79].

3.6.1 Positions et vitesses initiales

Nous l'avons déjà écrit dans l'introduction de ce chapitre, la dynamique moléculaire calcule la suite au cours du temps des positions des atomes à partir de leur position et de leur vitesse initiale. La position initiale des atomes est une donnée du problème. Et on calcule les vitesses initiales des atomes par une distribution de MAXWELL. C'est une distribution aléatoire des vitesses dont le module dépend de la température initiale d'étude du système.

Initialement, les composantes du vecteur vitesse $v_i(0)$ de l'atome i suivent la loi normale centrée d'écart-type : $\sqrt{\frac{K_b \mathtt{T}(0)}{m_i}}$.
Ici $\mathtt{T}(0)$ est la température initiale du système,

$$K_b = 1,380\ 662.10^{-23}\ J/K = 1,987\ 191.10^{-3}\ Kcal/K/mol$$

est la constante de BOLTZMANN, et m_i la masse de l'atome i.

3.6.2 Calcul de la température

Avant de calculer la température, on calcule l'énergie cinétique de l'ensemble des n atomes au temps t de la façon suivante :

$$E_{\text{cin}}(t) = \frac{1}{2} \sum_{i=0}^{n} m_i (v_i(t))^2$$

Ici m_i est la masse de l'atome i et $v_i(t)$ sa vitesse.
On calcule ensuite la température du système au temps t par :

$$\mathtt{T}(t) = \frac{2E_{\text{cin}}(t)}{3nK_b}$$

K_b est la constante de BOLTZMANN.

3.6.3 Dynamique de NEWTON

Le mouvement d'un ensemble de n atomes est régi par l'équation de NEWTON. Il s'agit d'un système non-linéaire d'équations différentielles ordinaires d'ordre deux.
L'équation du mouvement de NEWTON pour un ensemble de n atomes est :

$$\forall i = 1, \ldots, n \qquad m_i \left(\frac{\partial^2 r_i}{\partial t^2} \right)(t) = F_i(r_1(t), \ldots, r_n(t)) \tag{3.3}$$

m_i est la masse de l'atome i, et $r_i(t)$ sa position à l'instant t. $F_i(r_1(t), \ldots, r_n(t))$ est la somme des forces exercées sur l'atome i par les autres atomes du système au temps t. Les forces qui s'exercent sur l'atome i dérivent des potentiels d'énergie définis précédemment.

$$F_i = -\overrightarrow{\nabla_i} \left(\sum_j \left(E_{\text{VdW}(i,j)} + E_{\text{Coul}(i,j)} \right) + \sum_{\text{Liaisons}} E_{\text{L}} + \sum_{\text{Angles}} E_{\text{A}} + \sum_{\text{Dièdres}} E_{\text{D}} + E_{\text{H}(i)} + E_{\text{CB}(i)} \right)$$

Pour intégrer ce système d'équations différentielles, on utilise le schéma classique suivant [90] [16] :

$$\begin{cases} r_i(t+\delta t) & = & r_i(t) + v_i\left(t - \frac{\delta t}{2}\right)\delta t + \frac{\delta t^2}{m_i} F_i(r_1(t), \ldots, r_n(t)) \\ v_i\left(t + \frac{\delta t}{2}\right) & = & \frac{r_i(t+\delta t) - r_i(t)}{\delta t} \end{cases}$$

Ici, δt est l'intervalle d'intégration dans le temps, et $v_i(t)$ la vitesse de l'atome i à l'instant t.

Réajustement des vitesses

Pour contrôler la simulation il faut pouvoir modifier la température de la simulation. Pour cela, on agit directement sur les vitesses des atomes en les multipliant par un facteur fonction de la température-cible. On obtient un schéma d'intégration modifié :

$$\begin{cases} v_i^*\left(t-\frac{\delta t}{2}\right) &= \sqrt{\frac{\mathtt{T}_{\text{cible}}(t)}{\mathtt{T}(t)}}v_i\left(t-\frac{\delta t}{2}\right) \\ r_i(t+\delta t) &= r_i(t)+v_i^*\left(t-\frac{\delta t}{2}\right)\delta t+\frac{\delta t^2}{m_i}F_i(r_1(t),\ldots,r_n(t)) \\ v_i\left(t+\frac{\delta t}{2}\right) &= \frac{r_i(t+\delta t)-r_i(t)}{\delta t} \end{cases}$$

Ici $\mathtt{T}_{\text{cible}}(t)$ est la température-cible de simulation du système au temps t, et $\mathtt{T}(t)$ est la température du système à l'instant t.

3.6.4 Dynamique de LANGEVIN

Dans l'équation de la dynamique de LANGEVIN, on ajoute à l'équation classique du mouvement de NEWTON, un terme de friction et une force aléatoire pour simuler l'influence du milieu extérieur au système.

$$m_i\left(\frac{\partial^2 r_i}{\partial t^2}\right)(t)=F_i(t)+f_i(t)-m_i\beta_i v_i(t) \tag{3.4}$$

m_i est la masse de l'atome i et r_i sa position. F_i la somme des forces exercées par les autres atomes sur l'atome i. β_i est un terme de friction et $v_i(t)$ la vitesse de l'atome i au temps t. Et $f_i(t)$ est une force aléatoire dépendant de la température, dont les composantes suivent une loi normale centrée et d'écart-type : $\sqrt{\frac{2m_i\beta_i K_b \mathtt{T}_{\text{cible}}(t)}{\delta t}}$.
Ici $\mathtt{T}_{\text{cible}}(t)$ est la température-cible du système au temps t, K_b est la constante de BOLTZMANN, et δt est l'intervalle d'intégration.

Pour intégrer le système, on utilise le schéma suivant :

$$\begin{cases} r_i(t+\delta t) &= \left(1+\frac{\beta_i\delta t}{2}\right)^{-1}\left[2r_i(t)-r_i(t-\delta t)\left(1-\frac{\beta_i\delta t}{2}\right)+\frac{\delta t^2}{m_i}\left(F_i(t)+f_i(t)\right)\right] \\ v_i\left(t+\frac{\delta t}{2}\right) &= \frac{r_i(t+\delta t)-r_i(t)}{\delta t} \end{cases}$$

Pour démarrer l'intégration, lors du premier pas, on utilise le schéma d'intégration de l'équation de NEWTON.

3.7 Autres modèles de dynamique moléculaire

Il existe beaucoup de variantes du modèle classique de dynamique moléculaire. Les modèles utilisés dépendent des phénomènes étudiés. Par exemple, pour étudier des gaz, on se contente d'un modèle plus simple où toutes les particules sont identiques et ou seules les forces de VAN DER WAALS s'exercent. Autre exemple, pour étudier la diffusion d'éléments dopants dans un substrat de silicium, on utilise seulement les forces non-liées de COULOMB et de VAN DER WAALS dont les paramètres dépendent des types des atomes.

Le modèle utilisé en biologie est très complet. En effet, la modélisation des propriétés géométriques des protéines est primordiale. Pour renforcer les propriétés géométriques des protéines, des études sont menées pour considérer la géométrie des molécules comme des contraintes fixes sur les positions des atomes ; et plus comme des oscillations autour d'une position d'équilibre. Dans ce modèle, appelé dynamique dans l'espace des angles (propriété utilisée en dynamique des corps rigides), le problème change de nature. Il ne s'agit plus de calculer le mouvement de points matériels. Mais il faut calculer le mouvement de volumes structurés, représentant des ensembles d'atomes fixés entre eux et appartenant à des portions de molécules. Le lecteur intéressé trouvera une approche très mathématique dans [111] [22] et une approche plus pratique analogue à l'étude du mouvement de bras de robot dans [95] [78] [86]. Les équations qui résultent de ces modèles sont particulièrement ardues et ce type d'approche ne fait pas l'unanimité. De plus, comme nous le verrons dans le chapitre suivant, même si les forces géométriques ajoutent quelques contraintes, la difficulté principale du modèle de dynamique moléculaire réside dans le calcul efficace des forces non-liées.

Chapitre 4

Approximations pour la dynamique moléculaire

Dans ce chapitre, nous décrirons quelques méthodes pour approcher le calcul des forces non-liées. En particulier nous montrerons l'intérêt de la méthode du rayon de coupure. Nous terminerons en donnant quatres exemples de structure biologique qui nous ont servi à tester nos algorithmes.

4.1 Introduction

Comme, nous l'avons écrit au chapitre précédent, calculer le mouvement des atomes d'un système nécessite une intégration numérique des équations du mouvement. Il est donc nécessaire de calculer les forces d'interactions à chaque pas de temps. Or le calcul exact de l'ensemble des forces d'interaction non-liées est une opération coûteuse en terme de temps de calcul. En effet la complexité algorithmique de ces calculs est proportionnelle au carré du nombre d'atomes. Dans la pratique, pour des petits systèmes (moins de 1 000 atomes) le calcul complet des forces d'interactions non-liées représente plus de 90 % du temps de calcul d'une itération complète. Même avec un ordinateur parallèle, le calcul exact des forces non-liées n'est pas réaliste pour les gros systèmes. C'est pourquoi, pour réduire la complexité algorithmique de ce calcul, les programmes de dynamique moléculaire utilisent des approximations des forces d'interactions non-liées. Le calcul des autres forces et l'intégration de l'équation du mouvement ont des complexités algorithmiques proportionnelles au nombre d'atomes du système. Il existe essentiellement trois techniques pour réduire le volume de calcul des forces non-liées :

- **la méthode du rayon de coupure** (*cut off*) est très largement employée en dynamique moléculaire classique [16]. Cette méthode consiste à calculer seulement les interactions non-liées entre les atomes se trouvant à une distance inférieure à une distance donnée. La complexité algorithmique du calcul des forces des interactions non-liées devient proportionnelle au nombre des particules du système. Nous détaillerons cette méthode dans la suite de ce chapitre ;

- **le calcul des forces suivant la distance** (*distance classes*). TILDESLEY propose dans [121] de modifier le schéma d'intégration. Ceci a été repris plus tard par GRUBMÜLER et HELLER dans [67] [3]. La méthode se base sur une observation simple : les forces à longue distance varient moins rapidement que les forces à courte distance. La méthode consiste donc à calculer les forces qui s'exercent sur un atome avec une fréquence qui dépend de la distance entre l'atome considéré et les atomes qui composent la force. Plus précisément, comme le montre la figure 4.1 (a) sur une particule, on découpe l'espace en sphères de taille croissante. Puis, pour chacun des atomes, on calcule à chaque pas de temps les forces issues des atomes dont la distance est inférieure à R fixé. Un pas sur deux, on calcule les forces issues des atomes compris entre R et $2R$. On calcule un pas sur quatre les forces issues des atomes dont la distance est comprise entre $2R$ et $3R$, etc...

  Cette méthode réduit le volume des calculs des forces non-liées. Mais sa complexité reste proportionnelle au carré du nombre d'atomes. Le calcul des forces suivant la distance est donc difficilement envisageable pour des systèmes de grande taille ;

- **la méthode des moments multipolaires** (*FMA*) de BARNES et HUT [4] et la méthode rapide des moments multipolaires de GREENGARD et ROKHLIN [66] [64] [65] sont utilisées en dynamique moléculaire pour tenir compte des forces d'interaction électrostatique à longue distance [45] [10]. Ces deux méthodes utilisent un principe similaire. À la base, pour calculer les forces de VAN DER WAALS et les forces électrostatiques à courte distance, on utilise la méthode du rayon de coupure. La méthode des moments multipolaires est utilisée seulement pour le calcul des interactions électrostatiques au-delà du rayon de coupure.

  On commence par diviser l'espace de simulation avec des ensembles emboîtés d'atomes de forme cubique et de taille croissante, comme des boîtes gigognes (cf. figure 4.1 (b)). Il y a ensuite deux phases :

  - une phase ascendante, dans laquelle, pour chacune des boîtes, on calcule un potentiel électrostatique sous la forme d'une série multipolaire et en fonction des atomes de la boîte. Puis, pour les boîtes plus grosses, on construit la série multipolaire associée à chacune des boîtes en fonction des séries des boîtes de taille juste inférieure. Au final, on obtient une série multipolaire associée à chacune des boîtes du système. La série correspond au potentiel électrostatique exercé par l'ensemble des atomes inclus dans la boîte associée ;

  - une phase descendante, dans laquelle, on combine par niveau les potentiels électrostatiques de façon à obtenir, pour chacun des ensembles, le potentiel exercé par les atomes des autres boîtes sur les atomes de l'ensemble. La figure 4.1 (b) montre également les boîtes que l'on utilise pour trouver le potentiel qui s'exerce sur les atomes de la boîte C lors de cette deuxième phase.

  Les travaux sur ces méthodes sont nombreux et le lecteur intéressé par plus de détails peut aussi consulter les papiers de BOARD et ELLIOTT [51] [52] ou les travaux de WINDEMUTH [131] [132]. La complexité algorithmique de la méthode rapide des multipoles est proportionnelle au nombre d'atomes du système. De plus,

cette méthode permet d'estimer l'ordre de grandeur de l'erreur d'approximation des forces électrostatiques [50].

Des ensembles emboîtés d'atomes de forme non structurée sont également employés pour obtenir de meilleures approximations des forces électrostatiques à longue distance [102]. Cette méthode est notamment combinée avec la technique précédente de calcul des forces en fonction de la distance.

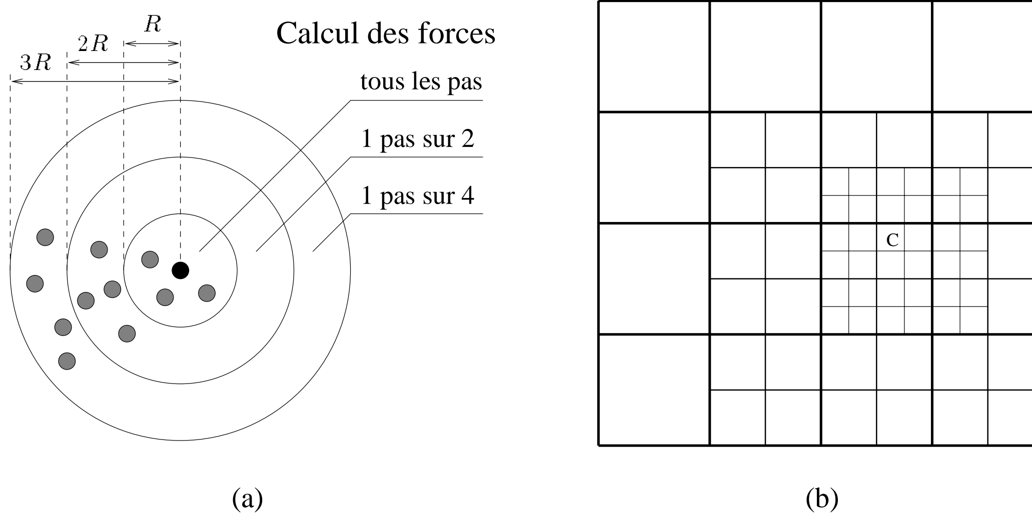


FIG. 4.1: (a) Calcul des forces suivant la distance. (b) Découpage de l'espace de simulation pour l'algorithme des moments multipolaires.

4.2 Méthode du rayon de coupure

La méthode du rayon de coupure (*cut off*), généralement employée dans les codes de dynamique moléculaire, calcule seulement les interactions non-liées entre les atomes voisins. C'est-à-dire, seulement les interactions entre atomes se trouvant à une distance inférieure à une distance donnée, appelé rayon de coupure. En effet, le module des forces d'interaction de VAN DER WAALS diminue rapidement quand la distance augmente (cf. équation 3.1). De même, si les charges électriques sont réparties de façon homogène dans le système, les forces de Coulomb s'annulent à longue distance (cf. équation 3.2). Il est ainsi possible de négliger les forces d'interactions non-liées au-delà du rayon de coupure. Pour un rayon de coupure fixé R_{coupure}, on définit les ensembles V_i^t des atomes voisins d'un atome i au temps t par :

$$V_i^t = \left\{ j \text{ tel que } \|r_i(t) - r_j(t)\|_2 < R_{\text{coupure}} \right\}$$

$r_i(t)$ et $r_j(t)$ sont respectivement la position de l'atome i et de l'atome j au temps t. L'équation des forces d'interactions non-liées qui s'exercent sur un atome i devient :

$$F_i(r_{V_i^t}(t)) = -\overrightarrow{\nabla_i}\left(\sum_{j\in V_i^t}\left(E_{\text{VdW}(i,j)} + E_{\text{Coul}(i,j)}\right)\right)$$

L'algorithme du rayon de coupure réduit considérablement le volume de calcul des interactions non-liées. En effet dans le cas d'une répartition homogène des atomes, et pour un rayon de coupure fixé, le nombre de voisins de chaque atome est majoré par une constante dépendant du volume de la sphère de coupure. C'est-à-dire qu'il existe une constante K telle que le cardinal de tous les ensembles de voisins soit majoré par le volume de la sphère de coupure :

$$\forall i \quad \#(V_i^t) \leq K \cdot R^3_{\text{coupure}}$$

En pratique, la répartition des atomes n'est pas homogène, mais la densité locale de particules est majorée. On peut donc quand même trouver une constante pour majorer le nombre des voisins de chacune des particules en fonction de la taille de la sphère de coupure. La complexité pour le calcul des forces d'interactions non-liées devient ainsi proportionnelle au nombre d'atomes. Ou plus précisément, pour n particules la complexité est $O(n \cdot R^3_{\text{coupure}})$.

4.2.1 Approximation des forces non-liées

Il ne faut pas perdre de vue que nous étudions un système dynamique. Une troncature brutale des forces et des énergies des interactions non-liées au bord de la sphère de coupure crée des variations brutales des forces et des énergies. Surtout lorsqu'un atome sort ou rentre dans la sphère de coupure. Pour éviter ces variations trop brutales, on utilise une fonction spéciale au bord de la sphère de coupure. Cette fonction lisse le potentiel d'énergie des intégrations non-liées au bord de la sphère de coupure. Ces fonctions de coupure ne font pas l'unanimité aussi nous avons laissé le choix à l'utilisateur. Dans la suite, on note $\texttt{sw}(x)$ cette fonction spéciale et $\texttt{dsw}(x)$ sa dérivée par rapport à x. L'énergie de VAN DER WAALS devient :

$$E_{\text{VdW}(i,j)} = \left(\frac{A_{(i,j)}}{r^{12}_{(i,j)}} - \frac{B_{(i,j)}}{r^{6}_{(i,j)}}\right)\texttt{sw}(r^2_{(i,j)})$$

Et les forces qui s'exercent sur les atomes i et j correspondants deviennent :

$$\overrightarrow{F_i} = \left(\left(\frac{12A_{(i,j)}}{r^{14}_{(i,j)}} - \frac{6B_{(i,j)}}{r^{8}_{(i,j)}}\right)\texttt{sw}(r^2_{(i,j)}) - 2\left(\frac{A_{(i,j)}}{r^{12}_{(i,j)}} - \frac{B_{(i,j)}}{r^{6}_{(i,j)}}\right)\texttt{dsw}(r^2_{(i,j)})\right)\begin{bmatrix} x_i - x_j \\ y_i - y_j \\ z_i - z_j \end{bmatrix}$$

$$\overrightarrow{F_j} = \left(\left(\frac{12A_{(i,j)}}{r^{14}_{(i,j)}} - \frac{6B_{(i,j)}}{r^{8}_{(i,j)}}\right)\texttt{sw}(r^2_{(i,j)}) - 2\left(\frac{A_{(i,j)}}{r^{12}_{(i,j)}} - \frac{B_{(i,j)}}{r^{6}_{(i,j)}}\right)\texttt{dsw}(r^2_{(i,j)})\right)\begin{bmatrix} x_j - x_i \\ y_j - y_i \\ z_j - z_i \end{bmatrix}$$

De même l'énergie de COULOMB devient :

$$E_{\text{Coul}(i,j)} = \left(\frac{Q_i Q_j}{4\pi\epsilon_0 r_{(i,j)}}\right) \texttt{sw}(r^2_{(i,j)})$$

Et les forces qui s'exercent sur les atomes i et j correspondants deviennent :

$$\overrightarrow{F_i} = \left(\left(\frac{Q_i Q_j}{4\pi\epsilon_0 r^3_{(i,j)}}\right) \texttt{sw}(r^2_{(i,j)}) - 2\left(\frac{Q_i Q_j}{4\pi\epsilon_0 r_{(i,j)}}\right) \texttt{dsw}(r^2_{(i,j)})\right) \begin{bmatrix} x_i - x_j \\ y_i - y_j \\ z_i - z_j \end{bmatrix}$$

$$\overrightarrow{F_j} = \left(\left(\frac{Q_i Q_j}{4\pi\epsilon_0 r^3_{(i,j)}}\right) \texttt{sw}(r^2_{(i,j)}) - 2\left(\frac{Q_i Q_j}{4\pi\epsilon_0 r_{(i,j)}}\right) \texttt{dsw}(r^2_{(i,j)})\right) \begin{bmatrix} x_j - x_i \\ y_j - y_i \\ z_j - z_i \end{bmatrix}$$

Dans la suite, on définit le rayon de coupure extérieur par $R_{\text{Ext}} = R_{\text{coupure}}$ et le rayon de coupure intérieur R_{Int} tel que $0 < R_{\text{Int}} < R_{\text{Ext}}$. Comme les programmes de dynamique moléculaire classiques, notre programme offre, en plus de la troncature brutale, un choix de trois fonctions spéciales de coupure :

– la fonction de coupure du programme XPLOR :

$$\texttt{sw}(x) = \begin{cases} 1 & \text{si} \quad x \leq R^2_{\text{Int}} \\ \frac{(R^2_{\text{Ext}} - x)(2x + R^2_{\text{Ext}} - 3R^2_{\text{Int}})}{R^2_{\text{Ext}} - R^2_{\text{Int}}} & \text{si} \quad R^2_{\text{Int}} < x < R^2_{\text{Ext}} \\ 0 & \text{si} \quad R^2_{\text{Ext}} \leq x \end{cases}$$

– la fonction de coupure du programme CHARMM :

$$\texttt{sw}(x) = \begin{cases} 1 & \text{si} \quad x \leq R^2_{\text{Int}} \\ \frac{(R^2_{\text{Ext}} - x)^2(2x + R^2_{\text{Ext}} - 3R^2_{\text{Int}})}{(R^2_{\text{Ext}} - R^2_{\text{Int}})^3} & \text{si} \quad R^2_{\text{Int}} < x < R^2_{\text{Ext}} \\ 0 & \text{si} \quad R^2_{\text{Ext}} \leq x \end{cases}$$

– ou bien la fonction de coupure *shift* suivante :

$$\texttt{sw}(x) = \begin{cases} \left(1 - \frac{x}{R^2_{\text{Ext}}}\right)^2 & \text{si} \quad x < R^2_{\text{Ext}} \\ 0 & \text{si} \quad R^2_{\text{Ext}} \leq x \end{cases}$$

Le lecteur intéressé peut trouver une évaluation de la précision de ces fonctions dans [46] et [120].

4.2.2 Construction de la liste des voisins

La construction pour chacun des atomes de la liste des voisins n'est pas un algorithme trivial. Un algorithme naïf qui compare les atomes deux à deux pour construire la liste des voisins est en $O(n^2)$.

De plus comme la position des atomes évolue au cours du temps, il est nécessaire de reconstruire ces listes de voisins régulièrement. Une solution simple permet d'éviter de reconstruire systématiquement la liste des voisins. On ajoute une distance de tolérance D_{tol} au rayon de coupure. Ainsi, on construit au temps t, une liste étendue des voisins pour chacun des atomes.

$$\forall i \quad \tilde{V}_i^t = \left\{ j \ \text{ tel que } \ \|r_i(t) - r_j(t)\|_2 < R_{\text{coupure}} + 2D_{\text{tol}} \right\}$$

On peut facilement montrer que $\forall\ \delta \in \mathbb{R}$:

$$(\forall i \quad \|r_i(t) - r_i(t+\delta)\|_2 \leq D_{\text{tol}}) \Rightarrow (\forall i \quad V_i^{t+\delta} \subset \tilde{V}_i^t)$$

Ainsi la liste des voisins normale $V_i^{t+\delta}$ de l'atome i reste valide tant que les atomes ne se déplacent pas au-delà de la distance de tolérance. La figure 4.2 (a) montre comment cela marche pour deux atomes. Cette astuce est très efficace en pratique, car dans la plupart des systèmes étudiés l'essentiel du mouvement des atomes est une oscillation autour d'une position d'équilibre. En effet, au lieu de construire la liste des interactions à chacune des itérations, cette astuce permet, pour une distance de tolérance de 1 Å, de construire la liste des interactions seulement tout les 40 à 50 itérations, cela dépend aussi du système étudié et du pas d'intégration dans le temps [16] [23].

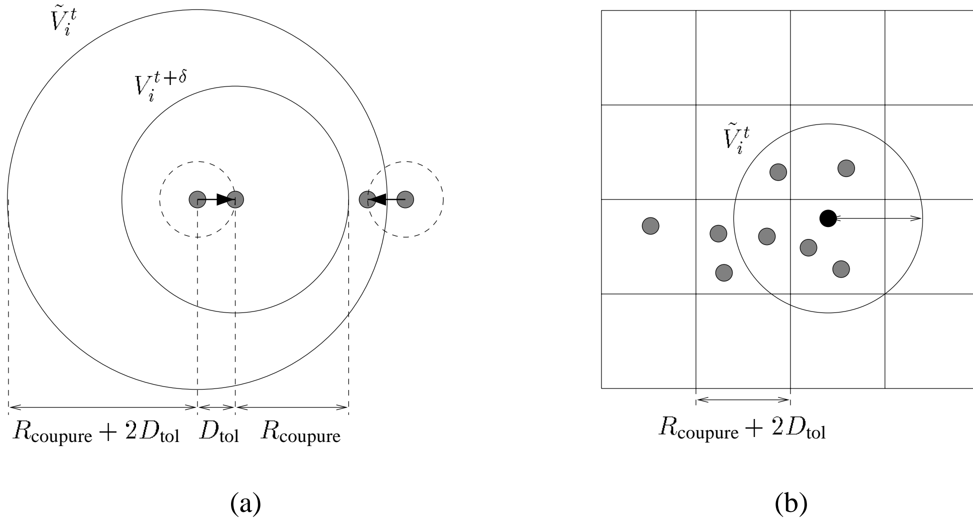


FIG. 4.2: (a) Utilité de la distance de tolérance. (b) Construction de la liste des voisins. Les voisins d'un atome se trouvent dans sa boîte et dans les boîtes voisines.

Pour construire la liste des voisins de chacun des atomes rapidement :

1. on découpe l'espace de simulation en boîtes cubiques de taille $\tilde{R}_{\text{coupure}} = R_{\text{coupure}} + 2D_{\text{tol}}$;
2. on détermine la boîte de chacun des atomes en fonction de sa position dans l'espace ;
3. on trie les atomes suivant le numéro de la boîte à laquelle ils appartiennent ;

4. On termine par la construction de la liste des paires des interactions non-liées à calculer, ce sont les paires d'atomes dont la distance est inférieure au rayon de coupure $\tilde{R}_{\text{coupure}}$. Pour cela on utilise le fait que les atomes voisins (au sens du rayon de coupure) d'un atome donné se trouvent dans la boîte de l'atome et dans ses 26 boîtes voisines. La figure 4.2 (b) montre une telle construction en deux dimensions.

La complexité de l'algorithme est fonction de l'opération de tri c'est-à-dire en $O(n \log n)$. Les programmes de dynamique moléculaire reconstruisent régulièrement les listes de voisins.

Nous utiliserons un algorithme légèrement différent. Nous placerons les atomes directement dans leur boîte au niveau de l'étape (1). On réalise ainsi une opération analogue à un tri par tas de complexité $O(n)$. Cette variante n'est généralement pas utilisée par les programmes de dynamique moléculaire car elle consomme plus de mémoire.

4.3 Conclusion

L'algorithme du rayon de coupure réduit considérablement le volume des calculs. Mais la taille des systèmes étudiés, plus de 30 000 atomes, et le nombre d'itérations nécessaire à une bonne simulation, de l'ordre de 100 000 itérations, nécessitent de grosses puissances de calcul. L'algorithme du rayon de coupure ne permet pas de modéliser tous les phénomènes électrostatiques. Son utilisation ne fait pas l'unanimité mais il est très largement employé dans les programmes traditionnellement utilisés en séquentiel pour l'étude des protéines [16] [23]. Il est également considéré comme suffisant pour la plupart des phénomènes étudiés par le mouvement des protéines [46] [120]. De plus, les autres techniques d'approximation, l'algorithme des moments multipolaires compris, utilisent l'algorithme du rayon de coupure pour le calcul des forces non-liées à courte distance. C'est cette dernière raison qui nous a incité à étudier une parallélisation efficace de la méthode du rayon de coupure.

4.4 Description des quatres structures test

Par la suite, pour tester et comparer les performances de nos algorithmes parallèles, nous avons utilisé quatres systèmes réalistes de tailles différentes :

- **une petite structure de 3 625 atomes,** assemblage hétérogène comprenant : une protéine de 3 544 atomes et 27 molécules d'eau ;
- **une structure moyenne de 11 615 atomes,** assemblage formant une membrane et comprenant : un canal ionique complètement fonctionnel (dioxolane-gramicidine), 1 ion potassium, 106 phospholipides DLPE pour former la membrane, et 2 281 molécules d'eau ;
- **une structure plus grosse de 35 349 atomes,** composée : d'un canal ionique complètement fonctionnel (dioxolane-gramicidine), de 200 phospholipides POPE pour former la membrane, et 8 210 molécules d'eau (cf. figure 4.3) ;

– **une structure de grande taille de 413 039 atomes,** composée d'une protéine de β-galactosidase de 65 240 atomes hydratés dans une sphère d'eau de 100 Å de rayon et comprenant 115 933 molécules d'eau. L'assemblage possède également 298 066 liaisons, 233 345 angles et 183 928 dièdres.

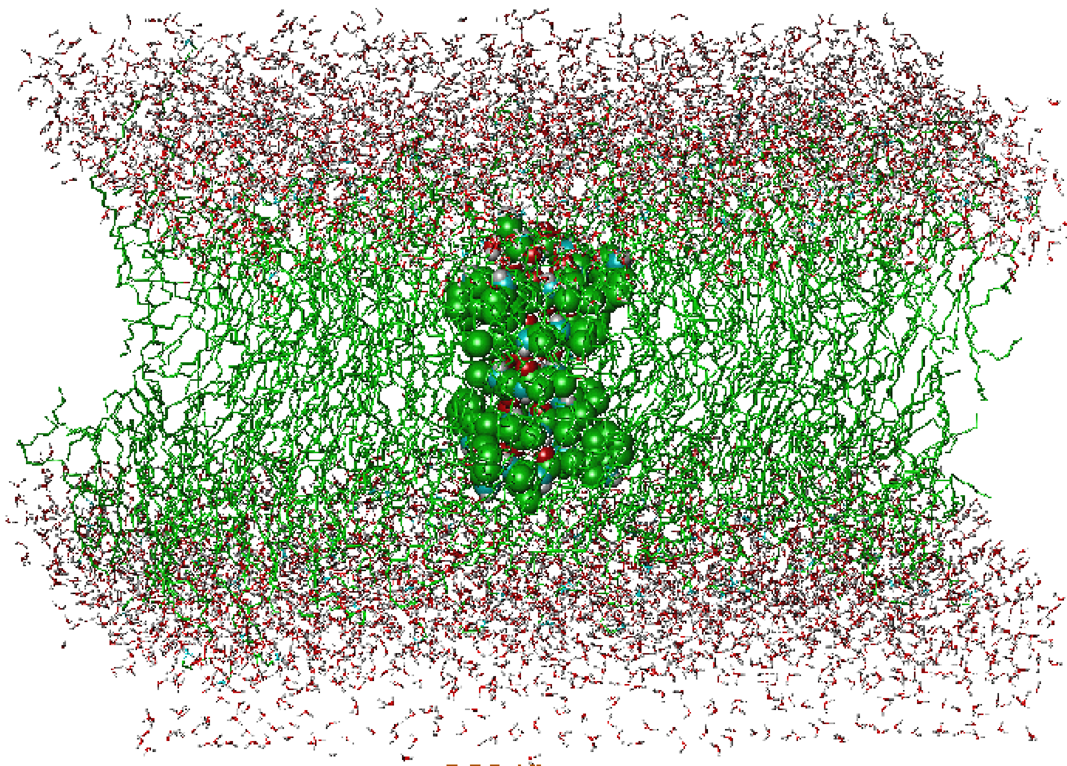


FIG. 4.3: Structure de 35 349 atomes d'un canal ionique dans une membrane de phospholipides POPE.

Pour chacune des structures, les données sont fournies sous la forme de trois fichiers dans le format utilisé par le programme XPLOR [23] :

– **les paramètres,** fichier qui comprend les listes des paramètres des forces géométriques pour tous les types de liaisons, d'angles et de dièdres possibles. Il contient également la liste des noms de chacun des types d'atomes présents dans la structure simulée, avec leurs paramètres de VAN DER WAALS associés ;
– **la structure.** Ce fichier contient :
  – la liste de chaque atome avec son type, sa masse et sa charge électrique ;
  – toutes les liaisons, tous les angles et tous les dièdres de la structure ;
  – la liste des paires d'atomes exclues par l'utilisateur du calcul des interactions non-liées.
– **les coordonnées,** fichier comprenant la position des atomes du système, puis éventuellement, les paramètres pour les contraintes harmoniques et le terme de friction β pour l'équation du mouvement de LANGEVIN.

Dans notre programme, les paramètres de simulation (le rayon de coupure, le temps d'intégration, la température de simulation, ...), le choix des fonctions de coupure pour l'approximation des forces non-liées et le type de dynamique (NEWTON ou LANGEVIN) peuvent être choisis facilement. Mais pour les mesures de performances, seul le rayon de coupure a une véritable influence [106]. Pour comparer facilement nos différentes options, nous avons choisi les paramètres de simulation qui correspondent à ceux qui sont généralement employés [16] [23]. Ainsi, sauf indications contraires, pour les trois premiers systèmes, nous avons utilisé les équations du mouvement de NEWTON, avec un réajustement de la température tous les 5 pas et un pas de temps (δt) de $0,25 \cdot 10^{-15}\ s$. Nous avons utilisé la fonction de *shift* pour les interactions électrostatiques de COULOMB, et la fonction de coupure de CHARMM pour les interactions de VAN DER WAALS. Nous avons également choisi un rayon de coupure extérieur ($R_{\text{Ext}} = R_{\text{coupure}}$) de 10 Å, un rayon de coupure intérieur (R_{Int}) de 9 Å, puis une distance de tolérance (D_{tol}) de 1 Å.

Pour la structure de 413 039 atomes, nous avons fait la simulation pour un rayon de coupure extérieur ($R_{\text{Ext}} = R_{\text{coupure}}$) de 10 Å, un rayon de coupure intérieur (R_{Int}) de 7 Å, puis une distance de tolérance (D_{tol}) de 1 Å, avec les mêmes fonctions de coupure que précédemment. Nous avons commencé avec un petit pas d'intégration de $0,1 \cdot 10^{-15}\ s$, pour équilibrer le système. Nous avons mené une première simulation de 300 pas pour chauffer progressivement la structure de 50 K à 300 K et avec un réajustement de la température tous les 5 pas. Puis nous avons mené une simulation de 700 pas à température constante (réajustement tout les 10 pas) pour bien équilibrer la structure. Soit au total, seulement pour équilibrer la structure une simulation de $100 \cdot 10^{-15}\ s$. C'est pendant cette deuxième simulation à température constante que nous avons fait nos mesures (cf. chapitre 6).

Nous avons ensuite mené une simulation longue de 4 fois 1400 pas avec un pas d'intégration de $0,25 \cdot 10^{-15}\ s$ et sans réajustement de la température pour vérifier la stabilité du système, mais avec un replacement des atomes sur les processeurs tous les 50 pas. Ce dernier test représente 35 heures de calcul sur 20 noeuds du SP1. Il n'est pas suffisant pour valider le programme numériquement, mais c'est un test pour vérifier la capacité de traiter des simulations longues sur des gros systèmes.

Troisième partie

Parallélisation

Chapitre 5

Parallélisation et répartition initiale des calculs

Après un rappel des principales méthodes de parallélisation utilisées en dynamique moléculaire, nous étudierons plus précisément la méthode de partition de l'espace de simulation. Cette méthode est théoriquement la plus prometteuse. Mais pour obtenir des bonnes performances lors de la simulation de protéines, nous montrons la nécessité d'obtenir un bon équilibre de charge. Enfin, après une analyse plus précise du problème, nous proposerons une étude comparative des différentes stratégies de placement des calculs et des données sur les processeurs au début de la simulation.

5.1 Méthodes usuelles de parallélisation

L'algorithme du rayon de coupure est irrégulier du point de vue du parallélisme. Les calculs à effectuer, pour obtenir les forces d'interactions non-liées d'un atome, dépendent de ses atomes voisins. Les calculs dépendent fortement de la position des atomes et ne sont donc pas indépendants de l'exécution du programme. Pour faciliter la compréhension, on peut exprimer l'intégration de l'équation du mouvement de NEWTON sous forme matricielle. On note $X(t) \in \mathbb{R}^{3n}$ le vecteur des positions des atomes au temps t, $V(t) \in \mathbb{R}^{3n}$ le vecteur des vitesses des atomes au temps t. Le schéma d'intégration s'écrit :

$$\left\{ \begin{array}{lcl} X(t+\delta t) & = & X(t) + \delta t \cdot V(t-\frac{\delta t}{2}) + \delta t^2 \cdot \mathcal{M} \cdot G(\ X(t)\ ) \cdot U \\ V(t+\frac{\delta t}{2}) & = & \frac{1}{\delta t} \cdot (\ X(t+\delta t) - X(t)\ ) \end{array} \right.$$

Ici $\mathcal{M} \in \mathbb{R}^{3n \times 3n}$ est une matrice diagonale contenant sur la diagonale l'inverse des masses des atomes répété trois fois. U est le vecteur unitaire de $\mathbb{R}^n$.

$$\mathcal{M} = \begin{bmatrix} 1/m_1 & 0 & 0 & & & & \\ 0 & 1/m_1 & 0 & & & & \\ 0 & 0 & 1/m_1 & & & 0 & \\ & & & \ddots & & & \\ & 0 & & & 1/m_n & 0 & 0 \\ & & & & 0 & 1/m_n & 0 \\ & & & & 0 & 0 & 1/m_n \end{bmatrix} \qquad U = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Et $G(\ X(t)\ ) \in \mathbb{R}^{3n \times n}$ est la matrice des forces. Ainsi, on peut définir le vecteur $f_{(i,j)}$ de la force exercée par l'atome j sur l'atome i à partir des éléments de la matrice $G(\ X(t)\ )$ par :

$$\forall\, i,j = 1,\cdots,n \quad f_{i,j} = \begin{bmatrix} g_{((3i-2),j)} \\ g_{((3i-1),j)} \\ g_{(3i,j)} \end{bmatrix}$$

D'après la troisième loi de NEWTON, les forces entre les atomes s'opposent, on a donc la propriété suivante sur les éléments de la matrice $G(\ X(t)\ )$:

$$\forall\, i,j = 1,\cdots,n \quad f_{(i,j)} = -f_{(j,i)} \quad - \quad \begin{bmatrix} g_{((3i-2),j)} \\ g_{((3i-1),j)} \\ g_{(3i,j)} \end{bmatrix} = - \begin{bmatrix} g_{((3j-2),i)} \\ g_{((3j-1),i)} \\ g_{(3j,i)} \end{bmatrix}$$

Si on calcule toutes les forces non-liées du système, G est une matrice pleine. Mais si on utilise un rayon de coupure, G est une matrice creuse sans structure particulière. L'essentiel du calcul dans une itération de dynamique moléculaire consiste à construire la matrice G des forces en fonction de la position des atomes du système. En pratique, on ne construit pas explicitement la matrice des forces, on calcule directement le vecteur F des forces qui s'exercent sur les atomes, en sommant au fur et à mesure les coefficients des lignes de G.

$$F \in \mathbb{R}^{3n}, \qquad F = G(\ X(t)\ ) \cdot U$$

Dans la littérature, il existe essentiellement trois familles d'algorithmes parallèles pour la méthode du rayon de coupure [106] qui correspondent chacune à une façon différente de répartir le calcul des coefficients non nuls de la matrice G :

- **la distribution des atomes** (*atom-decomposition*), cette méthode répartit les atomes et le calcul des forces correspondantes sur les processeurs ;
- **la distribution des forces** (*force-decomposition*) répartit le calcul des forces non-liées sur les processeurs ;
- **la décomposition spatiale** (*spatial-decomposition* ou *link-cell*) attribue une partie de l'espace de simulation à chacun des processeurs.

Dans les paragraphes qui suivent, nous présenterons plus en détail le principe des trois méthodes traditionnelles. Nous rappellerons les évaluations empiriques du volume de calculs et du volume de communications en fonction de la taille du problème [106]. Nous nous concentrerons sur l'itération de dynamique moléculaire, en faisant abstraction de la construction et de la modification de la liste des voisins des atomes. En effet, nous avons vu au chapitre précédent (cf. 4.2.2) que l'introduction d'une distance de tolérance au mouvement des atomes permet de reconstruire la liste des voisins seulement toutes les 40 ou 50 itérations.

5.1.1 Parallélisation par distribution des atomes

La première famille d'algorithmes, appelée algorithme de distribution des atomes (*atom-decomposition*), découpe l'ensemble des atomes sur les processeurs, sans contrainte sur la position des atomes. Chacun des processeurs calcule les forces seulement pour ces atomes. Mais pour calculer ces forces, les noeuds de calcul nécessitent la position de tous les atomes. Comme le montre la figure 5.1, cela revient à effectuer un découpage par ligne de la matrice G des forces.

Dans l'algorithme ci-dessous, on note $X_i \in \mathbb{R}^{3n/p}$ la partie du vecteur des positions placée sur le processeur i. On note $F_{loc(i)} \in \mathbb{R}^{3n}$ le vecteur des forces calculées localement sur le processeur i. Et on note $F_i \in \mathbb{R}^{3n/p}$ la partie du vecteur des forces placée sur le processeur i. F_i contient la somme des forces qui s'exercent sur les atomes de X_i.

$$F = \sum_{i=1}^{p} F_{loc(i)} = \begin{bmatrix} F_1 \\ \vdots \\ F_p \end{bmatrix} \qquad X = \begin{bmatrix} X_1 \\ \vdots \\ X_p \end{bmatrix}$$

Pour p processeurs et un système de n atomes, l'itération sur le processeur i s'écrit :

1	***Diffuser*** *X_i à tous les processeurs, résultat : X*	$O(n)$
2	***Calculer*** *les forces locales, résultat : $F_{loc(i)}$*	$O(\frac{n}{p})$
3	***Regrouper et sommer*** *les vecteurs de forces de tous les processeurs, résultat : F_i*	$O(n)$
4	***Intégrer*** *l'équation du mouvement pour les atomes locaux, résultat : X_i*	$O(\frac{n}{p})$

Dans l'étape 2, on tient compte de l'antisymétrie des forces, et on calcule globalement une seule fois les forces qui s'exercent entre deux atomes. Ainsi après l'étape 2, le vecteur des forces locales $F_{loc(i)}$ n'est pas complet. Il faut donc communiquer avec les autres processeurs et sommer les forces calculées sur les autres processeurs pour avoir le vecteur des forces complet. Au total, la complexité en temps de calcul est en $O(\frac{n}{p})$ et la complexité des deux phases de communication est en $O(n)$.

Cet algorithme est facile à mettre en oeuvre à partir d'un programme séquentiel et est de ce fait très utilisé [63] [90]. Cette méthode est également employée pour les ordinateurs à mémoire partagée [96] ou avec des librairies de mémoire virtuellement partagée [76] [77]. Mais elle nécessite, à chaque pas de temps, l'échange de toutes les positions des

atomes (étape 1) et le regroupement de toutes les forces qui s'exercent sur ces atomes (étape 3). La phase de communication qui en résulte est du type échange total. C'est une communication très coûteuse et peu extensible.

Il est possible de supprimer l'une ou l'autre des deux phases de communication en introduisant de la redondance dans les calculs des positions des atomes ou dans les calculs des forces :

– on peut éviter la diffusion des coordonnées des atomes (étape 1). Pour cela, on communique tout le vecteur des forces à l'étape 3, et on calcule à l'étape 4, sur chacun des processeurs, la nouvelle position de tous les atomes du système. Globalement, on calcule la nouvelle position des atomes p fois, une fois sur chacun des processeurs;

– il est possible d'éviter le regroupement et la somme des forces de l'étape 3 en ne tenant pas compte de l'antisymétrie des forces. Dans l'étape 2, globalement, on calcule les forces entre deux atomes deux fois. Ainsi, après l'étape 2, on obtient directement le vecteur F_i complet des forces qui s'exercent sur les atomes du processeur i. Les étapes 1 et 4 ne changent pas dans cette dernière variante.

Ces deux variantes ne modifient pas la complexité de communication de l'algorithme, elles jouent seulement sur l'importance relative des différentes phases de l'algorithme.

Un autre inconvénient de cet algorithme, est la nécessité de stocker l'ensemble des données sur les noeuds. Ceci limite la taille des problèmes qu'il est possible de traiter. Pour remédier à ce problème, plusieurs articles [71] proposent de faire circuler les coordonnées des atomes sur un anneau virtuel, au fur et à mesure des besoins du calcul des forces. De même, ils proposent de faire circuler les forces dans le sens contraire au fur et à mesure de leur calcul. Ceci évite effectivement de stocker la structure complète sur tous les noeuds. Mais la phase de communication est encore plus coûteuse. C'est un algorithme d'échange total sur un anneau virtuel, il n'est donc pas optimal sur la majorité des réseaux de communications actuels. En pratique cette variante est surtout utilisée pour le calcul complet des interactions non-liées où la complexité du calcul (en $O(\frac{n^2}{p})$) est supérieure au temps de communication.

Pour les structures réelles, un déséquilibre de charge peut apparaître. Un nombre d'atomes identique par processeur n'impliquant pas un nombre identique d'interactions non-liées à calculer. Suivant la densité locale de particules, les atomes peuvent avoir plus ou moins de voisins dans la sphère de coupure. La solution la plus courante [63] pour ajuster la charge d'une itération à l'autre, consiste à modifier la part d'atomes traités par chacun des processeurs. Ainsi, on modifie aussi le nombre de forces à calculer associées aux atomes de chacun des processeurs.

Malgré sa simplicité, nous n'avons pas retenu cet algorithme, car il n'est pas extensible. La taille des problèmes qu'il peut traiter efficacement est limitée par le temps des phases de communication et par la taille de la mémoire des noeuds de la machine.

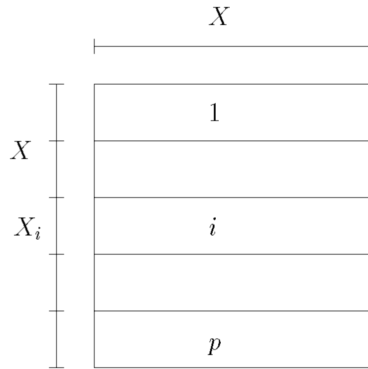


FIG. 5.1: Découpage en lignes de la matrice des forces pour les algorithmes qui distribuent les atomes.

5.1.2 Parallélisation par distribution des forces

La deuxième famille d'algorithmes est appelée algorithmes de distribution des forces (*force-decomposition*). Elle est basée sur un algorithme de calcul complet des interactions non-liées qui réalise une distribution par blocs de la matrice des forces G à calculer. La figure 5.2 montre un tel découpage.

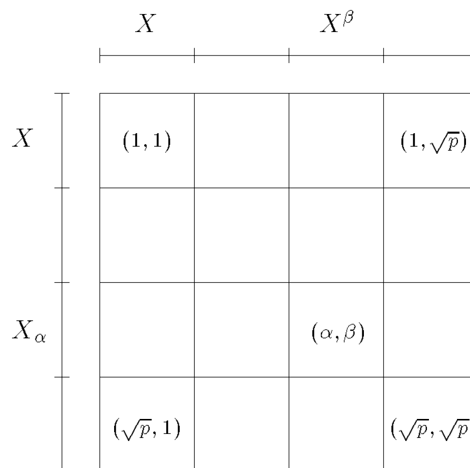


FIG. 5.2: Découpage en blocs de la matrice des forces pour les algorithmes qui distribuent les forces.

Pour simplifier la description de l'algorithme, on suppose que p, le nombre de processeurs, est un carré. Dans l'algorithme ci-dessous, on note X_α^β la partie du vecteur des positions des atomes placée sur le processeur (α, β). On note X_α la partie du vecteur des positions traitée par les processeurs de la ligne α, et X^β la partie du vecteur des positions

traitée par les processeurs de la colonne β.

$$X_\alpha \in \mathbb{R}^{(3n/\sqrt{p})}\ , \quad X = \begin{bmatrix} X_1 \\ \vdots \\ X_{\sqrt{p}} \end{bmatrix}\ , \qquad X_\alpha^\beta \in \mathbb{R}^{(3n/p)}\ , \quad X_\alpha = \begin{bmatrix} X_\alpha^1 \\ \vdots \\ X_\alpha^{\sqrt{p}} \end{bmatrix}$$

On note également F_α^β le vecteur des forces exercées par les atomes de X^β sur les atomes de X_α. Le calcul de F_α^β se fait sur le processeur (α, β). Enfin, F_α est la somme des forces qui s'exercent sur les atomes de X_α. L'itération sur le processeur (α, β) s'écrit :

1	***Diffuser*** *X_α^β à tous les processeurs de la ligne, résultat : X_α*	$O(\frac{n}{\sqrt{p}})$
2	***Diffuser*** *X_α^β à tous les processeurs de la colonne, résultat : X^β*	$O(\frac{n}{\sqrt{p}})$
3	***Calculer*** *les forces locales correspondant aux interactions entre les atomes de X_α et de X^β, résultat : F_α^β*	$O(\frac{n}{p})$
4	***Regrouper et sommer*** *les vecteurs des forces F_α^β sur la ligne, résultat : $F_\alpha = \sum_{i=1}^{\sqrt{p}} F_\alpha^i$*	$O(\frac{n}{\sqrt{p}})$
5	***Intégrer*** *l'équation du mouvement pour les atomes locaux, résultat : X_i*	$O(\frac{n}{p})$

L'étape 3 est en $O(\frac{n}{p})$ car on calcule seulement les forces entre les atomes voisins. Au total, le coût de calcul est en $O(\frac{n}{p})$ et le temps de communication est en $O(\frac{n}{\sqrt{p}})$. L'algorithme présenté ici ne tient pas compte de l'antisymétrie des forces, mais il existe une variante de même complexité qui en tient compte. Le lecteur intéressé par plus de détails pourra lire une description plus précise de cet algorithme dans [106] ou [73].

Dans cette méthode, pour que la charge soit équilibrée, en particulier lors de la phase 3, il faut que les coefficients non nuls de G soient distribués de façon homogène. Pour assurer cette propriété, il suffit de mélanger aléatoirement les atomes au début de la simulation. Ainsi, on assure statistiquement que les coefficients non nuls de G sont répartis de façon homogène. En revanche, pour calculer efficacement les forces géométriques, il faut que les atomes de ces interactions soient placés sur le même processeur. Ceci est contradictoire avec un bon équilibre de charge.

La distribution par blocs de la matrice des forces permet de diminuer la complexité algorithmique du coût des communications par rapport aux algorithmes décrits dans le paragraphe précédent. Mais dans cet algorithme la mauvaise localité des données nécessaire à un bon équilibre de charge est contradictoire avec une bonne localité des données nécessaire aux calculs des forces géométriques. Il n'est donc pas applicable à notre modèle.

5.1.3 Parallélisation par partition de l'espace de simulation

La dernière famille d'algorithmes est basée sur un découpage géométrique du système à simuler (*link-cell*), puis à une distribution de ce découpage de l'espace sur les processeurs (*spatial-decomposition*) [36] [84] [92] [74]. Chacun des processeurs s'occupe

d'une région donnée de l'espace de simulation. Contrairement aux méthodes précédentes, les atomes ne sont pas liés à un processeur, mais peuvent changer de processeur au cours de la simulation.

L'algorithme de découpage géométrique ne construit pas l'ensemble des voisins de chacun des atomes. Mais, comme lors de la construction des listes de voisins, il divise l'espace de simulation en boîtes cubiques, dont la taille dépend du rayon de coupure. Ainsi implicitement, les atomes voisins d'un atome donné se trouvent dans la boîte de l'atome considéré et dans ses 26 boîtes voisines [5]. La figure 5.3 montre une vue en deux dimensions d'un tel découpage de l'espace. Notons qu'il n'y a que huit boîtes voisines en deux dimensions.

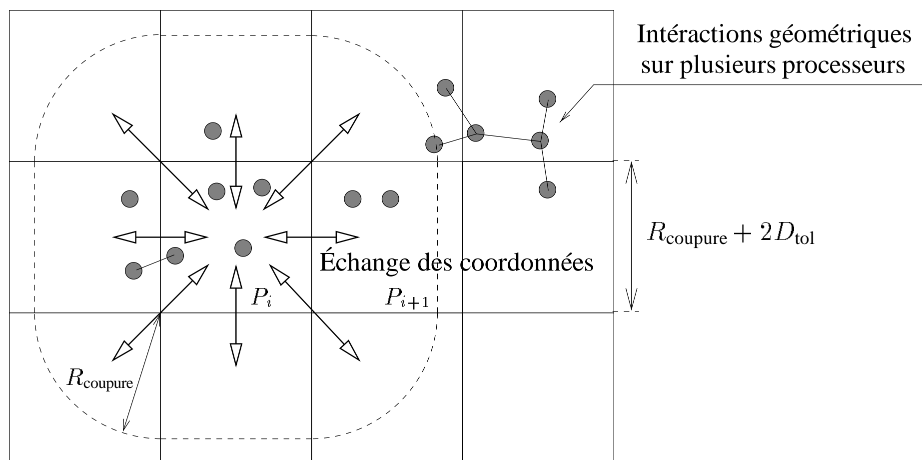


FIG. 5.3: Découpage spatial de l'espace de simulation.

Si on place une boîte par processeur, chacun des processeurs échange la position des atomes de sa boîte avec les 26 processeurs possédant les boîtes voisines. Après cette communication, chacun des processeurs possède les coordonnées des atomes de sa boîte et les coordonnées des atomes des boîtes voisines. Il peut ainsi calculer les forces d'interactions non-liées et les forces d'interactions géométriques qui s'exercent sur les atomes de sa boîte. Pour un ensemble de n atomes, et p processeurs, et si on place une seule boîte par processeur, l'algorithme s'écrit :

1	***Échanger** les coordonnées des atomes avec les processeurs contenant les boîtes voisines*	$O(\frac{n}{p})$
2	***Calculer** les forces qui s'exercent sur les atomes de la boîte locale*	$O(\frac{n}{p})$
3	***Intégrer** l'équation du mouvement pour les atomes de la boîte locale*	$O(\frac{n}{p})$

Comme nous le verrons plus tard, il est possible ici aussi d'utiliser l'antisymétrie des forces pour réduire le coût de calcul des forces d'interactions. Il suffit d'ajouter une phase de communication entre l'étape 2 et l'étape 3. Dans cette nouvelle étape, il faut regrouper et sommer les forces qui s'exercent entre les atomes des boîtes voisines.

Beaucoup d'articles ont montré l'efficacité de cet algorithme pour un ensemble d'atomes répartis de façon homogène [106] [5]. En effet, dans un tel cas, et si on place une boîte par processeur, le coût de calcul et de communication est en $O(n/p)$. De tous les algorithmes possibles il est le moins communicant. De plus, il distribue les atomes sur les processeurs et évite ainsi de stocker toute la structure sur un processeur. Il est donc le plus adapté pour simuler le mouvement de systèmes comportant beaucoup d'atomes.

Un premier essai pour préciser les hypothèses

Nous avons fait un premier essai pour cerner le comportement de cet algorithme de façon à connaître l'importance relative des différents paramètres comme la part du temps de calcul des interactions ou la part du déséquilibre de charge. Notre essai a porté sur la petite structure de 3 625 atomes décrite au paragraphe 4.4. Nous avons placé une boîte par processeur et nous avons utilisé un rayon de coupure de 15 Å. Pour un tel système, le temps d'exécution moyen d'une itération sur un noeud du SP1 est de 82,12 secondes. Pour le même système, le temps d'exécution moyen d'une itération sur 24 processeurs est de 9,7 secondes. On obtient une efficacité très médiocre de 35,3 %.

Mais, en étudiant plus précisément les structures réelles, on remarque que le nombre d'atomes par boîte est très variable (cf. figure 5.4). Il existe deux raisons à ce phénomène : la première raison est liée à la forme du système étudié. Le système n'est pas forcément un parallélépipède, ainsi les boîtes au bord du système ne sont pas pleines. La deuxième raison est liée à la nature du système étudié. Le nombre de particules par unité de volume n'est pas constant dans le système. En effet, par exemple il y a plus de particules par unité de volume dans une région où il y a des molécules d'eau que dans une région où il y a la protéine. Le déséquilibre du nombre d'atomes par processeur qui en résulte entraîne un déséquilibre de la charge de calcul encore plus important. En effet, le coût de calcul sur un processeur est essentiellement fonction du nombre d'interactions non-liées calculées. Si on place une boîte par processeur, on peut estimer le nombre d'interactions m_i à calculer sur le processeur i en fonction du nombre d'atomes de la boîte placée sur le processeur i, et du nombre d'atomes des boîtes voisines placées sur d'autres processeurs.

$$m_i = O\left(n_i.\left(n_i + \sum_{j\in V_i} n_j\right)\right)$$

Dans cette formule, V_i est l'ensemble des boîtes voisines de la boîte i au sens du rayon de coupure. La figure 5.4 montre le déséquilibre de charge pour la structure de 3 625 atomes. Dans ce graphique, on mesure également le temps moyen de calcul par itération sur chacun des processeurs, toujours en plaçant une boîte par processeur.

Axes de recherches

Ce premier programme nous a permis de préciser les hypothèses pour la suite du travail. Nous avons montré que dans les systèmes réels, les atomes ne sont pas répartis de façon homogène ce qui induit un fort déséquilibre de la charge de calcul sur les processeurs. La recherche d'une bonne répartition des tâches de calcul est primordiale. Nous

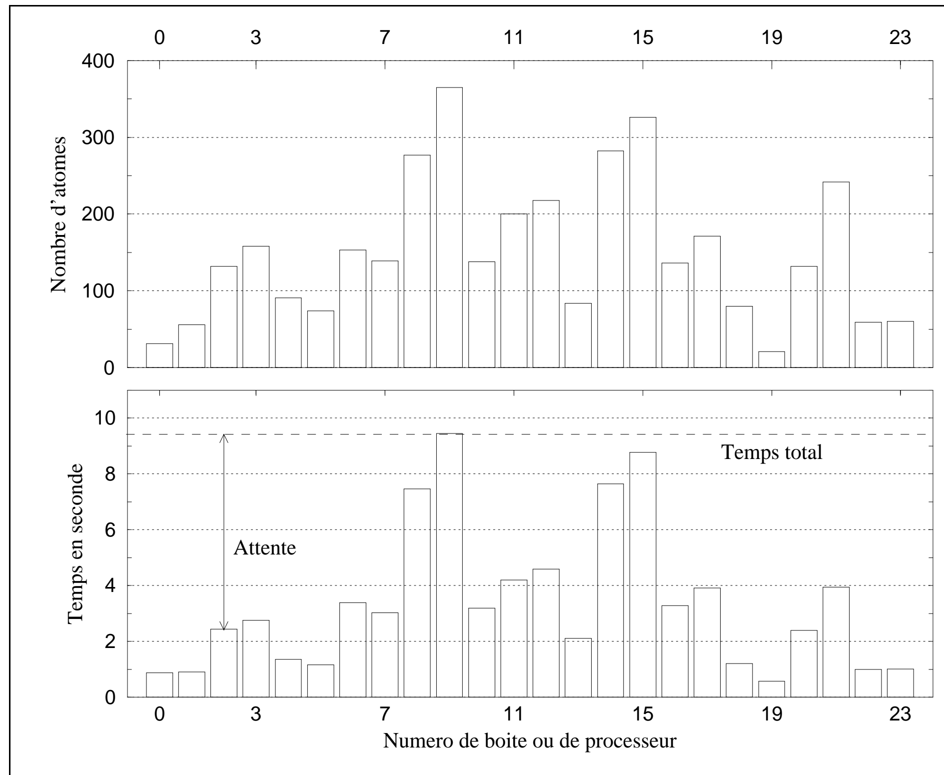


FIG. 5.4: Nombre d'atomes par boîte (ou processeur) pour un système réel de 3 625 atomes et temps de calcul moyen d'une itération sur les processeurs correspondants.

avons eu également la confirmation que la charge de calcul est essentiellement due au calcul des forces d'interactions non-liées. Répartir la charge signifie qu'il faut répartir le calcul des forces d'interactions non-liées sur les processeurs. Il faut également prendre en compte l'antisymétrie des forces d'interactions non-liées, afin de réduire les coûts de calcul.

Les études sur des structures plus importantes ont montré que le nombre de boîtes est bien supérieur au nombre des processeurs. De plus, dans cet algorithme, le nombre de processeurs utilisés est dépendant du nombre de boîtes, qui est lui-même dépendant du nombre d'atomes du système et de la taille du rayon de coupure. Le futur programme doit pouvoir utiliser un nombre quelconque de processeurs indépendamment des données du problème. En revanche le découpage géométrique de l'espace de simulation en boîtes cubiques de la taille du rayon de coupure semble bien adapté à l'algorithme du rayon de coupure. En particulier il évite la construction explicite de la liste des atomes voisins de chacun des atomes. Il nous est apparu judicieux de conserver ce découpage et de chercher un bon placement de ces boîtes et des tâches de calcul associées, plutôt que de chercher une partition géométrique quelconque de l'espace de simulation. Dans la suite, nous allons chercher un bon placement initial des données et des calculs des interactions non-liées, pour cet algorithme de partition géométrique de l'espace de simulation. Notons que [84] propose une approche similaire, mais le placement initial cherche à équilibrer le nombre

d'atomes par processeur. Pour obtenir des meilleurs résultats, nous chercherons avant tout à équilibrer les calculs sur les processeurs, et nous utiliserons un graphe de tâche plus précis.

5.2 Construction du graphe de tâches

Comme nous l'avons remarqué précédemment, le découpage spatial de l'espace de simulation sur les processeurs est l'approche la plus prometteuse pour traiter des systèmes de grande taille. Mais il faut trouver un bon placement des calculs et des données sur les processeurs. En particulier il faut équilibrer le calcul des forces non-liées. Le placement initial comporte plusieurs étapes :

1. Comme précédemment, on commence par diviser l'espace de simulation en boîtes cubiques de la taille du rayon de coupure. Le nombre de boîtes dépend de la taille du système. Puis on place chacun des atomes dans sa boîte. La position de l'atome permet directement de connaître la position de sa boîte ;
2. on construit le graphe de tâches associées à l'ensemble de boîtes ;
3. on évalue le coût de calcul des tâches. Comme nous le verrons par la suite, ceci consiste à évaluer le nombre de forces non liées à calculer dans chacune des tâches ;
4. on place ensuite les tâches sur les processeurs ;
5. on déduit du placement des tâches le placement des boîtes et de la structure ;
6. une fois connu le placement des boîtes, il faut identifier et construire les phases de communications.

Avant de présenter les différentes stratégies de placement que nous avons essayées, nous allons décrire le graphe de tâches à placer. Nous allons également introduire des contraintes sur le placement des tâches pour diminuer les communications et simplifier les structures de données. Nous terminerons par évaluer les coûts des tâches pour obtenir un bon équilibre de charge.

5.2.1 Identification des tâches

Dans une itération de dynamique moléculaire, nous avons identifié quatre types de tâches liées aux différentes phases de l'itération de dynamique moléculaire. Les tâches utilisent les données relatives aux boîtes du découpage géométrique de l'espace de simulation. Voici les quatre types de tâches :

– **la tâche** $T_{\text{ni}}(i)$ calcule les interactions non-liées de COULOMB et VAN DER WAALS entre les atomes de la boîte i (interactions non-liées internes à la boîte). Cette tâche nécessite les coordonnées des atomes de la boîte i, et retourne les forces qui s'exercent entre les atomes de la boîte ;
– **la tâche** $T_{\text{nv}}(i, j)$ calcule les interactions non-liées entre les atomes de deux boîtes i et j, voisines au sens du rayon de coupure (interactions non-liées entre des boîtes voisines). Cette tâche utilise les coordonnées des atomes des boîtes i et j. Elle retourne les forces des atomes de la boîte j qui s'exercent sur les atomes de la boîte i. Et par antisymétrie des forces, elle retourne les forces des atomes de la boîte i qui s'exercent sur les atomes de la boîte j ;
– **la tâche** $T_{\text{g}}(i)$ calcule les interactions géométriques relativement aux atomes d'une boîte i. C'est-à-dire les interactions géométriques ayant au moins un atome dans

cette boîte. Cette tâche a besoin de connaître les coordonnées des atomes appartenant à la boîte i et à ses boîtes voisines. En effet, comme le montre la figure 5.3, les interactions des liaisons, des angles et des dièdres peuvent faire intervenir des atomes appartenant à plusieurs boîtes voisines. La tâche retourne les forces géométriques qui s'exercent entre les atomes de ces boîtes ;

– **la tâche** $T_{\text{i}}(i)$ intègre le mouvement des atomes de la boîte i. Cette tâche nécessite l'ensemble des forces qui s'exercent sur les atomes de la boîte i, et retourne les nouvelles coordonnées des atomes de la boîte.

Dans cet algorithme on utilise l'antisymétrie des forces d'interactions non-liées. Ainsi, dans la suite, pour simplifier $T_{\text{nv}}(i, j)$ et $T_{\text{nv}}(j, i)$ désigneront la même tâche.

5.2.2 Contraintes de précédence

Après la définition des différentes tâches du problème, nous devons déterminer les dépendances de données entre ces tâches et construire le graphe de précédences de l'ensemble des tâches (cf. chapitre 1.3.1).

On construit ce graphe, d'une itération à l'autre. Avec la notation suivante $T_1 \ll T_2, T_3$ qui signifie que la tâche T_1 dépend des tâches T_2 et T_3. Cette relation définit un ordre partiel sur l'exécution des tâches. On note également $T_{\text{i-}}(i)$ la tâche d'intégration du mouvement pour les atomes de la boîte i à l'itération précédente. Et comme précédemment V_i est l'ensemble des boîtes voisines d'une boîte i au sens du rayon de coupure. Rappelons également que pour le moment, et compte tenu de la distance de tolérance (cf. 4.2.2), nous considérons que les atomes ne changent pas de boîtes d'une itération à l'autre. Voici le graphe de précédence d'une itération de dynamique moléculaire pour l'algorithme de découpages géométriques :

– le calcul des forces non-liées des atomes de la boîte i nécessite les positions courantes des atomes de la boîte. Ces positions sont calculées à l'itération précédente, par la tâche d'intégration du mouvement des atomes de la boîte. On a les précédences suivantes :

$$T_{\text{ni}}(i) \ll T_{\text{i-}}(i)$$

– le calcul des forces non-liées entre les atomes de deux boîtes voisines, au sens du rayon de coupure, nécessite les positions des atomes des deux boîtes :

$$T_{\text{nv}}(i, j) \ll T_{\text{i-}}(i), T_{\text{i-}}(j)$$

– le calcul des forces géométriques des atomes d'une boîte nécessite les coordonnées des atomes des boîtes voisines :

$$T_{\text{g}}(i) \ll T_{\text{i-}}(i), T_{\text{i-}}(V_i)$$

– l'intégration du mouvement des atomes d'une boîte nécessite la somme de toutes les forces qui s'exercent sur les atomes de la boîte. Donc les résultats de toutes

les tâches qui ont utilisé les coordonnées des atomes de la boîte pour calculer des forces. Ceci donne les précédences suivantes :

$$T_{\text{i}}(i) \ll T_{\text{ni}}(i), T_{\text{nv}}(i, V_i), T_{\text{g}}(i), T_{\text{g}}(V_i)$$

La figure 5.5 montre les dépendances relatives à une tâche de calcul des forces géométriques des atomes d'une boîte. Nous nous sommes limités à une vue en deux dimensions avec huit boîtes voisines pour faciliter la compréhension. De même, la figure 5.6 montre les dépendances relatives à quelques tâches de calcul des forces non-liées entre boîtes voisines.

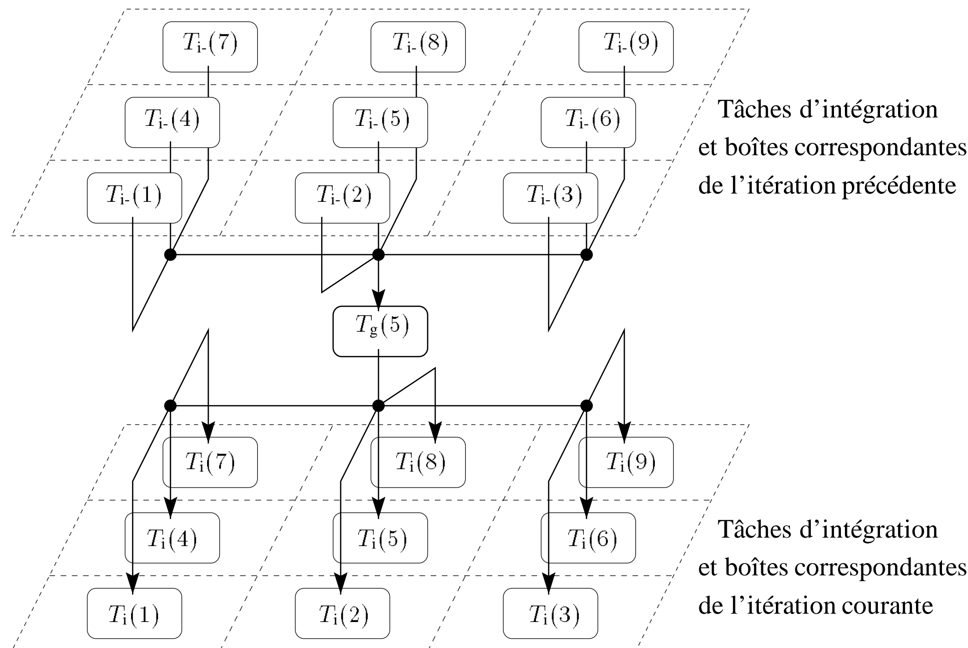


FIG. 5.5: Dépendances de la tâche $T_{\text{g}}(5)$ de calcul des forces géométriques des atomes de la boîte 5.

En analysant le graphe de tâches d'une itération on retrouve les deux phases de calculs et les deux phases de communications déjà présentes dans l'algorithme précédent. La figure 5.7 montre ces phases.

5.2.3 Contraintes de placement

Pour diminuer le volume des communications et simplifier les structures de données du programme, nous ajoutons des contraintes sur le placement des tâches. Nous montrerons expérimentalement par la suite que ces contraintes ne sont pas trop fortes. C'est-à-dire qu'elles ne nuisent pas à l'obtention d'un bon équilibre de charge.

Lors du premier programme, nous avons observé que les mouvements des atomes étaient essentiellement des oscillations autour d'une position d'équilibre. Dans ces conditions, il nous semble raisonnable de placer seulement les tâches relatives à la première

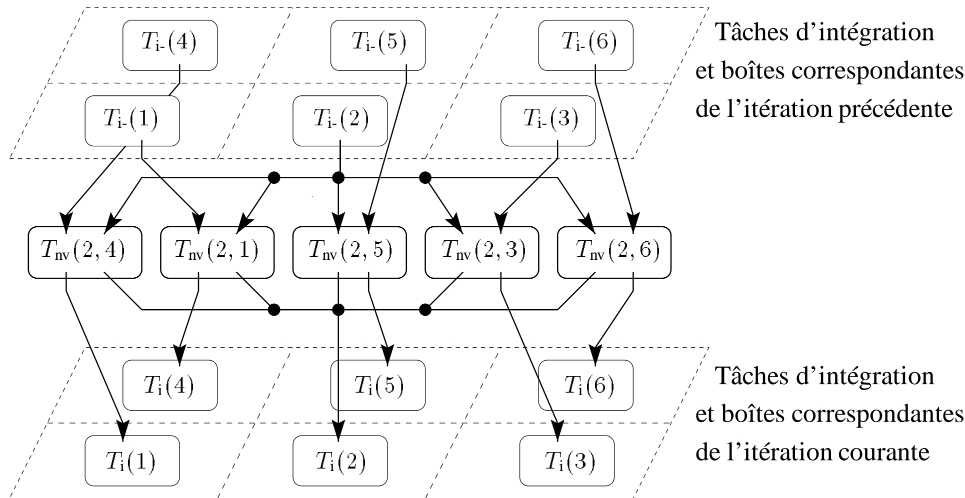


FIG. 5.6: Dépendances des tâches de calcul des forces non-liées entre les atomes des boîtes voisines de la boîte 2 : $T_{\text{nv}}(2,1)$, $T_{\text{nv}}(2,3)$, $T_{\text{nv}}(2,4)$, $T_{\text{nv}}(2,5)$, et $T_{\text{nv}}(2,6)$.

1	***Échanger*** *les coordonnées des atomes*
2	***Calculer*** *les forces (tâches : $T_{ni}(i)$, $T_{nv}(i,j)$ et $T_g(i)$)*
3	***Échanger*** *les forces*
4	***Intégrer*** *l'équation du mouvement (tâches : $T_i(i)$)*

FIG. 5.7: Phases de calculs et de communications d'une itération de dynamique moléculaire.

itération, et de supposer que ce placement restera grossièrement valable pour les itérations futures. Pour que le placement reste valable d'une itération à l'autre, il faut que les tâches relatives à une boîte ne bougent pas d'une itération à l'autre. Ainsi, concrètement, sur le graphe de tâches, cela signifie que les tâches $T_{\text{i-}}(i)$ et $T_{\text{i}}(i)$ seront placées sur le même processeur.

Pour diminuer les communications, on placera également la tâche $T_{\text{ni}}(i)$ des interactions non-liées entre les atomes de la boîte i, et la tâche $T_{\text{g}}(i)$ du calcul des forces géométriques relatives aux atomes de la boîte i sur le même processeur que la tâche d'intégration du mouvement des atomes de la boîte. Ces tâches utilisent et modifient les mêmes données, leur placement sur le même processeur évite la communication de ces données. Dans la suite, le placement des tâches $T_{\text{ni}}(i)$ implique le placement des tâches $T_{\text{i}}(i)$ et $T_{\text{g}}(i)$ sur le même processeur. Par extension, on parlera du placement de la boîte i pour désigner le placement de toutes ces tâches.

Toujours pour réduire les communications, on restreint le placement des tâches de type $T_{\text{nv}}(i,j)$ sur le processeur de $T_{\text{ni}}(i)$ ou de $T_{\text{ni}}(j)$. En effet, on peut remarquer que pour calculer les forces géométriques relatives aux atomes d'une boîte (les tâches $T_{\text{g}}(i)$), il faut

connaître la position des atomes des boîtes voisines. Et, après ce calcul, il faut mettre à jour les forces qui s'exercent sur les atomes des boîtes voisines. Donc le placement de ces tâches induit deux phases de communications : l'échange des positions des atomes entre les boîtes voisines, et l'échange des forces, toujours entre boîtes voisines. Ainsi, si on place la tâche $T_{\text{nv}}(i, j)$ sur le processeur qui possède la boîte i on peut profiter des communications nécessaires à $T_{\text{g}}(i)$. De même, si on place $T_{\text{nv}}(i, j)$ sur le processeur de la boîte j, on peut profiter des communications nécessaires à $T_{\text{g}}(j)$. On peut voir sur le graphe de tâches que les dépendances des tâches $T_{\text{nv}}(i, j)$ sont incluses dans celles des tâches $T_{\text{g}}(i)$ ou $T_{\text{g}}(j)$ (cf. figure 5.5 et figure 5.6). Cette contrainte de localité évite également les communications supplémentaires.

Au total, placer les tâches du graphe consiste :

- à placer les boîtes, et donc les tâches de type T_{ni}, T_{i} et T_{g} liées à leur placement ;
- puis à placer les tâches de calcul des forces non-liées entre les atomes de deux boîtes voisines (les tâches de type T_{nv}) en tenant compte des contraintes de localité définies précédemment.

5.2.4 Estimation des coûts

Comme nous l'avons dit précédemment (cf. 5.1.3), le déséquilibre de charge vient essentiellement du mauvais équilibre de calcul des forces d'interactions non-liées. Et sur une machine homogène, trouver un bon équilibre de charge signifie qu'il faut équilibrer le nombre d'interactions non-liées calculées sur chacun des processeurs. On peut négliger les autres coûts de calcul. Dans notre problème, deux types de tâches seulement calculent des forces d'interactions non-liées : les tâches de type T_{ni} et T_{nv}. Le placement de ces tâches induit le placement des boîtes et des autres tâches. Grâce à la complexité algorithmique de calcul, on peut estimer le nombre de forces non-liées calculées par ces tâches.

En utilisant n_i, le nombre d'atomes dans la boîte i, on peut avoir une estimation du nombre d'interactions non-liées calculées par les tâches :

- la tâche $T_{\text{ni}}(i)$ calcule les forces d'interactions non-liées entre les atomes de la boîte i, soit de l'ordre de n_i^2 interactions non-liées ;
- et la tâche $T_{\text{nv}}(i, j)$ calcule les forces non-liées entre les atomes de la boîte i et les atomes de la boîte j soit de l'ordre de $n_i \times n_j$ interactions non-liées.

Dans la suite, on notera $C(T)$ le coût de calcul de la tâche T.

5.3 Stratégies de placement

Indépendamment de la méthode de placement retenue, on commence par placer les tâches $T_{\text{ni}}(i)$, puis on termine par le placement des tâches $T_{\text{nv}}(i,j)$ en respectant les contraintes. De cette façon, trouver un placement des tâches consiste à définir une injection sur l'ensemble des p processeurs. Cette injection comporte deux parties :

– une première partie pour le placement des tâches $T_{\text{ni}}(i)$:

$$M : \left| \begin{array}{rcl} \{T_{\text{ni}}(i) \mid \forall i\} & \longrightarrow & \{1, \ldots, p\} \\ T_{\text{ni}}(i) & \longmapsto & q \end{array} \right.$$

– une deuxième partie pour le placement des tâches $T_{\text{nv}}(i,j)$ en respectant les contraintes de localité. Elle est fonction du placement précédent :

$$M : \left| \begin{array}{rcl} \{T_{\text{nv}}(i,j) \mid \forall (i,j)\} & \longrightarrow & \{1, \ldots, p\} \\ T_{\text{nv}}(i,j) & \longmapsto & q \end{array} \right. \qquad \text{avec} \quad q = \left\{ \begin{array}{c} M(T_{\text{ni}}(i)) \\ \text{ou} \;\; M(T_{\text{ni}}(j)) \end{array} \right.$$

Dans la suite, $M(T)$ désigne le processeur sur lequel est placé la tâche T indépendamment de son type.

La suite de ce paragraphe décrit les algorithmes de placement que nous avons essayés. Un premier placement aléatoire nous servira de référence. Puis un placement quantitatif essayera au mieux d'équilibrer les coûts de calculs des tâches. Les algorithmes suivants essayeront également de placer les boîtes voisines sur le même processeur afin de diminuer le volume des communications.

5.3.1 Placement aléatoire

Comme base de comparaison, nous avons programmé une stratégie de placement aléatoire satisfaisant aux contraintes de placement définies précédemment. On commence par placer les tâches de type T_{ni} aléatoirement sur les processeurs. On place ensuite les tâches de type T_{nv} toujours aléatoirement, mais en respectant les contraintes de placement. Concrètement, pour une tâche $T_{\text{nv}}(i,j)$ donnée, on choisit aléatoirement de la placer soit sur le processeur de $T_{\text{ni}}(i)$ soit sur le processeur de $T_{\text{ni}}(j)$.

5.3.2 Placement quantitatif

Le but de ce placement est d'équilibrer au mieux les calculs sur les processeurs, sans tenir compte des communications. On commence par le placement des tâches $T_{\text{ni}}(i)$ en utilisant la stratégie LPTF[1] bien connue [61]. On trie les tâches par coût décroissant et on les place au fur et à mesure sur le processeur disponible. Conjointement on calcule au fur et à mesure du placement de ces tâches la charge de calcul L_p de chacun des

[1] LPTF = *Largest Processing Time First*, plus gros temps de calcul en premier.

processeurs. À la fin de cette étape, L_p est la somme des coûts des tâches $T_{\text{ni}}(i)$ placées sur le processeur p :

$$L_p \leftarrow \sum_{T_{\text{ni}}(i) \in E_p} C(\,T_{\text{ni}}(i)\,) \quad \text{avec ici} \quad E_p = \{T_{\text{ni}}(i) \mid M(\,T_{\text{ni}}(i)\,) = p\}$$

Puis nous plaçons les tâches $T_{\text{nv}}(i,j)$ pour lesquelles, en raison des contraintes, le placement est imposé. C'est-à-dire, pour les tâches telles que $M(\,T_{\text{ni}}(i)\,) = M(\,T_{\text{ni}}(j)\,)$ on pose directement :

$$M(\,T_{\text{nv}}(i,j)\,) \leftarrow M(\,T_{\text{ni}}(i)\,)$$

Conjointement, on met à jour la charge de calcul L_p sur chacun des processeurs :

$$L_p \leftarrow L_p + \sum_{T_{\text{nv}}(i,j) \in E_p} C(\,T_{\text{nv}}(i,j)\,) \quad \text{avec ici} \quad E_p = \{T_{\text{nv}}(i,j) \mid M(\,T_{\text{ni}}(i)\,) = M(\,T_{\text{ni}}(j)\}$$

Puis nous plaçons les tâches $T_{\text{nv}}(i,j)$ pour lesquelles il existe un choix de placement. C'est-à-dire les tâches $T_{\text{nv}}(i,j)$ tel que $M(\,T_{\text{ni}}(i)\,) \neq M(\,T_{\text{ni}}(j)\,)$. Pour ces tâches, on utilise également une stratégie LPTF, en respectant les contraintes. Nous trions les m tâches $T_{\text{nv}}(i,j)$ restantes par ordre de coût décroissant. Puis nous les plaçons au fur et à mesure sur les processeurs les moins chargés. Ce dernier placement est défini de la façon suivante :

```
1  pour k ← 1 à m faire
         { on place la tâche T_nv(i_k, j_k) sur le processeur }
2     pi ← M(T_ni(i_k))                         {de la boîte i_k }
3     pj ← M(T_ni(j_k))                      {ou de la boîte j_k }
4     p ← minarg(L_pi, L_pj)               { sur le moins chargé. }
5     M(T_nv(i_k, j_k)) ← p
6     L_p ← L_p + C(T_nv(i_k, j_k))
7  fin pour
```

5.3.3 Placement par bi-partition récursive du graphe de tâches

Dans cette heuristique, plus qualitative, on cherche à prendre en compte la localité des données pour diminuer les communications. Plutôt que d'optimiser directement le placement d'un graphe de tâche, et de chercher directement à optimiser les communications, on cherche à placer astucieusement l'ensemble des tâches. Concrètement, on essaye de placer les boîtes voisines sur le même processeur.

L'idée consiste à découper récursivement l'ensemble des boîtes et des tâches de calcul associées en équilibrant la charge de calculs. Plus précisément, on coupe l'ensemble des boîtes et l'ensemble des processeurs en deux, en essayant d'équilibrer au mieux les coûts de calcul des deux sous-ensembles de tâches placées sur les deux sous-ensembles de

processeurs. Puis récursivement, suivant le même principe, on place chacun des deux sous-ensembles de boîtes, et les tâches de calcul associées, sur les sous-ensembles de processeurs correspondants. La figure 5.8 montre le principe de cette méthode. Enfin, comme précédemment, on termine par le placement des tâches $T_{\text{nv}}(i, j)$ non fixées (tâches dont les deux boîtes associées sont sur des processeurs différents) et toujours en essayant d'équilibrer les coûts de calcul.

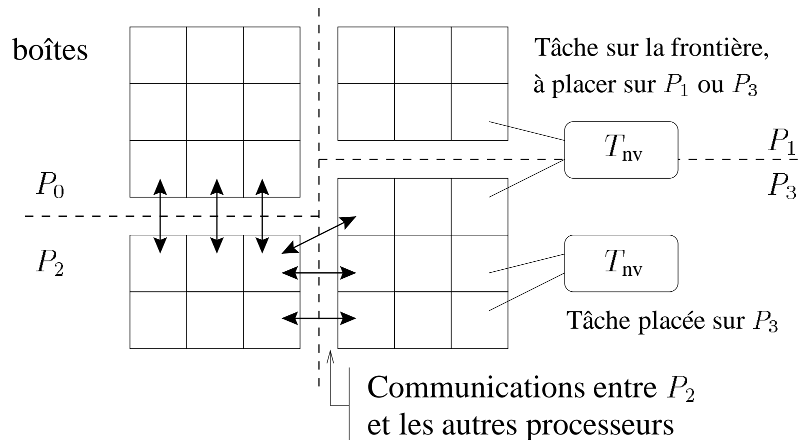


FIG. 5.8: Principe en deux dimensions d'un découpage récursif grossier de l'ensemble des boîtes pour quatre processeurs.

En fonction des coûts pris en compte pour équilibrer la charge, et de la façon de découper l'ensemble des boîtes, nous avons envisagé plusieurs variantes de cet algorithme :

– **BPR** (*Bi-Partition Récursive*), dans cette première variante, on coupe grossièrement l'ensemble des boîtes, comme le montre la figure 5.8. Pour faire le découpage, on utilise les estimations du coût de calcul des tâches.
– **BPR-CU** (*Bi-Partition Récursive et Coûts Unitaires des tâches*), pour cette variante, le découpage reste grossier et on considère que toutes les tâches ont le même coût de calcul.
– **BPR-Fin** (*Bi-Partition Récursive, découpage fin de l'ensemble des boîtes*), pour cette dernière variante, la partition de l'ensemble des boîtes est plus fine, comme le montre la figure 5.9. On utilise également les estimations du coût de calcul des tâches pour obtenir un bon équilibre.

Avant de présenter plus en détail ces trois variantes, nous allons définir quelques notations. Pour simplifier la partition, nous allons utiliser deux numérotations pour les boîtes de l'espace de simulation :

– une numérotation de position à trois indices (i_X, i_Y, i_Z) suivant les trois axes, avec $0 \leq i_X < \texttt{nbB}_X$, $0 \leq i_Y < \texttt{nbB}_Y$ et $0 \leq i_Z < \texttt{nbB}_Z$. Ici $\texttt{nbB}_X$, $\texttt{nbB}_Y$ et $\texttt{nbB}_Z$ représentent respectivement le nombre de boîtes suivant les trois axes X, Y et Z. La numérotation de position est également utilisée pour connaître les boîtes voisines

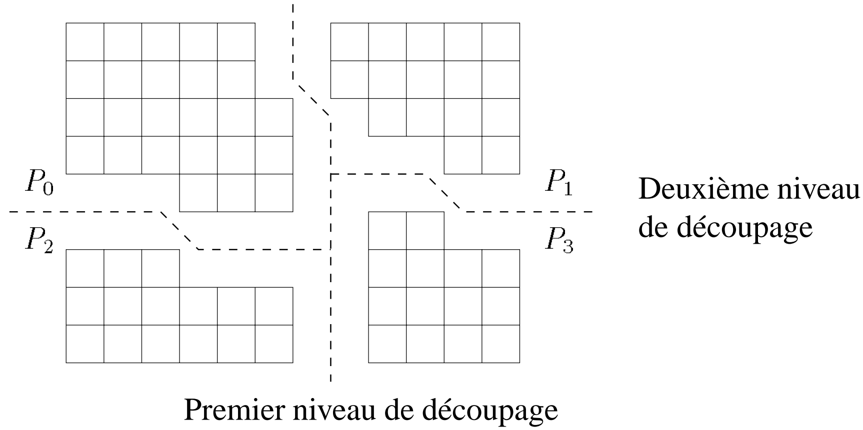


FIG. 5.9: Principe en deux dimensions d'un découpage récursif fin de l'ensemble des boîtes pour quatre processeurs.

d'une boîte donnée, ou pour placer les atomes dans les boîtes en fonction de leur position dans l'espace.

– et la numérotation linéaire que nous avons utilisée jusqu'à présent. On passe d'une numérotation à l'autre simplement en posant : $i = i_X + \texttt{nbB}_X \cdot (i_Y + \texttt{nbB}_Y \cdot iZ)$. Donc $0 \leq i < \texttt{nbB}$ et $\texttt{nbB} = \texttt{nbB}_X \cdot \texttt{nbB}_Y \cdot \texttt{nbB}_Z$ est le nombre de boîtes du système.

Placement BPR (*Bi-Partition Récursive*)

Dans la première variante, on utilise un découpage simple de l'ensemble des boîtes. À chaque étape de la récursivité, on cherche un plan perpendiculaire à un des trois axes et qui coupe l'ensemble des boîtes en deux parties. On cherche le plan qui minimise le déséquilibre de charge entre les deux parties.

Ensembles de départ : plus précisément, on part d'un ensemble EB de boîtes formant un pavé et délimité par les deux boîtes des sommets opposés du pavé : la boîte $(i_X^{\min}, i_Y^{\min}, i_Z^{\min})$ et la boîte $(i_X^{\max}, i_Y^{\max}, i_Z^{\max})$. Ainsi, on part de l'ensemble :

$$\texttt{EB} = \{(i_X, i_Y, i_Z) \mid (i_X^{\min}, i_Y^{\min}, i_Z^{\min}) \leq (i_X, i_Y, i_Z) < (i_X^{\max}, i_Y^{\max}, i_Z^{\max})\ \}$$

On tient compte également des tâches de type T_{nv} qui sont forcément placées sur les mêmes processeurs que les boîtes de l'ensemble EB. On note EV cet ensemble des tâches de type T_{nv} dont les deux boîtes associées sont dans EB.

$$\texttt{EV} = \{\ T_{\text{nv}}(i,j) \mid i,j \in \texttt{EB}\ \}$$

On doit placer ces tâches et ces boîtes sur un ensemble de Q processeurs. Pour le niveau l de récursion (l pour level) les boîtes de l'ensemble EB sont placées sur le même sous-ensemble de processeurs. Ce sous-ensemble de processeurs est caractérisé par les $l-1$ premiers bits du numéro de processeur final sur lequel seront placées les boîtes de

EB et les tâches de EV. Au niveau l, on détermine le bit l du numéro final du processeur de chacune de ces boîtes et de ces tâches. Cela revient à partager l'ensemble des processeurs en deux parties :

– la partie dont le bit l sera 1. Il y en a $Q_1 = \lfloor Q/2 \rfloor$;
– et la partie dont le bit l sera 0. Il y en a $Q_0 = Q - Q_1$.

On peut facilement remarquer que si Q est impair $Q_0 \neq Q_1$. Ainsi, pour équilibrer la charge, on doit évaluer les coûts des charges de calcul des deux parties du graphe de tâche et tenir compte du nombre de processeurs qui s'occupent de chacune des deux parties.

Partition des tâches et des boîtes : construire une partition de l'ensemble EB des boîtes consiste à couper le pavé de boîtes par un plan perpendiculairement à l'un des trois axes. Une coupe est donc définie par un axe et un index (i_X^{part}, i_Y^{part} ou i_Z^{part}) qui déterminent la position du plan de coupe. Par exemple si on coupe perpendiculairement à l'axe des X, on crée deux sous-ensembles de EB :

– la partie gauche $\mathtt{EB}_g(i_X^{\text{part}})$, définie par

$$\mathtt{EB}_g(i_X^{\text{part}}) = \{(i_X, i_Y, i_Z) \mid (i_X^{\text{min}}, i_Y^{\text{min}}, i_Z^{\text{min}}) \leq (i_X, i_Y, i_Z) < (i_X^{\text{part}}, i_Y^{\text{max}}, i_Z^{\text{max}})\ \}$$

– et la partie droite $\mathtt{EB}_d(i_X^{\text{part}})$, définie par

$$\mathtt{EB}_d(i_X^{\text{part}}) = \{(i_X, i_Y, i_Z) \mid (i_X^{\text{part}}, i_Y^{\text{min}}, i_Z^{\text{min}}) \leq (i_X, i_Y, i_Z) < (i_X^{\text{max}}, i_Y^{\text{max}}, i_Z^{\text{max}})\ \}$$

La partition de l'ensemble des boîtes induit une partition de l'ensemble EV des tâches de type T_{nv} en trois parties :

– l'ensemble des tâches $T_{\text{nv}}(i, j)$ dont les deux boîtes associées sont dans la partie gauche

$$\mathtt{EV}_g(i_X^{\text{part}}) = \{\ T_{\text{nv}}(i, j) \mid i, j \in \mathtt{EB}_g(i_X^{\text{part}})\ \}$$

– l'ensemble des tâches $T_{\text{nv}}(i, j)$ dont les deux boîtes associées sont dans la partie droite

$$\mathtt{EV}_d(i_X^{\text{part}}) = \{\ T_{\text{nv}}(i, j) \mid i, j \in \mathtt{EB}_d(i_X^{\text{part}})\ \}$$

– et l'ensemble des tâches $T_{\text{nv}}(i, j)$ dont l'une des deux boîtes associées se trouve dans la partie droite et l'autre dans la partie gauche

$$\mathtt{EV}_c(i_X^{\text{part}}) = \{\ T_{\text{nv}}(i, j) \mid (i \in \mathtt{EB}_g(i_X^{\text{part}})) \wedge (j \in \mathtt{EB}_d(i_X^{\text{part}}))\ \}$$

la figure 5.10 montre une telle partition d'un ensemble de boîtes.

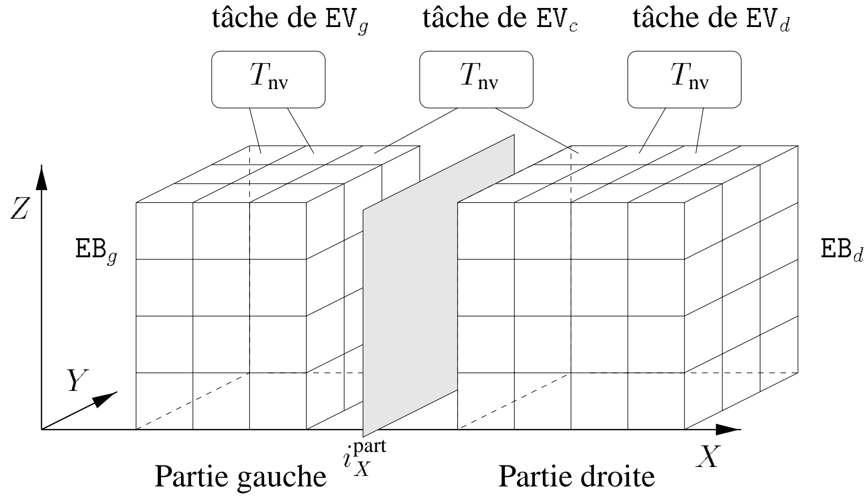


FIG. 5.10: Partition simple de l'ensemble des boîtes, et des ensembles correspondants.

Estimation des coûts de la partition : pour estimer le déséquilibre de charge d'une partition, on doit estimer le nombre d'interactions non-liées calculées par chacune des deux parties. On note $L_g(i_X^{\text{part}})$ et $L_d(i_X^{\text{part}})$ le coût de calcul de la partie gauche et de la partie droite respectivement.

On commence par calculer la charge induite par les tâches de type T_{ni} associées aux boîtes de EB :

$$L_g(i_X^{\text{part}}) \leftarrow \sum_{i \,\in\, \mathtt{EB}_g(i_X^{\text{part}})} C(\ T_{\text{ni}}(i)\ ) \tag{5.1}$$

$$L_d(i_X^{\text{part}}) \leftarrow \sum_{i \,\in\, \mathtt{EB}_d(i_X^{\text{part}})} C(\ T_{\text{ni}}(i)\ ) \tag{5.2}$$

Puis on ajoute le coût induit par les tâches de type T_{nv} :

$$L_g(i_X^{\text{part}}) \leftarrow L_g(i_X^{\text{part}}) + \sum_{T_{\text{nv}}(i,j) \,\in\, \mathtt{EV}_g(i_X^{\text{part}})} C(\ T_{\text{nv}}(i,j)\ ) \tag{5.3}$$

$$L_d(i_X^{\text{part}}) \leftarrow L_d(i_X^{\text{part}}) + \sum_{T_{\text{nv}}(i,j) \,\in\, \mathtt{EV}_d(i_X^{\text{part}})} C(\ T_{\text{nv}}(i,j)\ ) \tag{5.4}$$

Évaluation d'une partition : on termine par évaluer le déséquilibre de charge entre les deux parties. Par exemple, si on place la partie gauche ($\mathtt{EB}_g(i_X^{\text{part}})$ et $\mathtt{EV}_g(i_X^{\text{part}})$) sur l'ensemble Q_0 de processeurs et la partie droite ($\mathtt{EB}_d(i_X^{\text{part}})$ et $\mathtt{EV}_d(i_X^{\text{part}})$) sur l'autre ensemble Q_1, le déséquilibre de charge est :

$$\Delta C(i_X^{\text{part}}) = \left| \frac{L_g(i_X^{\text{part}})}{Q_0} - \frac{L_d(i_X^{\text{part}})}{Q_1} \right|$$

Et inversement, si on place la partie gauche sur l'ensemble Q_1 de processeurs et la partie droite sur l'ensemble Q_0, le déséquilibre de charge est :

$$\Delta C(i_X^{\text{part}}) = \left| \frac{L_g(i_X^{\text{part}})}{Q_1} - \frac{L_d(i_X^{\text{part}})}{Q_0} \right|$$

En résumé, on cherche la partition qui minimise le déséquilibre de charge en coupant suivant les trois axes et en cherchant à chaque fois à placer la partie gauche sur l'ensemble Q_0 et la partie droite sur l'ensemble Q_1 et inversement en cherchant à placer la partie gauche sur l'ensemble Q_1 et la partie droite sur l'ensemble Q_0. Donc au total, on cherche la partition qui réalise le minimum de :

$$\min \left\{ \begin{array}{l} \displaystyle\min_{i_X^{\text{part}}=i_X^{\min}}^{i_X^{\max}-1} \left\{ \left| \frac{L_g(i_X^{\text{part}})}{Q_0} - \frac{L_d(i_X^{\text{part}})}{Q_1} \right| , \left| \frac{L_g(i_X^{\text{part}})}{Q_1} - \frac{L_d(i_X^{\text{part}})}{Q_0} \right| \right\} , \\ \displaystyle\min_{i_Y^{\text{part}}=i_Y^{\min}}^{i_Y^{\max}-1} \left\{ \left| \frac{L_g(i_Y^{\text{part}})}{Q_0} - \frac{L_d(i_Y^{\text{part}})}{Q_1} \right| , \left| \frac{L_g(i_Y^{\text{part}})}{Q_1} - \frac{L_d(i_Y^{\text{part}})}{Q_0} \right| \right\} , \\ \displaystyle\min_{i_Z^{\text{part}}=i_Z^{\min}}^{i_Z^{\max}-1} \left\{ \left| \frac{L_g(i_Z^{\text{part}})}{Q_0} - \frac{L_d(i_Z^{\text{part}})}{Q_1} \right| , \left| \frac{L_g(i_Z^{\text{part}})}{Q_1} - \frac{L_d(i_Z^{\text{part}})}{Q_0} \right| \right\} \end{array} \right\}$$

Puis on découpe chacune des parties récursivement de la même manière jusqu'à l'obtention d'un nombre de parties équivalentes au nombre de processeurs.

Comme pour le placement précédent, on termine par placer les tâches de calculs des interactions non-liées entre boîtes voisines placées sur les processeurs différents. C'est-à-dire les tâches $T_{\text{nv}}(i,j)$ telles que $M(T_{\text{ni}}(i)) \neq M(T_{\text{ni}}(j))$. Pour cela, on estime la charge L_p de calculs sur les processeurs en fonction des tâches placées.

$$L_p \leftarrow \sum_{M(T_{\text{ni}}(i))=p} C(T_{\text{ni}}(i)) + \sum_{M(T_{\text{ni}}(i))=M(T_{\text{ni}}(j))=p} C(T_{\text{nv}}(i,j))$$

Puis, on place les tâches $T_{\text{nv}}(i,j)$ restantes par ordre décroissant de coût sur les processeurs les moins chargés.

BPR-CU (*Bi-Partition Récursive et Coûts Unitaires des tâches*)

Le principe de ce placement est identique au précédent, seule l'estimation du coût de la partition change. En effet, dans cette variante, on considère que toutes les tâches ont des coûts de calcul identiques. On peut ainsi considérer que toutes les tâches sont de coût unitaire. L'estimation des coûts des deux parties de la partition change. Ainsi dans l'algorithme précédent on remplace les équations 5.1 et 5.2 par respectivement :

$$L_g(i_X^{\text{part}}) \leftarrow \texttt{card}\left(\texttt{EB}_g(i_X^{\text{part}}) \right) \quad , \quad L_d(i_X^{\text{part}}) \leftarrow \texttt{card}\left(\texttt{EB}_d(i_X^{\text{part}}) \right)$$

Ici $\texttt{card}(E)$ désigne le cardinal de l'ensemble E. De même on remplace les équations 5.3 et 5.4 par respectivement :

$$L_g(i_X^{\text{part}}) \leftarrow L_g(i_X^{\text{part}}) + \texttt{card}\left(\texttt{EV}_g(i_X^{\text{part}}) \right)$$

$$L_d(i_X^{\text{part}}) \leftarrow L_d(i_X^{\text{part}}) + \texttt{card}\left(\ \texttt{EV}_d(i_X^{\text{part}})\ \right)$$

Enfin, pour la dernière partie de l'algorithme qui place les tâches de type T_{nv} dont les deux boîtes associées sont sur des processeurs différents, on considère également que les tâches sont de coût unitaire.

BPR-Fin (*Bi-Partition Récursive, découpage fin de l'ensemble des boîtes*)

Cette variante se différentie par la façon de couper l'ensemble des boîtes. Dans les variantes précédentes de l'algorithme, on découpe l'ensemble des boîtes par un plan. C'est un découpage grossier. En effet, l'équilibre est possible au plan près. Pour cette variante, on cherche un découpage plus fin, avec une granularité de découpage de l'ordre de la boîte. Pour simplifier la compréhension, on place toujours la partie gauche sur l'ensemble Q_0 de processeurs et la partie droite sur l'ensemble Q_1 de processeurs. Le partitionnement de l'ensemble en deux comporte trois étapes :

1. On découpe l'ensemble en trois parties, comme le montre la figure 5.11, la partie centrale est un plan de boîtes de l'épaisseur d'une boîte. On le choisit tel que si on l'ajoute à la partie gauche, la partie gauche est surchargée et inversement si on l'ajoute à la partie droite c'est la partie droite qui est surchargée. Autrement dit, si on commence par couper perpendiculairement à l'axe des X, on cherche i_X^{part} tel que :

$$\frac{L_g(i_X^{\text{part}}+1)}{Q_0} \geq \frac{L_d(i_X^{\text{part}}+1)}{Q_1} \quad \text{et} \quad \frac{L_g(i_X^{\text{part}})}{Q_0} \leq \frac{L_d(i_X^{\text{part}})}{Q_1}$$

2. On découpe ensuite ce plan, toujours en essayant d'équilibrer la charge. On obtient un segment de boîtes de l'épaisseur d'une boîte ;
3. On termine par couper ce segment en deux perpendiculairement au dernier axe.

On obtient un découpage fin en deux parties, comme le montre la figure 5.11.

Remarquons que, toujours pour minimiser les communications, on essaye de construire des ensembles de boîtes sur les processeurs qui soient le plus proches possibles d'un cube. En effet, ainsi on minimise globalement le nombre de boîtes sur le bord des ensembles. Pour cela, lors de la première étape d'une partition, on commence toujours par couper l'ensemble de boîtes perpendiculairement à l'axe comportant le plus de boîtes.

5.3.4 Placement de la structure

Le placement des informations de la structure est conditionné par le placement des atomes correspondants. En effet, le placement des tâches détermine complètement le placement des boîtes et des atomes sur les processeurs. Donc, on place les informations et le calcul d'une interaction géométrique sur un processeur possédant au moins un des atomes de l'interaction géométrique. On essaye aussi d'équilibrer le nombre d'interactions géométriques calculées sur chacun des processeurs. Par exemple, pour une liaison entre deux atomes, deux cas sont possibles pour choisir le processeur où placer la liaison :

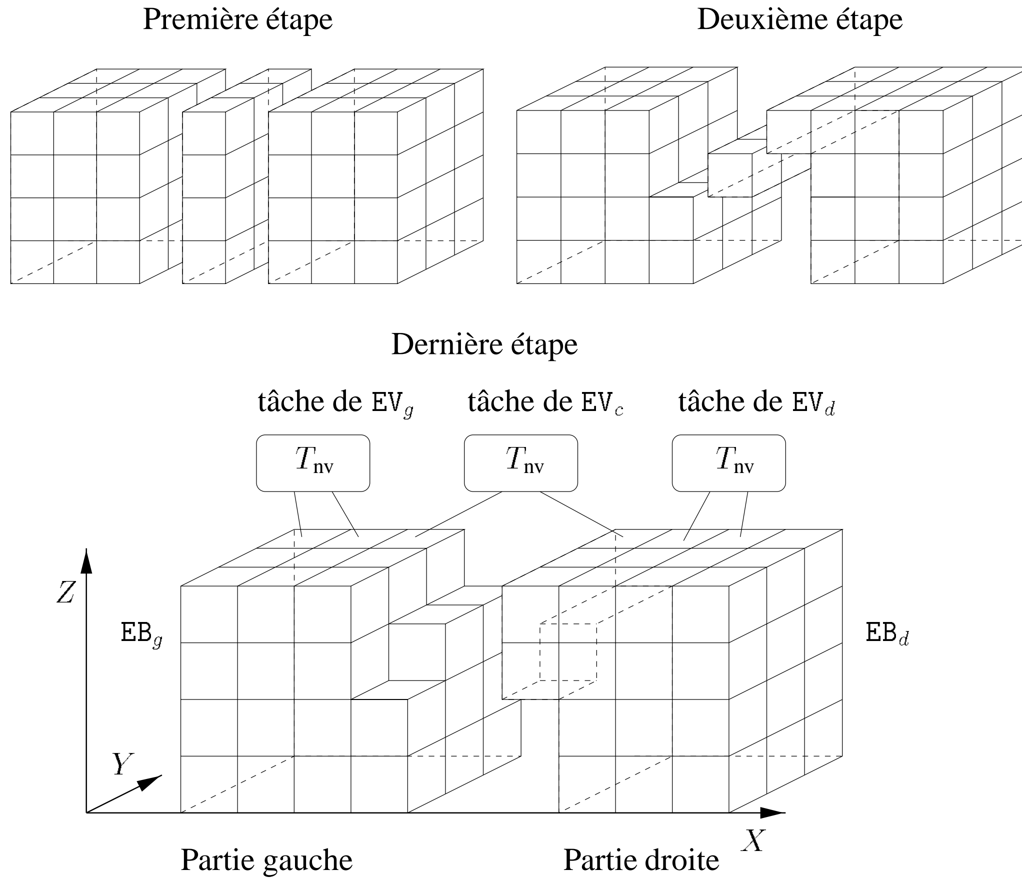


FIG. 5.11: Les trois étapes d'une partition fine de l'ensemble des boîtes, et des ensembles correspondants.

- si les deux atomes de la liaison sont placés sur le même processeur, on place la liaison sur ce processeur ;
- si les deux atomes sont placés sur des processeurs différents, on place la liaison sur le processeur le moins chargé. C'est-à-dire le processeur possédant le moins de liaisons.

On procède de façon analogue pour placer les interactions des angles et les interactions des angles dièdres.

5.3.5 Schémas de communication

Comme nous l'avons dit précédemment, deux phases de communications séparent les deux phases de calculs dans une itération de dynamique moléculaire (cf. figure 5.7). La première phase échange les coordonnées des atomes, pour calculer les forces. Et la deuxième phase échange les forces pour intégrer les équations du mouvement. Ces deux phases de communications sont irrégulières et dépendent du placement des boîtes sur les processeurs.

Nous avons vu en construisant le graphe de tâches (cf. figure 5.5) que les communications sont dues aux dépendances de données des tâches de type T_{g}, tâches de calcul des forces d'interactions géométriques. De plus, grâce aux contraintes de localité, les dépendances des tâches T_{nv} de calcul des forces d'interactions non-liées entre boîtes voisines sont incluses dans les dépendances des tâches de type T_{g}. On peut également remarquer, sur le graphe de tâches, que plusieurs tâches de calcul des forces d'interactions géométriques peuvent nécessiter des données communes. Pour éviter de communiquer plusieurs fois les mêmes données entre les processeurs, on effectue en fonction du placement des boîtes une programmation des phases de communications. C'est-à-dire que pour chacune des deux phases de communications, chacun des processeurs détermine les données qu'il doit envoyer aux autres processeurs en regroupant les communications de données communes. On peut qualifier ces deux phases de communication d'échange partiel personnalisé, chacun des processeurs envoie des données différentes à un nombre variable de processeurs.

Concrètement, pour chacun des processeurs, on construit l'ensemble $\texttt{Eb}_p$ des boîtes placées sur le processeur p. Et pour chacune des boîtes, on construit l'ensemble V_i des boîtes voisines de la boîte i au sens du rayon de coupure. C'est-à-dire l'ensemble des boîtes j tel qu'il existe une tâche $T_{\text{nv}}(i, j)$.

$$\texttt{Eb}_p = \{i \mid M(T_{\text{ni}}(i)) = p\} \quad \text{et} \quad V_i = \{j \mid \exists T_{\text{nv}}(i, j)\}$$

Puis on construit pour chacun des processeurs q les ensembles $E_{p \to q}$, ensemble des boîtes du processeur p à envoyer sur le processeur q.

$$E_{p \to q} = \{i \in \texttt{Eb}_p \mid \exists j \in V_i \text{ avec } M(T_{\text{ni}}(j)) = q\}$$

On doit envoyer les coordonnées des atomes de la boîte i placée sur p au processeur q, s'il existe une boîte j voisine de i placée sur le processeur q. En effet, pour pouvoir calculer les forces d'interactions géométriques relativement aux atomes de la boîte j (tâches $T_{\text{g}}(j)$), il faut connaître les coordonnés des atomes de toutes les boîtes voisines de j (et en particulier les coordonnés des atomes de la boîte i). On note que si $E_{p \to q}$ est l'ensemble vide alors il n'y a pas de communications entre les processeurs p et q.

Les ensembles $E_{p \to q}$ permettent ainsi l'envoi unique des coordonnées des atomes des boîtes aux processeurs qui en ont besoin. On utilise un procédé similaire pour la deuxième phase de communications qui regroupe les forces s'exerçant sur les atomes des boîtes.

Remarquons que généralement les heuristiques de placement statique des tâches qui prennent en compte les communications entre les tâches considèrent que ces communications sont indépendantes [69]. En regroupant les communications, nous utilisons justement le fait que, pour ce problème, les communications entre les tâches ne sont pas indépendantes.

5.3.6 Critères pour l'évaluation des placements

Pour évaluer la qualité des placements et mieux observer leurs propriétés, nous allons définir quelques critères mathématiques sur ces placements. En particulier, pour les

comparer, on cherche à évaluer : le déséquilibre de charge, le volume moyen des communications par processeurs et la localité du placement. Notons que la localité et le volume des communications sont fortement liés, une bonne localité diminue le volume des communications.

Déséquilibre de charge

Pour évaluer le déséquilibre de charge, on calcule l'écart-type du nombre d'interactions non-liées calculées sur chacun des processeurs. Si L_p est la charge du processeur p, on définit le déséquilibre de charge ΔL sur $\mathtt{P}$ processeurs par :

$$\Delta L = \sqrt{\frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P}-1} (L_p - \overline{L})^2} \quad \text{avec} \quad \overline{L} = \frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P}-1} L_p$$

Ici $\overline{L}$ est la charge moyenne des processeurs. Pour avoir un bon équilibre de charge, il faut que ΔL soit le plus petit possible.

Volume moyen des communications

Le coût de communication provient essentiellement du volume des informations relatives aux atomes échangées entre les processeurs, à savoir, l'échange des coordonnées des atomes et l'échange des forces. Ainsi, on définit le coût de communication d'un processeur p vers un processeur q ($L_{\text{com}}(p \to q)$) comme le nombre d'atomes de p dont on doit envoyer les coordonnées vers le processeur q.

$$L_{\text{com}}(p \to q) = \sum_{i \in E_{p \to q}} \mathtt{nbAt}_i$$

Ici $\mathtt{nbAt}_i$ est le nombre d'atomes de la boîte i.

Puis on définit naturellement le volume moyen de communication par processeur comme la somme des informations échangées divisé par le nombre de processeurs $\mathtt{P}$:

$$\bar{L}_{\text{com}} = \frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P}-1} \sum_{q=0}^{\mathtt{P}-1} L_{\text{com}}(p \to q)$$

Pour obtenir des bonnes performances le volume des communications doit être le plus faible possible.

Localité

Concrètement une bonne localité signifie que les boîtes voisines au sens du rayon de coupure sont placées sur le même processeur. Pour mesurer la proportion de boîtes voisines placées sur le même processeur, on utilise les tâches de type T_{nv}. On définit la

localité comme la proportion de tâches de type T_{nv} dont les deux boîtes associées sont sur le même processeur. Plus précisément, on définit la localité par le pourcentage suivant :

$$loc = \frac{\text{nombre de tâches } T_{\text{nv}} \text{ dont les deux boîtes associées sont sur le même processseur}}{\text{nombre total de tâches } T_{\text{nv}}} \times 100$$

Ce critère est le contraire du *cut edge* (proportion d'arêtes ayant des sommets sur des processeurs différents) utilisé dans le partitionnement de maillages [44]. La localité dépend du nombre de processeurs et de la taille du problème, comparer la localité entre deux placements a un sens seulement pour un même problème et pour un nombre de processeurs fixé. Plus ce pourcentage est grand, meilleur est la localité et plus le volume des communications est faible.

5.4 Comparaison des placements

5.4.1 Propriétés des placements

Nous commençons par comparer les cinq stratégies de placement à l'aide des trois critères de comparaison définis précédemment. Le tableau 5.1 donne une évaluation de chacun des critères pour les cinq placements en utilisant la structure de 11 615 atomes décrite dans le chapitre 4.4. Nous utilisons un rayon de coupure de 10 Å. Ainsi la structure comprend 216 boîtes, c'est également le nombre de tâches de type T_{ni} ; et elle comprend 1 940 tâches de type T_{nv}.

TAB. 5.1: Critères de comparaison des placements.

Équilibre de charge $\Delta L(\mathtt{P})$ $(\times 10^3)$							
Nb. Proc. ($\mathtt{P}$)	1	2	4	8	16	24	32
Placement aléatoire	0,0	635,0	730,0	676,0	408,0	317,0	252,0
Placement quantitatif	0,0	0,0	0,0	0,0	0,4	0,1	4,1
Placement BPR	0,0	0,0	1,7	0,7	8,5	18,2	33,9
Placement BPR-CU	0,0	138,8	45,8	137,0	475,7	304,2	296,6
Placement BPR-FIN	0,0	0,0	0,0	0,4	2,2	13,0	10,0

Volume de communications $\bar{L}_{\text{com}}(\mathtt{P})$ $(\times 10^3)$							
Nb. Proc. ($\mathtt{P}$)	1	2	4	8	16	24	32
Placement aléatoire	0	5,81	8,63	9,56	8,16	6,92	5,76
Placement quantitatif	0	5,80	8,70	9,72	8,68	7,20	6,02
Placement BPR	0	3,15	3,99	3,87	2,99	2,90	2,39
Placement BPR-CU	0	2,97	3,77	2,86	2,51	2,30	2,07
Placement BPR-FIN	0	3,37	4,29	3,63	3,18	3,44	2,77

Localité $loc(\mathtt{P})$ (%)							
Nb. Proc. ($\mathtt{P}$)	1	2	4	8	16	24	32
Placement aléatoire	100	49,5	24,9	12,8	6,3	4,4	2,8
Placement quantitatif	100	51,8	26,4	15,0	8,5	5,9	4,8
Placement BPR	100	86,8	67,8	52,2	44,0	36,5	31,1
Placement BPR-CU	100	86,8	75,3	65,2	45,0	24,7	21,9
Placement BPR-FIN	100	85,9	74,1	63,3	43,6	36,1	30,3

Les critères d'évaluation des placements confirment les propriétés attendues de chacun des placements. En effet, le placement quantitatif, indépendamment du nombre de

processeurs, offre le meilleur équilibre de charge mais génère autant de communications que le placement aléatoire.

Les placements par bi-partition récursive permettent de diminuer le volume des communications grâce à une meilleure localité des données que les autres placements. Les placements BPR et BPR-FIN permettent également d'obtenir des équilibres de charge bien meilleurs qu'un placement aléatoire même s'ils sont légèrement moins bons que pour le placement quantitatif. Au vu de ces critères, les deux placements par bi-partition récursive, BPR et BPR-FIN, semblent avoir les mêmes caractéristiques.

Notons, pour terminer que le placement BPR-CU qui cherche seulement à équilibrer le nombre de tâches sur chacun des processeurs n'équilibre pas pour autant le nombre d'interactions non-liées calculées sur chacun des processeurs. Ceci confirme nos premières observations, à savoir que le nombre d'interactions non-liées à calculer n'est pas uniformément réparti dans le système.

Ces critères permettent facilement de comparer les propriétés des heuristiques de placements, ils ne permettent pas de connaître l'impact du déséquilibre de charge et des communications sur les performances du programme. Plus précisément, ces critères d'évaluation des placements ne permettent pas de trancher entre un placement BPR-FIN qui diminue le volume des communications légèrement au détriment de l'équilibre de charge, et un placement quantitatif qui offre un meilleur équilibre de charge mais qui ne minimise pas les communications. Pour trancher, nous allons présenter les résultats des mesures de performance.

5.4.2 Résultats

Comme précédemment, toutes les mesures ont été réalisées sur l'IBM-SP1 de l'IMAG. Nous commencerons par comparer les trois heuristiques de placements principales, à savoir : le placement aléatoire, le placement quantitatif (LPTF), et le placement par bi-partition récursive fine (BPR-FIN).

Le premier graphique (figure 5.12) compare les trois heuristiques de placements principales, à savoir : le placement aléatoire, le placement quantitatif (LPTF), et le placement par bi-partition récursive fine (BPR-FIN).

Plus précisément, ce graphique donne le temps d'exécution moyen des 100 premières itérations de dynamique moléculaire toujours pour le système de 11 615 atomes, et avec un rayon de coupure de 10 Å. Rappelons que nous ne tenons pas compte dans ce chapitre du temps nécessaire à la mise à jour des atomes dans les boîtes. Normalement cette mise à jour a lieu environ toutes les 50 itérations. Le temps optimal théorique ne tient pas compte des communications, c'est simplement le temps d'exécution moyen d'une itération de dynamique moléculaire sur un processeur divisé par le nombre de processeurs.

Pour ce système, et indépendamment du nombre de processeurs, le placement par bi-partition récursive du graphe de tâche donne les meilleurs résultats. Dans la suite, nous allons analyser plus finement pourquoi.

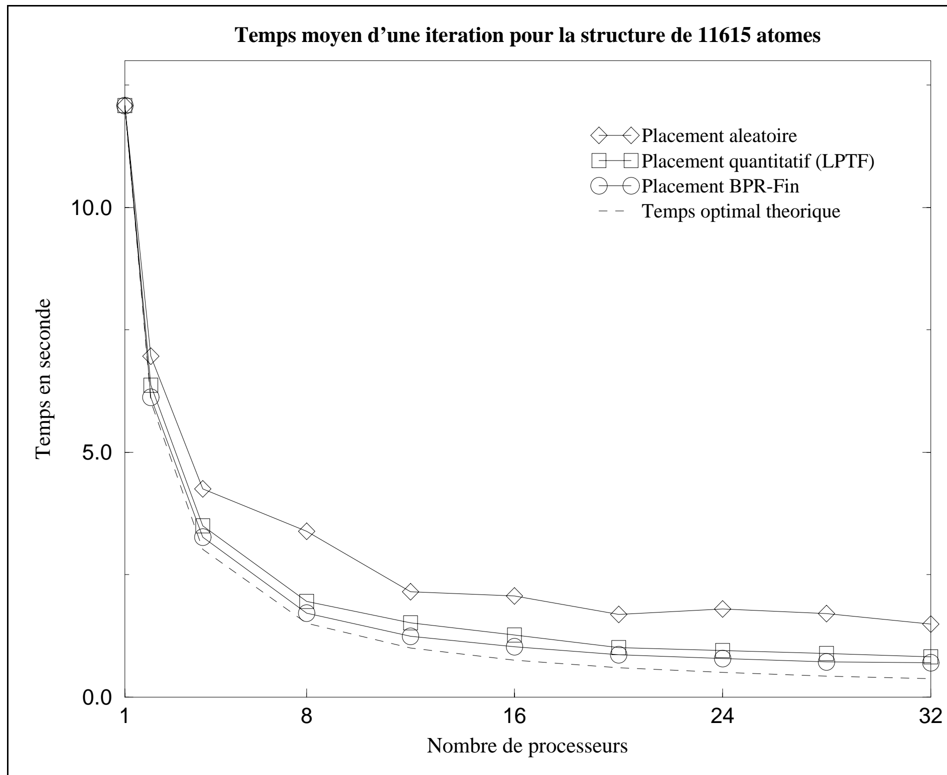


FIG. 5.12: Comparaison du temps d'exécution moyen d'une itération de dynamique moléculaire pour le système de 11 615 atomes en utilisant différents placements initiaux.

5.4.3 Comparaison plus fine

Le tableau 5.2 donne le détail des temps d'exécution moyens d'une itération de dynamique moléculaire, en utilisant différents nombres de processeurs. Les moyennes ont été calculées sur les 100 premières itérations et toujours avec le même système de 11 615 atomes.

Rappelons que :

- t_{calc}, le temps de calcul, représente le temps passé par le processeur pour faire les calculs directement utiles pour la simulation numérique ;
- t_{comm} est le temps passé par le processeur pour réaliser et gérer les communications ;
- t_{att} est le temps passé par le processeur à attendre une communication. Il est caractéristique de l'inactivité du processeur ;
- $t_{\text{tot}} = t_{\text{calc}} + t_{\text{comm}} + t_{\text{att}}$ est le temps moyen total de l'itération de dynamique moléculaire.

La mesure très précise de ces différents temps est techniquement difficile. Il n'est pas toujours possible de déterminer à chaque instant l'état du processeur, pour savoir s'il calcule pour la simulation, s'il gère des communications ou s'il attend. En effet, les mesures ont été réalisées sur la plate-forme ATHAPASCAN-0A (cf. 2.2.3) qui utilise la librairie

TAB. 5.2: Détail des temps d'exécution d'une itération de dynamique moléculaire, pour un système de 11 615 atomes.

Placement aléatoire (temps en seconde)							
Nb. Proc. (p)	1	2	4	8	16	24	32
$t_{\text{calc}}(p)$	12,08	6,03	3,04	1,57	0,78	0,53	0,40
$t_{\text{comm}}(p)$	0,00	0,20	0,25	0,26	0,25	0,25	0,26
$t_{\text{att}}(p)$	0,00	0,72	0,96	1,55	1,04	1,02	0,84
$t_{\text{tot}}(p)$	12,08	6,95	4,25	3,38	2,07	1,80	1,50

Placement quantitatif (temps en seconde)							
Nb. Proc. (p)	1	2	4	8	16	24	32
$t_{\text{calc}}(p)$	12,08	6,04	3,06	1,55	0,78	0,51	0,40
$t_{\text{comm}}(p)$	0,00	0,20	0,25	0,27	0,27	0,26	0,26
$t_{\text{att}}(p)$	0,00	0,13	0,19	0,14	0,22	0,17	0,17
$t_{\text{tot}}(p)$	12,08	6,37	3,50	1,96	1,27	0,94	0,83

Placement BPR-FIN (temps en seconde)							
Nb. Proc. (p)	1	2	4	8	16	24	32
$t_{\text{calc}}(p)$	12,08	5,94	3,03	1,53	0,77	0,52	0,39
$t_{\text{comm}}(p)$	0,00	0,14	0,14	0,12	0,11	0,11	0,11
$t_{\text{att}}(p)$	0,00	0,04	0,09	0,06	0,15	0,16	0,20
$t_{\text{tot}}(p)$	12,08	6,12	3,26	1,71	1,03	0,79	0,70

de communication PVM. Ainsi une partie du temps d'attente mesuré correspond en réalité au temps consommé par le démon de communication de PVM. Plus précisément, le temps d'attente (t_{att}) mesure l'inactivité du processus utilisateur mais pas l'inactivité du processeur physique sur lequel il s'exécute. En réalité, le processeur physique peut utiliser ce temps pour exécuter les autres processus, dont le démon de communication de PVM. C'est pourquoi nous nous contenterons d'apprécier seulement les différences importantes entre les chiffres.

Dans le calcul des itérations de dynamique moléculaire, et suite à un placement initial aléatoire, les processeurs passent en moyenne beaucoup de temps à attendre. Le phénomène empire quand on augmente le nombre de processeurs. Ainsi, par exemple, sur 16 processeurs, c'est environ 50 % du temps total d'une itération qui est perdu dans l'attente des communications. Le placement aléatoire n'équilibre pas la charge de calcul. Le temps de l'itération est le temps du processeur le plus chargé, les autres processeurs doivent l'attendre. De plus, quand on utilise beaucoup de processeurs, le déséquilibre de charge est

très pénalisant. Avec ce placement, l'accélération sur 16 processeurs n'est que de 5,8.

L'heuristique de placement quantitatif (LPTF), comme le montre le critère d'équilibre de charge, génère un placement plus équitable. En pratique, ceci se traduit par une diminution sensible du temps d'attente moyen par processeur. Ainsi, sur 16 processeurs, le temps d'attente ne représente plus que 17 % du temps moyen d'une itération de dynamique moléculaire. Les performances globales de l'itération sont meilleures que pour le placement aléatoire précédent. Toujours sur 16 processeurs, l'accélération est de 9,5.

Pour ce placement, le temps de communications reste important. Sur 16 processeurs, il représente au moins 21 % du temps total de l'itération. Il représente même certainement beaucoup plus. En effet, une partie du temps d'attente correspond en réalité à l'exécution du démon de communication de PVM. Ceci explique pourquoi, malgré un meilleur équilibre de charge, les temps d'attente restent importants pour ce placement.

L'heuristique de placement BPR-FIN permet, indépendamment du nombre de processeurs, d'obtenir les meilleures performances. Il réduit les temps moyens de communications, par rapport aux deux autres placements, sans dégrader l'équilibre de charge. Sur 16 processeurs, le temps nécessaire aux communications ne représente plus qu'environ 10 % du temps total. Globalement, cette heuristique permet d'obtenir une accélération de 11,7 sur 16 processeurs pour le calcul du mouvement de la structure de 11 615 atomes. La diminution du volume de communication diminue également le temps consommé par le démon de communication. Ceci se traduit par des temps d'attente généralement inférieurs à ceux du placement quantitatif.

Le placement BPR-FIN diminue le volume de communication, mais également le nombre de liens de communications par rapport au placement quantitatif. En effet, pour le placement quantitatif, chacun des 16 processeurs communique avec les 15 autres processeurs. En revanche, pour le placement BPR-FIN, en moyenne, chacun des processeurs communique avec seulement 8 autres processeurs.

Pour lever l'ambiguïté sur l'interprétation du temps d'attente, nous avons observé le détail des temps d'exécution sur chacun des processeurs. En effet, rappelons que le temps d'attente est considéré, soit comme le résultat d'un déséquilibre de charge, soit comme le temps d'exécution du démon de communication de PVM. La figure 5.13 montre le détail d'une exécution sur 16 processeurs. On observe ainsi plus clairement les différences entre les trois heuristiques de placement initial de la structure de 11 615 atomes.

Pour le placement aléatoire, on observe un important déséquilibre de charge sur les processeurs. La deuxième figure montre le bon équilibre de charge de la méthode de placement quantitatif LPTF. Elle montre également le volume important des communications dans cette méthode. Enfin, pour le placement BPR-FIN, les coûts de communications sont plus faibles, et la charge de calcul est bien équilibrée sur les processeurs.

Notons que même pour le placement BPR-FIN, à partir de 16 processeurs, l'efficacité est inférieure à 70 %. Les communications et le temps d'attente restent importants. Deux raisons expliquent cela :

– les communications en PVM sur le réseau rapide de l'IBM-SP1 restent lentes par

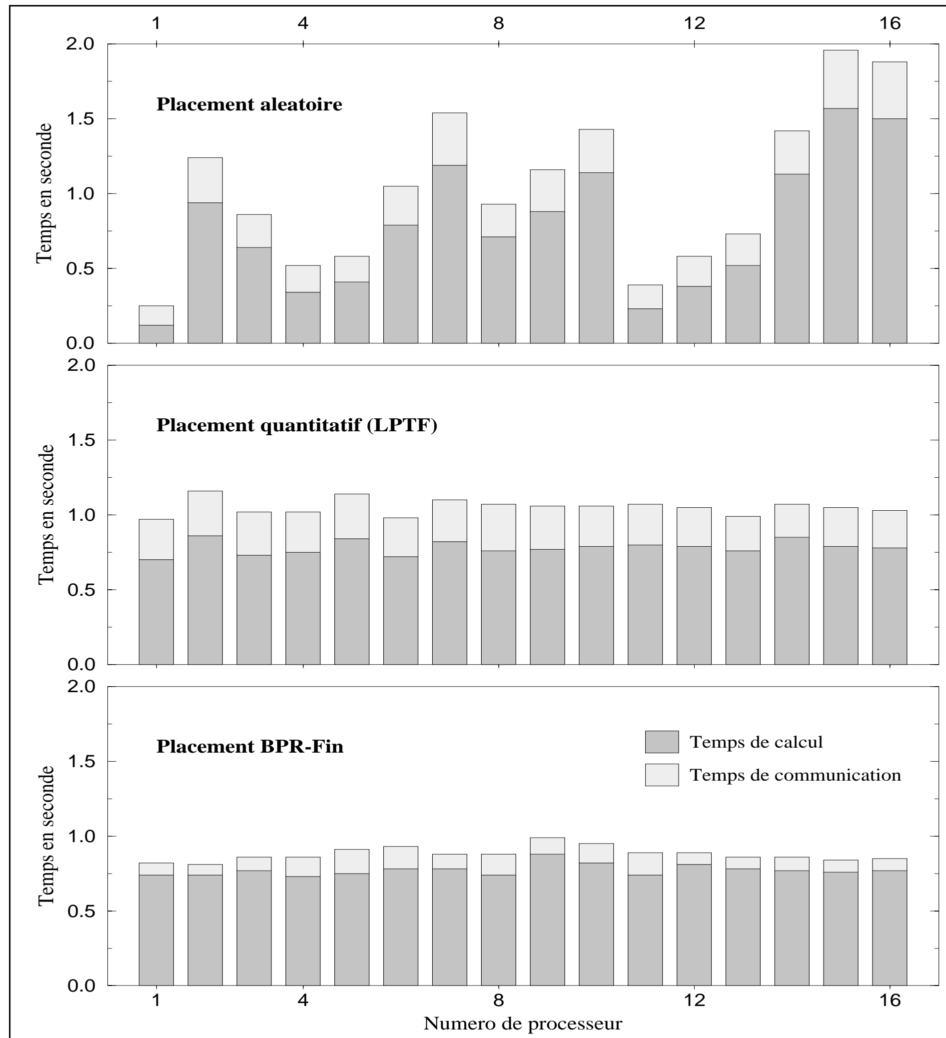


FIG. 5.13: Détail de l'exécution d'une itération de dynamique moléculaire sur 16 processeurs, pour un système de 11 615 atomes.

rapport aux possibilités réelles de la machine ;

- de plus, à partir de 16 processeurs, la structure de 11 615 atomes est un peu petite pour espérer avoir des meilleurs résultats. En effet les communications sont très importantes par rapport aux calculs à effectuer.

La version suivante de ATHAPASCAN-0, écrite sur MPI, résout ce problème de performance des communications en utilisant mieux les possibilités du réseau de communications de l'IBM-SP1 (cf. 6.2).

5.4.4 Comparaison des placements par bi-partition

En utilisant la même méthode que précédemment, nous nous intéressons maintenant aux différences entre les variantes de l'heuristique de placement par bi-partition récur-

sive. La figure 5.14 donne le temps d'exécution moyen des 100 premières itérations de dynamique moléculaire pour les trois variantes de cette heuristique de placement.

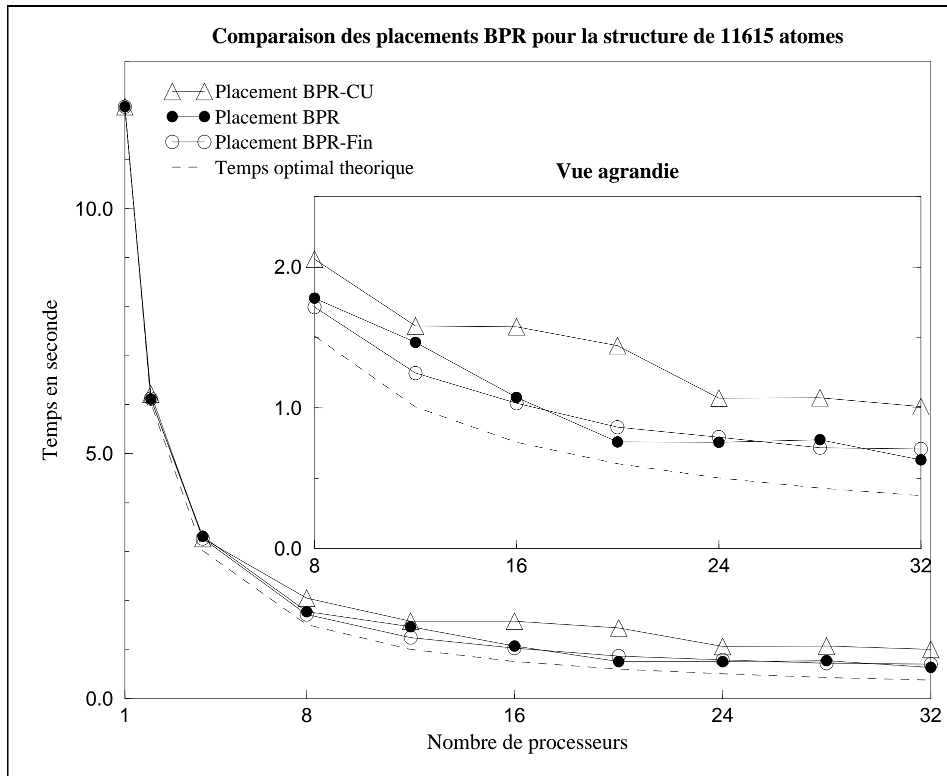


FIG. 5.14: Comparaison des placements par bi-partition récursive.

L'heuristique BPR-CU qui considère que les tâches ont des coûts de calcul uniformes, conduit à un placement mal équilibré. En effet, comme les autres heuristiques de placement par bi-partition récursive, elle permet de réduire le volume des communications. Mais elle ne permet pas d'obtenir un bon équilibre de charge. Utiliser la complexité algorithmique pour évaluer le coût d'exécution des tâches permet réellement d'améliorer l'équilibre de charge.

Il n'existe pas de différences significatives entre les performances obtenues suite à un placement initial par bi-partition récursive simple, et les performances obtenues suite à un placement initial plus fin. Cependant nous verrons, dans le chapitre suivant (cf. 6.3), l'avantage de l'heuristique de placement BPR-FIN par rapport à la version plus simple.

5.4.5 Résultats pour les autres systèmes

Nous avons obtenu des résultats analogues avec des systèmes comportant un nombre différent d'atomes. La figure 5.15 montre les temps d'exécution pour les trois systèmes de tailles différentes qui sont décrits dans le chapitre 4.4. Nous avons utilisé les trois heuristiques de placement principales et un rayon de coupure de 10 Å.

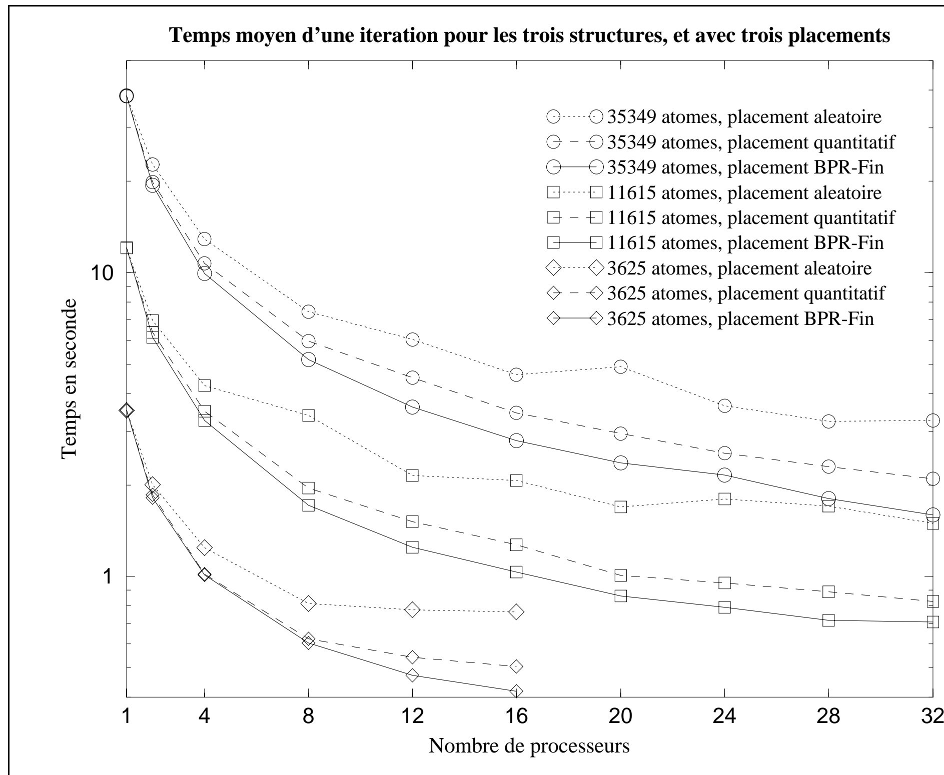


FIG. 5.15: Comparaisons des temps d'exécution pour trois systèmes différents, en utilisant les différentes stratégies de placement.

L'heuristique de placement BPR-FIN permet d'obtenir les meilleures performances, indépendamment de la taille du système et du nombre de processeurs. Avec la structure de 35 349 atomes, on obtient une accélération de 13,6 sur 16 processeurs.

5.5 Influence de la granularité et du rayon de coupure

5.5.1 Modification de la granularité

Dans l'algorithme de découpage géométrique que nous utilisons, la taille des boîtes est directement fonction du rayon de coupure. Comme nous l'avons dit ce découpage permet un accès rapide aux atomes voisins d'un atome donné. Cependant un tel découpage est assez restrictif. En effet, cela limite le nombre de processeurs maximum utilisable au nombre de boîtes du système. Ceci peut empêcher l'étude de petites structures sur beaucoup de processeurs ou l'utilisation d'un gros rayon de coupure. De plus, un tel découpage impose une granularité sur les tâches à placer. Pour remédier à cette limite, on peut découper le système en boîtes de taille plus petites que le rayon de coupure. En effet, si on prend des boîtes dont le coté est la moitié ou le tiers du rayon de coupure,on peut conserver l'accès rapide aux atomes voisins d'un atome donné. Plus précisément, on

prend des boîtes de coté l_k, avec :

$$l_k = \left(\frac{R_{\text{coupure}} + 2D_{\text{tol}}}{k}\right) \qquad \text{avec} \qquad k \in \mathbb{N}^*$$

On obtient simplement les atomes voisins d'un atome donné en considérant les $((2k + 1)^3 - 1)$ boîtes voisines de la boîte de l'atome donné. Dans la suite, k désigne la granularité de découpage du système. Jusqu'à présent, nous avons donné uniquement des résultats pour une granularité simple ($k = 1$), mais conformément aux remarques du premier chapitre (cf. 1.3.2) nous avons écrit un programme acceptant une modification de la granularité. On peut ainsi facilement modifier la granularité en choisissant la valeur de k.

Les heuristiques de placement initial définies précédemment fonctionnent indépendamment de la granularité. Elles ont seulement besoin de l'ensemble des tâches de type T_{ni} et de l'ensemble des tâches de type T_{nv}. La construction de l'ensemble des tâches de type T_{ni} ne pose pas de difficultés. On associe simplement une tâche $T_{\text{ni}}(i)$ à chacune des boîtes i du système. Et, comme pour une granularité simple, l'ensemble des boîtes dépend de la taille du système et de la taille des boîtes l_k.

La construction de l'ensemble des tâches de type T_{nv} est un peu plus compliquée. Les tâches $T_{\text{nv}}(i, j)$ correspondent à l'ensemble des paires de boîtes voisines au sens du rayon de coupure. C'est-à-dire, les paires de boîtes dont la distance minimum entre les deux boîtes de la paire est inférieure à la distance de coupure $R_{\text{coupure}} + 2D_{\text{tol}}$. Ainsi pour construire les tâches $T_{\text{nv}}(i, j)$ on regarde pour chacune des boîtes parmi ses $((2k+1)^3-1)$ boîtes voisines. Comme le montre la figure 5.16, c'est un cube de boîtes de $2k + 1$ boîtes de coté entourant la boîte considérée. Remarquons que dès que $k \geq 3$ toutes les boîtes du cube ne sont pas forcément voisines (au sens du rayon de coupure) de la boîte considérée.

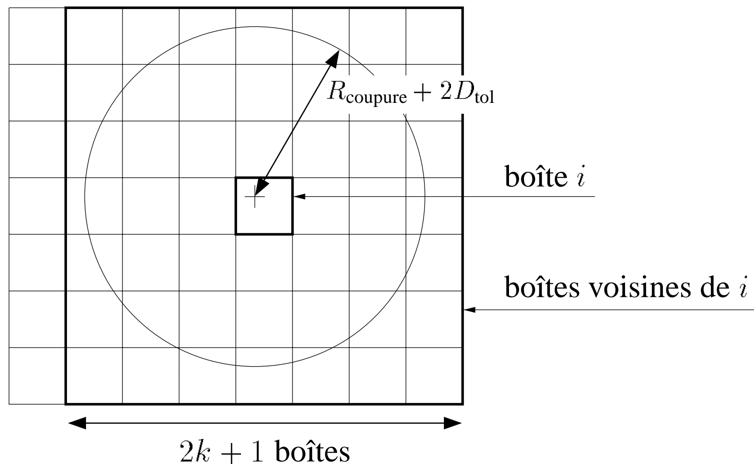


FIG. 5.16: Ensemble des $((2k + 1)^3 - 1)$ boîtes entourant la boîte i et potentiellement voisines de i au sens du rayon de coupure. Vue en deux dimensions et pour $k = 3$.

Pour une granularité k fixée, et pour un système comprenant $\texttt{nbB}_X$, $\texttt{nbB}_Y$, et $\texttt{nbB}_Z$ boîtes suivant respectivement les trois axes X, Y et Z, le nombre de tâches du système

est majoré par :

$$\frac{1}{2}\Bigg[\Big(\mathtt{nbB}_X \cdot \mathtt{nbB}_Y \cdot \mathtt{nbB}_Z \cdot ((2k+1)^3 - 1)\Big) - \Big(2(\mathtt{nbB}_X \cdot \mathtt{nbB}_Y + \mathtt{nbB}_X \cdot \mathtt{nbB}_Z + \mathtt{nbB}_Y \cdot \mathtt{nbB}_Z) \cdot (2k+1)^2 \cdot \tfrac{k(k+1)}{2}\Big) + \Big(4(\mathtt{nbB}_X + \mathtt{nbB}_Y + \mathtt{nbB}_Z) \cdot (2k+1) \cdot (\tfrac{k(k+1)}{2})^2\Big) - \Big(8(\tfrac{k(k+1)}{2})^3\Big) \Bigg]$$

Explication :

Le premier terme représente les $((2k+1)^3 - 1)$ boîtes entourant chacune des boîtes. On enlève le deuxième terme qui représente les boîtes voisines des boîtes appartenant aux faces du pavé qui délimite le système. On rajoute le troisième terme qui représente l'ensemble des boîtes voisines des boîtes appartenant aux arêtes du pavé délimitant le système. En effet, ces boîtes ont été enlevées deux fois par le deuxième terme. Enfin, on enlève le quatrième terme qui représente les boîtes voisines des boîtes appartenant aux huit sommets du pavé délimitant le système. Le facteur de $1/2$ s'explique par symétrie. En effet, nous avons vu que $T_{\mathrm{nv}}(i,j)$ et $T_{\mathrm{nv}}(j,i)$ désignent la même tâche.

Ainsi, pour un système donné, en fonction de la granularité, le nombre de tâches de type T_{ni} croît en $O(k^3)$, et le nombre de tâches de type T_{nv} croît en $O(k^6)$. Dès que l'on augmente k, le nombre de tâches de type T_{nv} augmente rapidement. L'algorithme qui construit l'ensemble des tâches est un algorithme séquentiel, il peut nécessiter beaucoup de mémoire. C'est pourquoi nous avons besoin de cette bonne majoration du nombre de tâches.

5.5.2 Résultats

La figure 5.17 donne la moyenne des temps d'exécution des 100 premières itérations de dynamique moléculaire, en utilisant BPR-FIN comme heuristique de placement initial. Les mesures ont été réalisées avec différents rayons de coupure, pour une granularité normale ($k = 1$), et pour une granularité plus fine ($k = 2$). Les mesures portent toujours sur la structure de 11 615 atomes.

Pour un rayon de coupure de 10 Å, quand on utilise une granularité plus fine, les performances sont moins bonnes qu'avec la granularité normale. En effet, diminuer la granularité augmente le nombre de tâches, et donc le coût nécessaire à la gestion de ces tâches.

En revanche, dès que l'on utilise un rayon de coupure de 15 Å ou de 20 Å, la granularité plus fine permet d'obtenir des meilleures performances qu'avec la granularité normale. Pourquoi, même en séquentiel, avec ces deux rayons de coupure, les résultats sont-ils meilleurs quand on génère plus de tâches ? Le coût de gestion des tâches est compensé par une meilleure approximation des interactions à calculer dans la sphère de coupure de chacun des atomes. Ainsi, plus précisément, avec un rayon de coupure de

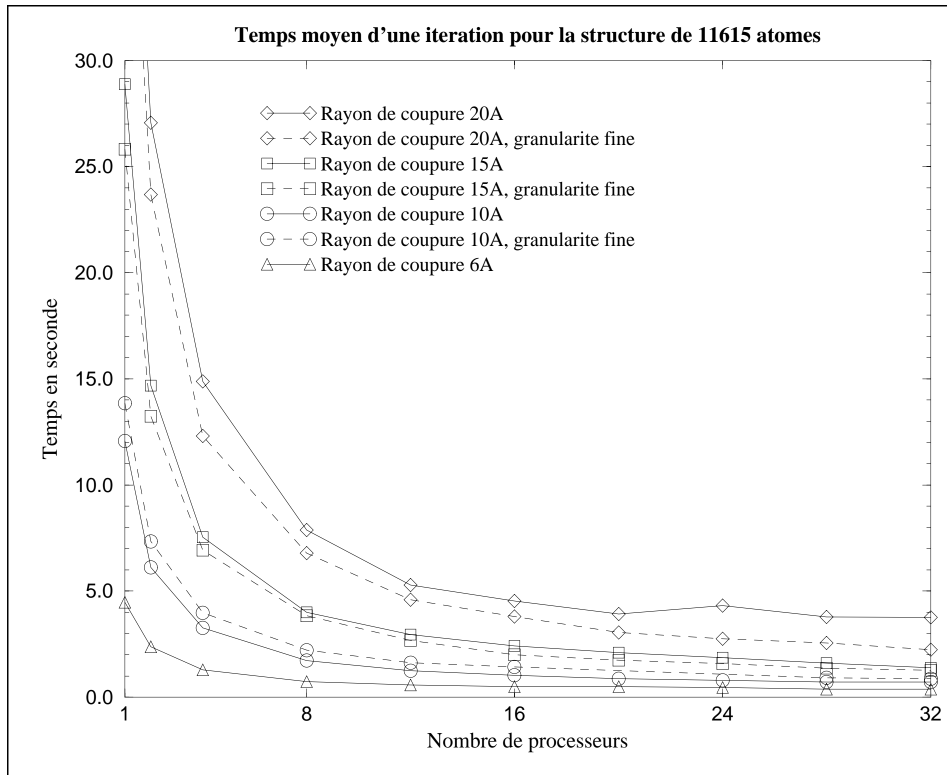


FIG. 5.17: Mesure du temps d'exécution pour différents rayons de coupure et différentes granularités, pour le système de 11 615 atomes.

15 Å et une granularité normale, on calcule les interactions non-liées non nulles sur les paires d'atomes voisines au sens du rayon de coupure. On cherche ces paires parmi 25,824 millions (ce chiffre est la somme des complexités algorithmiques des tâches, qui est aussi le nombre d'interactions parmi lesquelles se trouvent les interactions non nulles). Avec la granularité plus fine, on cherche ces paires non nulles parmi seulement 17,115 millions. Une granularité plus fine offre une meilleure approximation des paires d'interactions non nulles.

Notons que nous avons également fait des mesures avec une granularité encore plus fine ($k = 3$). Mais même pour un rayon de coupure de 20 Å, le coût de gestion des tâches est trop important, une telle granularité est trop fine.

On peut également remarquer que pour un rayon de coupure de 20 Å, avec la granularité normale, les performances se dégradent rapidement au-delà de 16 processeurs. Avec cette granularité, le temps d'exécution ne s'améliore plus beaucoup quand on augmente le nombre de processeurs au-delà de 16. Alors qu'il s'améliore avec une granularité plus fine. Le tableau 5.3 donne précisément les temps d'exécution totale au-delà de 16 processeurs pour les deux granularités.

On observe effectivement une chute rapide de l'efficacité [2] au-delà de 16 processeurs.

[2]Pour calculer l'efficacité, on n'utilise pas le meilleur algorithme séquentiel mais le temps d'exécution de l'algorithme avec un placement sur un processeur unique.

TAB. 5.3: Moyenne des temps d'exécution totale, des temps d'attente et des efficacités pour les 100 premières itérations de dynamique de la structure de 11 615 atomes, avec un rayon de coupure de 20 Å. Et évaluation du déséquilibre de charge pour deux granularités différentes.

granularité normale ($k = 1$)				
Nb. Proc. (p)	1	16	24	32
t_{tot} (s)	52,43	4,54	4,32	3,77
dont t_{att} (s)	0,00	0,99	1,91	1.97
efficacité (%)	100,0	72,2	50,6	43,5
ΔL ($\times 10^3$)	0,0	142,8	858,3	874,3

granularité fine ($k = 2$)				
Nb. Proc. (p)	1	16	24	32
t_{tot} (s)	46,67	3,80	2,76	2,24
dont t_{att} (s)	0,00	0,59	0,63	0,51
efficacité (%)	100,0	76,7	70,4	65,1
ΔL ($\times 10^3$)	0,0	30,6	44,1	34,3

L'inefficacité provient du déséquilibre de charge, comme en témoigne le temps d'attente moyen par itération (t_{att}) et le critère d'équilibre de charge (ΔL). En fait, le nombre de boîtes est trop faible (seulement 36 boîtes et 227 tâches de type T_{nv}) pour obtenir un bon placement au-delà de 16 processeurs avec une granularité normale. En revanche, en diminuant la granularité, le nombre de tâches est suffisant (252 boîtes et 8 226 tâches de type T_{nv}) pour permettre à l'heuristique BPR-FIN d'obtenir un bon placement.

Évidemment, pour cet exemple, il est de toute façon meilleur d'utiliser une petite granularité. Mais pour un système plus petit, comme la structure de 3 615 atomes, diminuer la granularité peut permettre d'utiliser plus de processeurs efficacement. Ainsi, si on veut reprendre les mêmes paramètres que notre simulation test du paragraphe 5.1.3, c'est-à-dire, si l'on souhaite simuler la petite structure sur 24 processeurs, en utilisant un rayon de coupure de 15 Å, il faut utiliser une granularité plus fine. Comme le montrent les mesures sur un processeur du tableau 5.4, le coût de gestion des tâches n'est pas compensé par une meilleure approximation de la sphère de coupure. Sur cet exemple, il est préférable d'utiliser une granularité normale. Mais sur 24 processeurs, une granularité plus fine est préférable car elle permet d'obtenir un meilleur équilibre de charge.

TAB. 5.4: Moyenne des temps d'exécution totale, pour les 100 premières itérations de dynamique de la structure de 3 625 atomes, avec un rayon de coupure de 15 Å.

Nb. Proc. (p)	1	24
Granularité normale ($k = 1$)	7,18 s	0,96 s
Granularité fine ($k = 2$)	7,25 s	0,63 s

En conclusion, diminuer la granularité augmente le coût de gestion des tâches, mais améliore l'approximation de la sphère de coupure. Il est donc intéressant de diminuer la granularité quand on utilise un grand rayon de coupure. Même si une meilleure approximation de la sphère de coupure ne permet pas de compenser le coût de gestion des tâches, utiliser une granularité plus fine permet d'utiliser efficacement plus de processeurs.

5.6 Conclusion

Nous avons montré l'intérêt d'utiliser la complexité algorithmique des tâches de calcul pour évaluer le coût des tâches et obtenir ainsi un placement bien équilibré. Nous avons également montré que les placements par bi-partition récursive qui possèdent des bonnes propriétés de localité permettent également d'obtenir des bons équilibres de charge. Ils permettent d'obtenir un bon compromis entre l'équilibre de charge et la minimisation des communications. De plus ils ne nécessitent pas de modélisation des communications, le placement reste indépendant de la machine-cible.

Bien que meilleur, l'équilibre de charge n'est pas parfait. Nous pouvons avancer trois explications :

- nous utilisons des heuristiques de placement qui ne fournissent pas des placements optimaux ;
- même avec un placement optimal, il serait sans doute très difficile de faire mieux. En effet, nous utilisons une estimation grossière du coût des tâches de calcul, mais une estimation plus précise est très difficile. Une connaissance plus précise du nombre d'interactions non nulles n'est pas suffisante, le nombre d'opérations nécessaire au calcul d'une interaction dépend de la distance entre les deux atomes de l'interaction (cf. chapitre 4.2.1) ;
- enfin, la simulation est dynamique. Les atomes bougent et le nombre des interactions à calculer varie au cours de la simulation. Même si ces variations sont faibles, un placement initial statique ne peut rendre compte de cette dynamicité.

Nous pensons cependant qu'il est encore possible d'améliorer les performances notamment en utilisant un ajustement dynamique du placement, pour améliorer le placement initial. En effet, il ne faut pas perdre de vue que pour ce type d'application l'étude de systèmes réels nécessite de calculer plusieurs milliers d'itérations. La meilleure utilisation possible des ressources de calcul d'une machine est primordiale.

Chapitre 6

Ajustement dynamique

Dans ce chapitre, nous montrerons comment nous avons tiré avantage de la multiprogrammation dans notre application. Nous étudierons ensuite un mécanisme d'ajustement dynamique de la charge. Nous verrons comment il arrive à corriger les imperfections du placement statique initial en introduisant un minimum de sur-coût de calculs et de communications. Nous donnerons ensuite l'algorithme qui permet de déplacer les atomes entre les processeurs, en fonction de leur mouvement dans l'espace de simulation. En particulier, nous insisterons sur l' interaction entre cet algorithme et le mécanisme d'ajustement dynamique de la charge. Nous terminerons par une étude de l'extensibilité de notre algorithme et par des mesures sur le CRAY-T3E.

6.1 Ordonnancement entrelacé

Dans le chapitre 2 (cf. 2.2), nous avons écrit que la multiprogrammation des noeuds de calcul d'une machine parallèle permet d'entrelacer l'utilisation des ressources du noeud. On peut ainsi garder le processeur actif le plus longtemps possible pour les calculs directement utiles à l'application. Pour exploiter la multiprogrammation, nous savons qu'il faut procéder en deux étapes :

- trouver un ordonnancement compatible avec l'utilisation entrelacée du réseau et des processeurs. Nous parlerons également de recouvrement des communications par des calculs ;
- exploiter cet entrelacement grâce à l'utilisation des processus légers.

Nous allons expliquer, pour notre application de dynamique moléculaire, comment construire un ordonnancement compatible avec l'utilisation concurrente du réseau et des processeurs. Nous détaillerons ensuite comment exploiter cet ordonnancement grâce à l'environnement ATHAPASCAN.

6.1.1 Recouvrement des communications

Recouvrement de l'échange des coordonnées et de l'échange des forces

La figure 6.1 rappelle, en utilisant les notations du chapitre 5 (cf 5.2), les phases de calculs et de communications d'une itération de dynamique moléculaire sur un processeur. Il faut donc chercher à recouvrir l'échange des coordonnées et l'échange des forces (étapes 1 et 3).

1	***Échanger*** *les coordonnées des atomes*
2	***Calculer*** *les forces (tâches : $T_{ni}(i)$, $T_{nv}(i,j)$ et $T_g(i)$)*
3	***Échanger*** *les forces*
4	***Intégrer*** *l'équation du mouvement (tâches : $T_i(i)$)*

FIG. 6.1: Phases de calculs et de communications d'une itération de dynamique moléculaire.

Rappelons également que la parallélisation est issue d'une décomposition du domaine de simulation sur les processeurs. Donc, comme dans l'exemple du chapitre 2 (cf. 2.3), pour recouvrir les phases de communications, on cherche les tâches qui utilisent uniquement des données locales au processeur.

Plus précisément, on regroupe, sur chacun des processeurs, les tâches de la phase 2 en fonction des données dont elles ont besoin et des résultats qu'elles produisent. Ainsi pour un processeur p, on a les groupes suivants :

- $G_{\text{nv}}(p,q)$ ensemble des tâches $T_{\text{nv}}(i,j)$ placées sur le processeur p et qui ont besoin des données des boîtes placées sur le processeur q. Plus précisément, il s'agit de l'ensemble des tâches $T_{\text{nv}}(i,j)$ du processeur p dont une des deux boîtes associées (boîtes i ou j) est placée sur q (Bien sur, conformément aux contraintes de placement du chapitre précédent, l'autre boîte associée à $T_{\text{nv}}(i,j)$ est sur p.). Ces groupes correspondent aux tâches de calcul de la frontière entre p et chacun des autres processeurs. Attention, il n'y a pas de symétrie, ici $G_{\text{nv}}(p,q) \neq G_{\text{nv}}(q,p)$;
- $G_{\text{l}}(p)$ ensemble des tâches $T_{\text{ni}}(i)$ et $T_{\text{nv}}(i,j)$ dont les boîtes associées sont sur p. Ce groupe correspond aux calculs sur le domaine intérieur, il ne nécessite pas de données des autres processeurs. Ce sont les calculs de ce groupe qui permettent de recouvrir l'échange des coordonnées et l'échange des forces ;
- $G_{\text{g}}(p)$ ensemble des tâches $T_{\text{g}}(i)$ placées sur le processeur p. Cet ensemble utilise les données des boîtes placées sur le processeur p et les données des boîtes voisines de celles de p ;
- $G_{\text{i}}(p)$ ensemble des tâches $T_{\text{i}}(i)$ de l'intégration courante placées sur le processeur p. De même que dans le chapitre 5, on note $G_{\text{i-}}(p)$ ce groupe de tâches quand il s'agit de l'itération précédente.

La figure 6.2 montre la dépendance entre les groupes de tâches du processeur P_2 avec les groupes de tâches des processeurs P_1 et P_3 qui possèdent chacun des boîtes voisines

de celles de P_2. Pour distinguer les groupes de tâches qui communiquent avec les autres processeurs, nous avons isolé l'échange des coordonnées et l'échange des forces dans les nouveaux groupes de tâches suivants :

- $EC(p,q)$ Envoi des Coordonnées des atomes de p placés dans les boîtes voisines des boîtes du processeur q. Ces données sont nécessaires au processeur q pour le calcul des forces non-liées de $G_{\text{nv}}(q,p)$ et pour le calcul des forces géométriques de $G_{\text{g}}(q)$;
- $RC(q,p)$ Réception sur le processeur q des Coordonnées des atomes du processeur p. C'est la réception sur q des données envoyées par la tâche $EC(p,q)$ depuis p ;
- $EF(q,p)$ Envoi des Forces qui s'exercent sur les atomes de p et qui ont été calculées sur le processeur q. Ce sont les atomes dont q a reçu les coordonnées par $RC(q,p)$. C'est-à-dire les atomes de p placés dans les boîtes voisines des boîtes du processeur q ;
- $RF(p,q)$ Réception des Forces qui s'exercent sur les atomes de p et qui ont été calculées sur q. C'est la réception sur p des forces envoyées par $EF(q,p)$ depuis le processeur q.

Rappelons que pour diminuer le volume des communications, nous les avons regroupées (cf. 5.3.5). Ces nouveaux groupes de tâches permettent également de mettre cette étape en évidence.

Si on masque les groupes qui calculent les forces géométriques (les groupes de type G_{g}), les dépendances entre les groupes de tâches restants sont très semblables à celles de l'exemple du chapitre 2 (cf. 2.3). On voit ainsi clairement, que la dépendance de données permet le calcul des tâches locales du groupe G_{l}, en concurrence avec les communications nécessaires à l'échange des coordonnées et l'échange des forces. On a également intérêt à exécuter concurremment les groupes G_{nv} relatifs à des frontières différentes. Ainsi sur notre figure, le calcul de $G_{\text{nv}}(2,1)$ peut contribuer au recouvrement des communications de $EC(2,1)$, $RF(2,1)$, $EC(2,3)$, $RC(2,3)$, $EF(2,3)$, et $RF(2,3)$.

Recouvrement du contrôle de la simulation

Nous avons vu dans le chapitre 3 que pour contrôler le bon déroulement de la simulation après chaque itération, on calcule les énergies correspondantes à chacun des types de forces. On calcule également l'énergie cinétique pour connaître la température globale du système. Enfin, dans le cas d'une dynamique de NEWTON, on doit contrôler la température (cf. 3.6.3) grâce à un facteur de rajustement des vitesses qui dépend de la température courante et de la température-cible du système.

Les énergies sont calculées localement sur chacun des processeurs esclaves, puis communiquées au processeur maître à la fin de l'intégration. Le processeur maître somme ensuite les énergies reçues de chacun des processeurs esclaves pour avoir les termes globaux au système. Le facteur de rajustement des vitesses est calculé sur le processeur maître, puis diffusé ensuite à chacun des processeurs esclave pour être utilisé au début de l'intégration suivante. Il est ainsi possible de recouvrir la communication des informations de

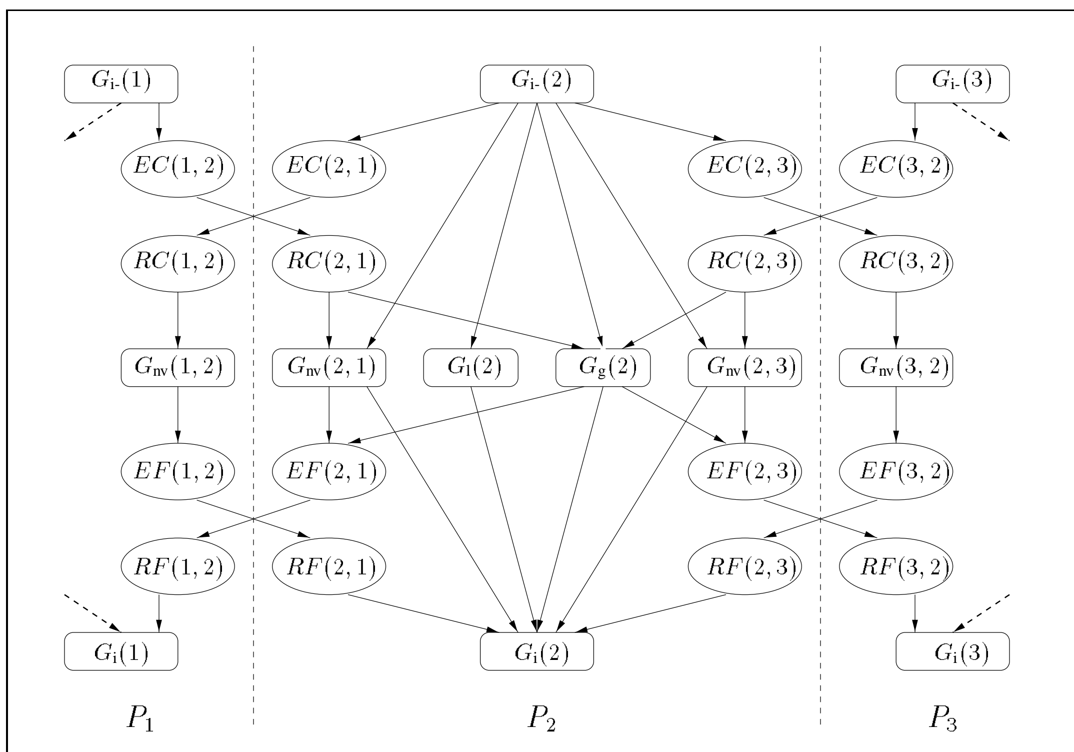


FIG. 6.2: Graphe de précédences d'une itération de dynamique moléculaire après regroupement des tâches en fonction du placement et de la dépendance des données.

contrôle par le calcul des forces non-liées, et d'éviter ainsi de synchroniser les processeurs esclaves.

6.1.2 Expression en ATHAPASCAN-0A

Nous allons nous concentrer sur l'expression de l'itération de dynamique moléculaire. Rappelons d'abord qu'en ATHAPASCAN-0A le parallélisme s'exprime par des appels de procédures distantes : les services (cf. 2.2.3). Les communications entre les processeurs sont encapsulées dans le mécanisme de passage des paramètres et des résultats aux procédures distantes. Les services placés sur un même processeur peuvent partager des données, mais ils doivent garantir la cohérence des accès en utilisant des sémaphores.

L'affectation des groupes de tâches aux services est fonction des communications, et de la façon d'entrelacer ces communications avec les calculs. Ainsi, sur chacun des processeurs esclaves, nous avons plusieurs types de services. Plus précisément, sur un processeur p, nous avons les services suivants :

– **le service de contrôle**, créé par le processeur maître sur chacun des processeurs esclaves et à chacune des itérations, il est chargé du contrôle de la simulation. Plus précisément, il communique la température cible de la simulation et le rapport pour ajuster les vitesses des atomes sur chacun des processeurs. Les résultats des services de contrôle permettent de combiner sur le processeur maître l'énergie totale

de l'ensemble du système. Ils permettent également de regrouper les informations sur la charge des processeurs esclaves ;

- **le service principal**, créé au début de la simulation par le processeur maître, et sur chacun des processeurs esclaves, il contrôle localement l'itération de dynamique moléculaire. Il assure la cohérence des données partagées en synchronisant les différents services. Il intègre les équations du mouvement des atomes (groupe $G_{\mathrm{i}}(p)$) et calcule les forces des interactions géométriques (groupe $G_{\mathrm{g}}(p)$) quand toutes les coordonnées des atomes des boîtes voisines sont arrivées. Il crée également les services de calcul des forces non-liées inter-processeur sur ces processeurs voisins (groupes $EC(p,\cdot)$) et reçoit de ces services les forces calculées (groupes $RF(p,\cdot)$) ;
- **le service de calcul des forces locales**, créé localement à chacune des itérations par le service principal, il est chargé de calculer les forces d'interactions non-liées locales au processeur p (groupe $G_{\mathrm{l}}(p)$) ;
- **les services de calcul des forces inter-processeurs**. Ce modèle de service est le seul à avoir plusieurs instances en même temps sur un processeur. Ainsi il y a autant d'instances de ce service que de processeurs voisins[1] de p. Ces services permettent de recevoir les coordonnées des atomes des processeurs voisins (groupes $RC(p,\cdot)$), de calculer les forces non-liées inter-processeur (groupes $G_{\mathrm{nv}}(p,\cdot)$) et de retourner les forces ainsi calculées (groupes $EF(p,\cdot)$).

Bien que cette version de ATHAPASCAN ne dispose pas de priorités dans sa politique d'auto-ordonnancement, nous les avons simulées, afin de donner une priorité moindre au service de calcul des forces locales.

La figure 6.3 montre l'activité des services en fonction du temps, pour 4 itérations et sur 4 processeurs esclaves. Nous avons utilisé la structure de 11 615 atomes (cf. 4.4) et un placement initial BPR-FIN (cf. 5.3). Le processeur maître n'est pas représenté. Les traits horizontaux en pointillé séparent les services des différents processeurs. Les traits verticaux permettent, individuellement pour chacun des processeurs, de séparer les itérations. Les rectangles indiquent les différents services, de leur création à leur terminaison. Les services sont classés suivant leur type, et toujours dans le même ordre, comme c'est indiqué pour le processeur P_2. Soit ils sont actifs (en noir), lorsqu'ils calculent ou lorsqu'ils communiquent. Soit ils sont bloqués (en blanc) sur un sémaphore ou sur l'attente d'un message. À part le service principal, les services sont créés à chacune des itérations.

Cette trace d'exécution comporte beaucoup d'informations. Elle permet d'observer l'entrelacement des processus légers qui exécutent les services. Globalement, les processeurs sont toujours actifs. C'est-à-dire qu'il y a toujours au moins un service actif sur un processeur. Ceci n'est pas surprenant, c'est conforme à nos conclusions du chapitre précédent sur la bonne qualité du placement BPR-FIN.

En observant plus finement, on peut également remarquer les situations suivantes sur les services d'un processeur p:

- le service principal est actif quatre fois pendant une itération :

[1] processeurs voisins au sens du rayon de coupure, c'est-à-dire les processeurs ayant des boîtes voisines des boîtes de p.

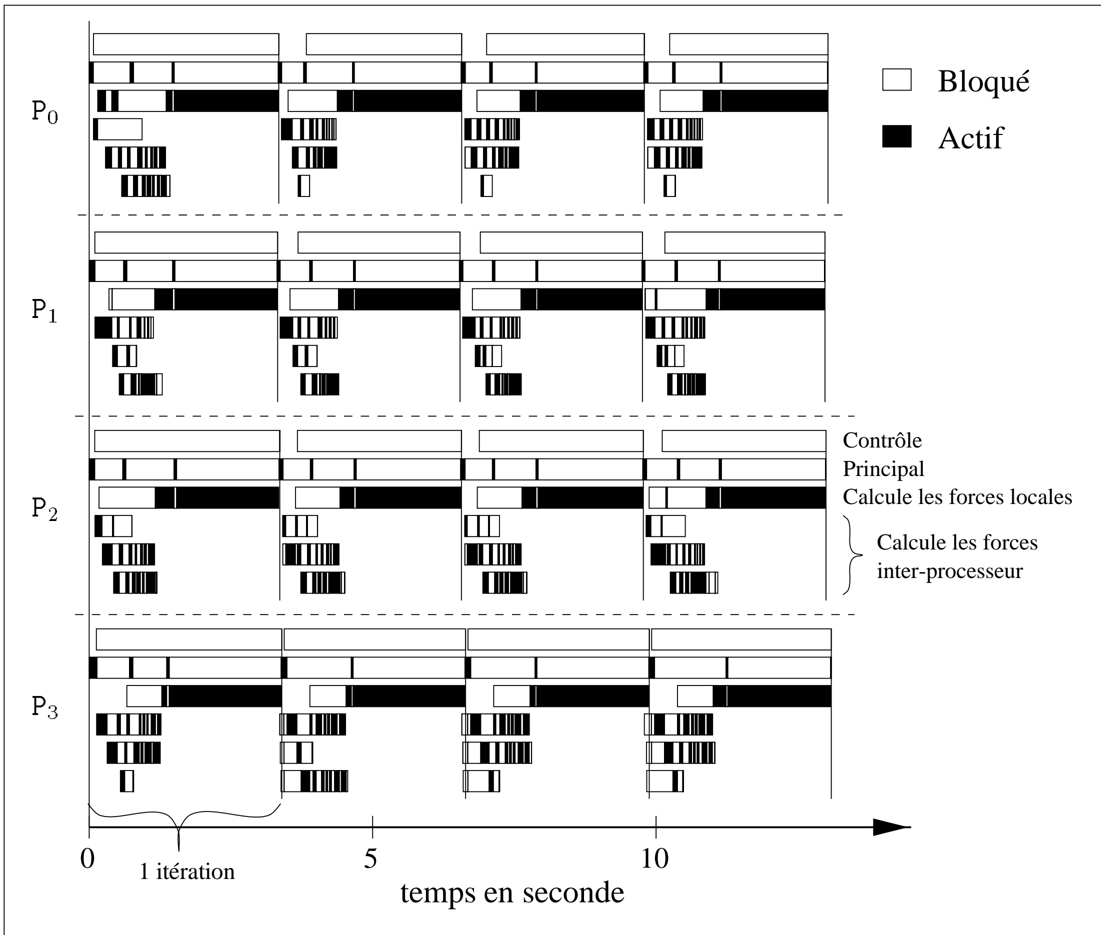


FIG. 6.3: Diagramme de GANTT sur 4 processeurs de 4 itérations, sous ATHAPASCAN-0A pour le placement BPR-FIN.

1. au début de l'itération, pour lancer les services de calcul des forces non-liées inter-processeur (groupes $EC(p, \cdot)$) ;
2. après l'arrivée localement de tous les services de calcul des forces non-liées inter-processeurs. Les coordonnées des atomes des boîtes voisines de celles de p sont toutes arrivées. Donc, on peut calculer les forces des interactions géométriques (groupe $G_{\mathrm{g}}(p)$) ;
3. pour la réception des forces calculées par les autres processeurs (groupes de tâches $RF(p, \cdot)$) ;
4. à la fin de l'itération, pour l'intégration des équations du mouvement des atomes de p (groupe $G_{\mathrm{i}}(p)$).

– le service de calcul des forces locales s'exécute essentiellement à la fin de l'itération car il est moins prioritaire. Mais si les services de calcul des forces inter-processeurs ne sont pas créés tout de suite au début de l'itération, le service de calcul des forces locales prend la main pour combler l'attente des communications, comme le montre

la première itération sur le premier processeur. L'auto-ordonnancement des processus légers permet ici de garder le processeur actif en modifiant automatiquement l'ordre de calcul des tâches ;

- les services de calcul des forces inter-processeur partagent le temps du processeur de façon coopérative.
- Nous pouvons également observer un léger décalage de la fin des itérations sur chacun des processeurs. En effet sur le processeur P_3, pour les trois dernières itérations, les services de calcul des forces inter-processeur sont créés avant la fin de l'itération locale, ceci n'est possible que si ce processeur et en retard par rapport aux autres.

Pour démontrer la souplesse qu'offre l'auto-ordonnancement des processus légers, associé à une description entrelacée de l'utilisation des ressources, la figure 6.4 montre une trace d'exécution pour 3 itérations et toujours pour 4 processeurs, mais avec un placement initial aléatoire (RND).

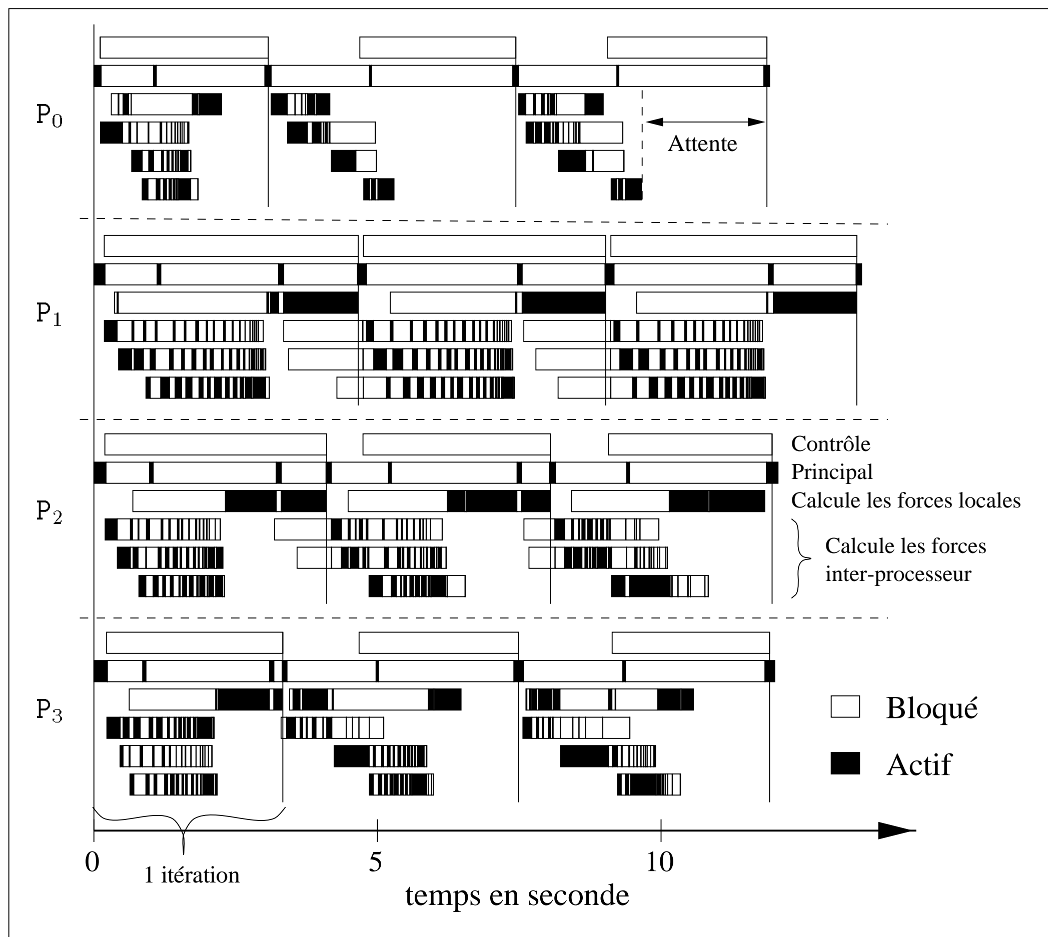


FIG. 6.4: Diagramme de GANTT sur 4 processeurs de 3 itérations, sous ATHAPASCAN-0A pour un placement aléatoire.

Comme nous l'avons écrit dans le chapitre précédent, pour ce type de placement, la charge de calcul est mal équilibrée. On peut ainsi mieux voir comment l'auto-ordonnancement des processus légers s'adapte pour garder le processeur actif le plus possible. On observe les situations suivantes :

- le processeur P_1 est le plus chargé, il travaille en permanence et termine le dernier. Le calcul des forces locales autorise ce retard par rapport aux autres processeurs. On peut également remarquer la coopération entre les processus légers qui exécutent les services de calcul des forces inter-processeur ;
- inversement le processeur P_0 est très peu chargé, il n'est pas actif en permanence. Ainsi pour intégrer les équations du mouvement, il doit attendre, à la fin de l'itération, que le calcul des forces inter-processeurs soit terminé sur ces processeurs voisins ;
- le processeur $P2$, est moins chargé que le processeur P_1 cependant il reste actif pendant les trois premières itérations. Ici, la modification de l'ordre du calcul des forces inter-processeur suffit à garder le processeur actif ;
- le dernier processeur, bien que sous-chargé, arrive à rester actif pendant la première itération.

L'auto-ordonnancement des processus légers, associé à une description entrelacée de l'utilisation des ressources, autorise un léger décalage entre les processeurs dans le calcul des itérations, tout en les gardant actifs. Concrètement, un placement initial BPR-FIN assez bien équilibré se traduit par une phase transitoire de quelques itérations pendant laquelle les processeurs se décalent tout en restant actifs. Nous reviendrons sur cet aspect dans la suite, notamment avec la figure 6.6.

6.1.3 Expression en ATHAPASCAN-0B

Comme nous l'avons écrit dans le chapitre 2 (cf. 2.2.4), dans cette version de la plate-forme d'exécution du projet APACHE, l'expression du parallélisme par des appels de procédures distantes a été abandonnée au profit d'une expression par processus légers communicants. Outre une meilleure efficacité sur notre plate-forme d'expérimentation (le IBM-SP1 de l'IMAG), cette version offre une expressivité plus importante.

L'affectation des groupes de tâches aux processus légers est très proche de la version précédente. Nous avons profité de la plus grande expressivité pour créer les processus légers une seule fois au début du programme, et pour placer l'envoi des coordonnées et la réception des forces sur des processus légers indépendants. Ainsi sur un processeur esclave p, nous avons les processus légers suivants :

- **le thread de contrôle**, qui communique les informations de contrôle de la simulation entre le processeur maître et les esclaves ;
- **le thread principal**, qui s'occupe de la gestion locale des accès concurrents aux données entre les processus légers. Il intègre également les équations du mouvement (groupe $G_{\mathrm{i}}(p)$) et calcule les forces des interactions géométriques (groupe $G_{\mathrm{g}}(p)$) ;

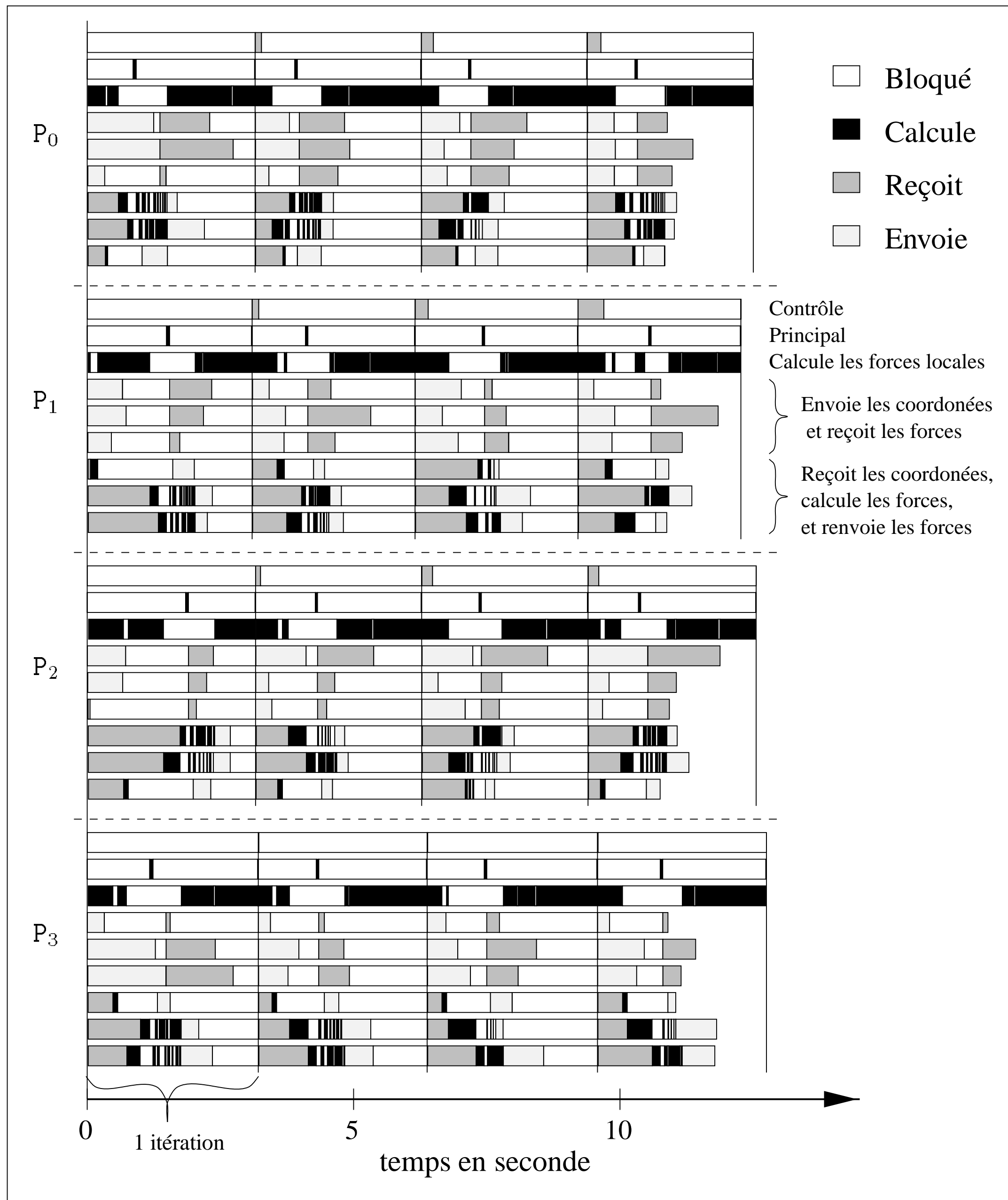


FIG. 6.5: Diagramme de GANTT sur 4 processeurs de 4 itérations, sous ATHAPASCAN-0B pour le placement BPR-FIN.

– **le thread de calcul des forces locales**, il calcule les forces non-liées entre les atomes locaux de p (groupe $G_l(p)$) ;

– **les threads d'envoi des coordonnées et de réception des forces**. Il existe sur le processeur p, une instance de ce type de thread pour chacun des processeurs voisins (au sens du rayon de coupure) de p. Ces threads envoient les coordonnées des atomes (groupes de tâches $EC(p, \cdot)$) et reçoivent les forces calculées (groupes

$RF(p, \cdot)$) des processeurs voisins ;
- **les threads de calcul des forces inter-processeurs**. Il existe également autant d'instances de ce type de processus légers que de processeurs voisins de p. Ils sont chargés de la réception des coordonnées des atomes (groupes $RC(p, \cdot)$), du calcul des forces non-liées inter-processeur (groupes $G_{\text{nv}}(p, \cdot)$) et du renvoi des forces ainsi calculées (groupes $EF(p, \cdot)$).

Nous avons affecté une priorité élevée au processus principal qui gère la simulation. Les processus légers qui font des communications (à savoir : le thread de contrôle, les processus légers qui envoient les coordonnées et reçoivent les forces, et les threads qui reçoivent les coordonnées et renvoient les forces calculées) ont des priorités intermédiaires. Enfin, le processus léger qui calcule les forces locales, a la priorité la plus basse. Son exécution n'est pas prioritaire, car, pour l'auto-ordonnancement, il sert à recouvrir les communications des autres threads.

La figure 6.5 montre l'activité des différents threads en fonction du temps, pour 4 itérations et sur 4 processeurs esclaves. Nous avons également utilisé la structure de 11 615 atomes (cf. 4.4) et un placement initial BPR-FIN (cf. 5.3).

Les processus légers sont classés suivant leur type, et toujours dans le même ordre, comme c'est indiqué pour le processeur P_1. La couleur indique l'activité des threads :
- en noir, lorsqu'ils calculent ;
- en blanc, lorsqu'ils sont bloqués sur un sémaphore ;
- en gris clair, lorsqu'ils envoient des données dans un tube de communication ;
- en gris foncé, lorsqu'ils reçoivent des données d'un tube de communication ;

Attention, pour communiquer les données entre les processeurs, on utilise des tubes de communication (cf. 2.2.4). Ils découpent l'envoi des données en une suite de messages bloquant de taille fixe, et exercent un contrôle de flux entre l'émetteur et le récepteur. Les états de communication ne représentent pas le temps de communication mais marquent l'utilisation d'un tube de communication, dont l'exécution est entrelacée avec les calculs des autres processus légers. Ainsi, un processeur est inactif si aucun de ses threads ne calcule.

Cette trace montre parfaitement l'entrelacement entre les phases de communications et les phases de calculs. On peut, comme précédemment observer que le léger déséquilibre de charge décale progressivement les processeurs entre eux. Mais l'auto-ordonnancement des processus légers permet de les garder actifs. Bien sûr cette situation est transitoire, le retard d'un processeur par rapport à ses voisins ne peut pas être supérieur au temps de calcul de ses forces locales.

6.2 Comparaison entre les bibliothèques de communications

Nous cherchons à vérifier que le coût de gestion des processus légers n'est pas trop important. La thèse de Michel CHRISTALLER [33] présente déjà de nombreuses mesures,

pour évaluer l'impact de la couche ATHAPASCAN-0A. Notamment, elle présente des mesures sur les performances des communications par rapport à PVM (la bibliothèque de communications de la base de cette version). De même, la thèse de Ilan GUINZBURG [70] présente des mesures analogues pour évaluer ATHAPASCAN-0B, la deuxième version du support d'exécution.

Dans ce paragraphe, nous avons plutôt cherché à savoir quel était l'impact global de la multiprogrammation sur les performances de notre application. Ainsi le tableau 6.1 compare les temps d'exécution moyens d'une itération de dynamique moléculaire en utilisant PVM seule et en utilisant les deux versions d'ATHAPASCAN. La version PVM utilise des communications non-bloquantes mais l'ordonnancement des tâches est statique. C'est-à-dire que, contrairement aux versions multiprogrammées, l'ordonnancement ne dépend pas de l'avancement des communications. Nous avons écrit dans le chapitre 2 (cf.2.2.2 et 2.3), qu'il est possible, à l'aide des primitives de communication non-bloquantes, de décrire un ordonnancement dynamique entrelaçant l'utilisation du réseau et des processeurs. Mais la réalisation d'un tel mécanisme passe par la difficile construction d'un automate qui tienne compte de l'avancement des communications.

TAB. 6.1: Moyenne sur les premières itérations des temps d'exécution totale d'une itération pour la structure de 11 615 atomes. Comparaisons des librairies de communications, pour un placement initial de type BPR-FIN.

		Temps moyen d'une itération					
Nb. Proc.	(p)	1	2	4	8	16	24
PVM	(s)	11,84	6,19	3,30	1,68	0,99	0,81
ATHAPASCAN-0A	(s)	11,84	6,12	3,27	1,68	0,98	0,79
ATHAPASCAN-0B	(s)	11,84	6,03	3,11	1,61	0,97	0,75

Les temps d'exécution obtenus en utilisant ATHAPASCAN-0A sont très proches de ceux obtenus avec PVM. Grâce à l'ordonnancement dynamique offert par ATHAPASCAN-0A, ils sont mêmes légèrement meilleurs. Pour notre application, la souplesse de la multiprogrammation compense complètement son léger sur-coût. La version ATHAPASCAN-0B donne des meilleurs résultats que la première version. En effet cette dernière version est écrite au-dessus de MPI, bibliothèque de communications qui exploite mieux les possibilités du réseau de l'IBM-SP1 que PVM.

6.3 Ajustement dynamique

Nous venons de voir que l'auto-ordonnancement des processus légers autorise un léger déséquilibre de charge entre les processeurs, sans perte d'efficacité. Mais le temps de calcul des itérations est stable d'une itération à l'autre. Ainsi, pendant une phase transitoire de quelques itérations, les processeurs se décalent jusqu'à la limite permise par la dépendance des données. Ensuite, les processeurs les moins chargés (donc les plus en avance) attendent les plus chargés, en conservant la même avance.

La partie gauche de la figure 6.6 montre ce phénomène transitoire de décalage. Le graphique en haut à gauche donne, en fonction des itérations, le retard pris, par chacun des huit processeurs, par rapport à la date moyenne de fin d'itération (moyenne sur les huit processeurs). Le graphique en bas à gauche indique le temps total moyen (t_{tot}) de l'itération courante. La partie en gris correspond au temps d'inactivité moyen (t_{att}) des processeurs pendant l'itération . Les mesures présentées dans ce paragraphe sont faites en utilisant ATHAPASCAN-0B.

Sur cette figure, les 20 premières itérations correspondent à la phase transitoire ; les processeurs se décalent progressivement mais restent actifs. À l'itération 20, le programme atteint le décalage maximum possible entre le processeur le plus chargé et le processeur le moins chargé. Ce décalage est d'environ 3 secondes soit environ 60 % du temps moyen de l'itération. Au-delà de la vingtième itération, le processeur le plus en avance attend le processeur le plus en retard. Ceci se traduit par une augmentation du temps d'attente moyen (t_{att}) et par conséquent par une augmentation équivalente du temps total moyen de calcul d'une itération (t_{tot}). Le temps moyen de calcul d'une itération augmente ensuite progressivement au fur et à mesure que les processeurs les moins chargés atteignent le décalage maximum possible.

Le but du mécanisme de régulation de charge est d'éviter qu'un processeur attende. On cherche ainsi à utiliser toute la surface de calcul. Pour satisfaire cela, le mécanisme de régulation de charge va déplacer la charge de calcul entre les processeurs pour compenser d'une itération à l'autre le retard ou l'avance pris par un processeur. Mais attention, comme nous l'avons écrit dans le chapitre 1 (cf. 1.3.5), les mécanismes d'équilibre de charge augmentent le volume des communications et le coût des calculs. Pour être efficace, il doit introduire un sur-coût inférieur au coût du déséquilibre de charge. Nous avons donc cherché un mécanisme de régulation qui introduise un sur-coût de communications et de calculs minimum.

6.3.1 Principe

Le temps d'exécution des tâches de calcul des forces non-liées est assez stable d'une itération à l'autre. En effet, les atomes bougent peu et donc le nombre des interactions à calculer ne varie pas beaucoup d'une itération à l'autre. De plus, rappelons que pour le moment, nous ne tenons toujours pas compte du déplacement des atomes sur les processeurs. Pour réguler la charge, nous allons déplacer les tâches de calcul des forces non-liées, en fonction du retard ou de l'avance pris par les processeurs par rapport à la date moyenne de fin de l'itération.

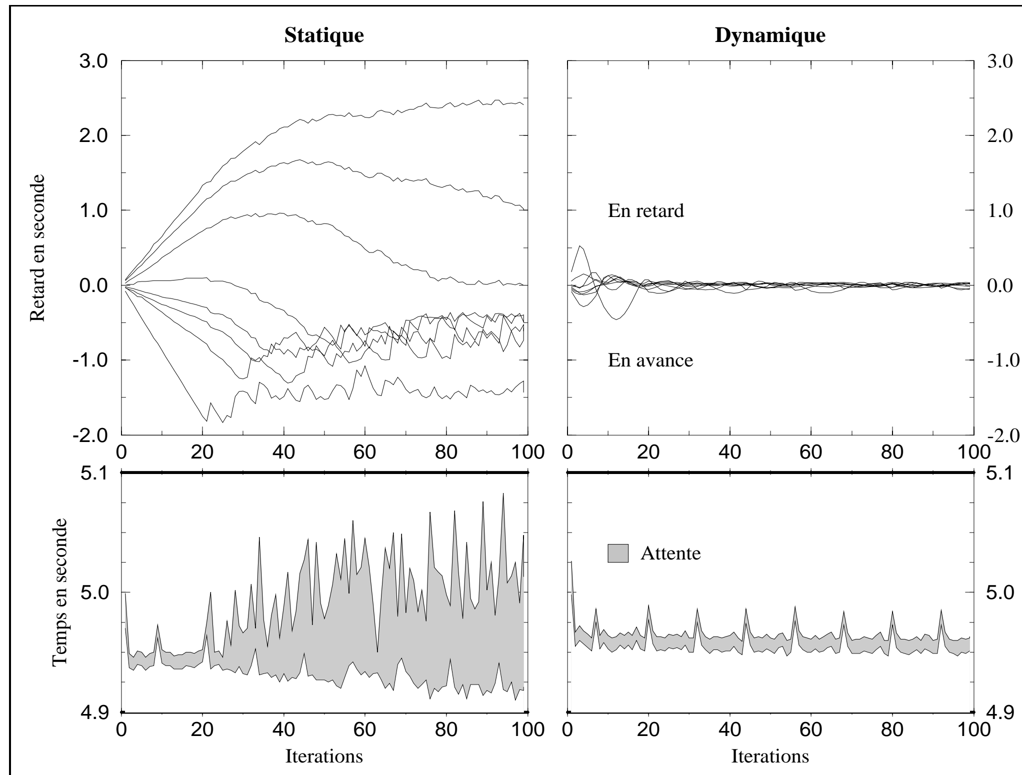


FIG. 6.6: Comparaisons de l'évolution des temps d'exécution en statique et en dynamique, pour la structure de 35 349 atomes sur 8 processeurs en utilisant ATHAPASCAN-0B. Les graphiques du haut indiquent le retard ou l'avance relative de chacun des processeurs. Les graphiques du bas indiquent la moyenne, sur les 8 processeurs, des temps d'exécutions de l'itération courante. La zone en gris indique la part du temps d'attente.

Choix des tâches à déplacer

Déplacer n'importe quelle tâche de calcul des forces non-liées sur n'importe quel processeur est une opération très coûteuse. En effet, la structure de données du système simulé est complètement distribuée sur les processeurs. Des modifications importantes dans cette structure impliquent de nouvelles communications, pour déplacer les données relatives aux atomes, et pour maintenir la cohérence de données distribuées. Nous avons fait beaucoup d'études dans le chapitre précédent pour obtenir une bonne localité et pour minimiser les communications grâce à leurs regroupements. Pour être performant, le mécanisme d'équilibre de charge doit absolument tenir compte de ces propriétés. Sous ces conditions, rendre complètement dynamique le placement des tâches sur le processeur semble peu réaliste.

En fait, nous avons abordé le problème de régulation de charge en cherchant à répondre à la question suivante : quels sont les mouvements de charge de calcul possibles qui permettent de conserver les propriétés de localité du placement initial et qui introduisent un minimum de communications supplémentaires ?

Le déplacement d'une tâche $T_{\text{nv}}(i, j)$, dont les deux boîtes associées sont sur deux pro-

cesseurs différents, sur l'un ou l'autre des processeurs, est le seul mouvement de charge de calcul qui ne nécessite pas de communiquer des coordonnées et des forces supplémentaires. Ou plus précisément, cela signifie que l'on peut déplacer, pour l'itération suivante, le calcul d'une tâche $T_{\text{nv}}(i, j)$ de son groupe $G_{\text{nv}}(p, q)$ (donc une tâche du processeur p) vers le groupe $G_{\text{nv}}(q, p)$ (donc sur le processeur q), sans communications supplémentaires relatives aux atomes. En effet, comme nous l'avons écrit dans le chapitre précédent pour définir les contraintes de placement (cf.5.2.3), les communications nécessaires aux calculs de ces tâches sont déjà nécessaires aux calculs des forces d'interactions géométriques. Concrètement dans le programme, migrer des tâches T_{nv} d'un groupe $G_{\text{nv}}(p, q)$ au groupe $G_{\text{nv}}(q, p)$, consiste simplement à envoyer l'identité des tâches déplacées au moment de la communication des coordonnées (groupe de tâches $EC(p, q)$). Notons que ces mouvements de tâches conservent complètement la distribution initiale des atomes. Ils ne modifient pas l'espace géométrique dont s'occupent chacun des processeurs. Ainsi, la localité des données est aussi conservée.

D'un point de vue plus général, on déplace dynamiquement des calculs relatifs à la frontière entre deux processeurs, sur l'un ou l'autre des deux processeurs. Avec cette stratégie, seule quelques tâches peuvent bouger (et encore, leur mouvement est limité à une migration vers un processeur déterminé). C'est pourquoi nous avons préféré parler **d'ajustement dynamique** de la charge. Il reste à vérifier expérimentalement que ces mouvements sont suffisants pour équilibrer la charge dynamiquement.

Choix du placement initial

Comme toutes les tâches ne peuvent pas être déplacées, le placement initial est important. C'est pourquoi nous partirons des deux placements qui donnent les meilleurs résultats : le placement par *bi-partition récursive simple* (BPR), et le placement par *bi-partition récursive, avec un découpage fin de l'ensemble des boîtes* (BPR-FIN). En effet, au vu des mesures du chapitre précédent (cf. 5.3.6) ces deux placements ont sensiblement les mêmes propriétés. C'est-à-dire qu'ils permettent d'obtenir un bon équilibre de charge tout en conservant une bonne localité des données.

Une partie de la charge de chacun des processeurs est fixée, elle ne peut pas être déplacée. Pour offrir le maximum de souplesse à notre régulateur de charge, nous avons intérêt à ce que cette charge fixe soit la plus petite possible sur chacun des processeurs. Concrètement cela signifie qu'il faut pouvoir déplacer un maximum de tâches de type T_{nv}. Mais pour offrir une alternative de déplacement à un maximum de tâches de type T_{nv}, il faut qu'un maximum de boîtes voisines, au sens du rayon de coupure, soit sur des processeurs différents. Ceci est contraire à la recherche d'une bonne localité des données. En effet, pour avoir une bonne localité des données et minimiser les communications, il faut chercher à minimiser la taille des frontières des sous-domaines de chacun des processeurs. Plus simplement, le placement initial quantitatif offre la meilleure souplesse pour l'ajustement dynamique (beaucoup de tâches de type T_{nv} peuvent être déplacées). Mais la localité de ce placement est mauvaise, il y a beaucoup de communications.

Nous avons définitivement tranché la question au chapitre précédent, les deux place-

ments par bi-partition récursive (BPR et BPR-FIN) donnent les meilleurs performances. Au vu des critères d'équilibre de charge et de localité du chapitre précédent, ces deux placements ont sensiblement les mêmes caractéristiques. Cependant le placement BPR-FIN est préférable pour notre méthode d'ajustement dynamique de la charge. En effet, il découpe l'espace de simulation plus finement, et il permet ainsi d'obtenir un meilleur équilibre de la charge fixe de chacun des processeurs. Pour montrer cette propriété, nous avons défini $\tilde{L}_p$ la charge fixe du processeur p comme la somme des coûts des tâches de p qui ne peuvent pas être déplacées. C'est-à-dire les tâches $T_{\text{ni}}(i)$ de p et les tâches $T_{\text{nv}}(i,j)$ dont les deux boîtes associées sont sur un même processeur p.

$$\tilde{L}_p = \sum_{T \in \tilde{E}_p} C(T) \quad \text{avec}$$

$$\tilde{E}_p = \{T_{\text{ni}}(i) \mid M(T_{\text{ni}}(i)) = p\} \cup \{T_{\text{nv}}(i,j) \mid (M(T_{\text{ni}}(i)) = p) \wedge (M(T_{\text{ni}}(j)) = p)\}$$

Puis, comme au chapitre précédent pour le calcul du déséquilibre de charge, on calcule le déséquilibre de la charge de calcul fixé $\Delta\tilde{L}$ sur les $\mathtt{P}$ processeurs par :

$$\Delta\tilde{L} = \sqrt{\frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P}-1} (\tilde{L}_p - \overline{\tilde{L}})^2} \quad \text{avec} \quad \overline{\tilde{L}} = \frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P}-1} \tilde{L}_p$$

Le tableau 6.2 donne une évaluation du critère d'équilibre de la charge fixe pour chacun des deux placements par bi-partition récursive.

TAB. 6.2: Critère de comparaison de la qualité des placements pour la structure de 11 615 atomes, en fonctions du nombre de processeurs.

Équilibre de charge des tâches fixées $\Delta\tilde{L}(\mathtt{P})$ $(\times 10^3)$						
Nb. Proc. (P)	1	2	4	8	16	32
Placement BPR	0,0	22,1	257,2	38,2	42,0	25,6
Placement BPR-FIN	0,0	82,5	43,3	20,8	27,2	9,7

Le placement BPR-FIN offre généralement un meilleur équilibre de la charge fixe sur les processeurs. Ce qui offre plus de souplesse à notre méthode d'ajustement dynamique.

Évaluation de la charge

Nous avons abordé la régulation de charge un peu comme le contrôle d'un automate mécanique. Rappelons que pour assurer un travail permanent aux processeurs, nous cherchons à minimiser le décalage entre eux. Plus simplement, nous allons essayer de déplacer la charge pour que chacune des itérations se termine en même temps sur tous les processeurs. Pour cela, on modélise le comportement de chacun des processeurs pour estimer la date de fin des itérations futures. Puis à partir de la charge moyenne des processeurs,

on modélise le comportement global du programme pour prévoir la date moyenne de la fin des itérations futures. Ces dates moyennes de fin d'itération servent d'objectif au régulateur. En effet, en fonction de l'estimation locale de leur charge, les processeurs vont modifier leur charge de calcul pour que leurs itérations futures terminent effectivement à la date prévue.

Pour simplifier, nous supposons que les processeurs sont homogènes et que nous disposons d'une horloge commune à chacun de ces processeurs.

Ainsi, plus précisément, à la fin d'une itération k, on évalue la charge d'un processeur p à l'aide de deux valeurs :

- $d_p(k)$ la date de fin de l'itération k ;
- $c_p(k)$ une évaluation du temps de calcul de l'itération suivante si on ne fait pas de modification de charge.

On évalue le temps de calcul de l'itération suivante $c_p(k)$ grâce au temps de calcul des itérations précédentes.

$$c_p(k) = \alpha c_p(k-1) + (1-\alpha)\left(d_p(k) - d_p(k-1)\right)$$

Ici $\alpha \in [0,1]$ est un paramètre empirique qui permet de choisir l'importance relative des itérations passées par rapport à la dernière itération. En effet, si on évalue la charge seulement à partir de la dernière itération ($\alpha = 0$), les valeurs obtenues sont trop sensibles aux erreurs de mesures. Une valeur de α non nulle permet d'obtenir une moyenne pondérée sur plusieurs itérations du temps de calcul d'une itération. Inversement, une valeur de α proche de 1 signifie que l'on tient peu compte de la modification de charge du processeur dans l'évaluation du temps de calcul d'une itération. Nous avons utilisé une valeur moyenne ($\alpha = 1/2$).

Ensuite, à partir de la date de fin de l'itération et de l'estimation du temps d'une itération, on peut évaluer $\hat{d}_p(k+i)$ la date de fin des itérations futures du processeur p. Bien sûr, on suppose pour cela que le processeur p n'attend pas. Ainsi, on prévoit $\hat{d}_p(k+i)$ la date de fin de l'itération i itérations à l'avance par :

$$\hat{d}_p(k+i) = d_p(k) + i \cdot c_p(k)$$

De la même manière, on cherche ensuite à prévoir la date de fin de l'itération idéale pour tout les processeurs. On évalue ainsi la date moyenne de fin de l'itération $\bar{d}(k)$ et le temps moyen de calcul d'une itération $\bar{c}(k)$.

$$\bar{d}(k) = \frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P-1}} d_p(k) \quad \text{et}$$

$$\bar{c}(k) = \frac{1}{\mathtt{P}} \sum_{p=0}^{\mathtt{P-1}} c_p(k) = \alpha \bar{c}_p(k-1) + (1-\alpha)\left(\bar{d}_p(k) - \bar{d}_p(k-1)\right)$$

On peut ainsi estimer la date moyenne idéale $\hat{d}(k+i)$ de fin des itérations futures.

$$\hat{d}(k+i) = \bar{d}(k) + i \cdot \bar{c}_p(k)$$

Cette date, prévue i itérations à l'avance sert d'objectif à la régulation locale de la charge sur chacun des processeurs esclaves.

Évaluation du déséquilibre de charge

Pour réguler la charge, nous avons besoin de calculer la charge globale du système ($\bar{d}(k)$ et $\bar{c}(k)$). Pour éviter d'ajouter des nouveaux liens de communications entre les processeurs, nous faisons circuler les informations de charge avec les informations de contrôle de la simulation. La figure 6.7 montre comment circulent ces informations de charge entre le processeur maître et les processeurs esclaves au cours des itérations. Sur le processeur maître, à partir des informations de charge de l'itération k, en provenance de tous les processeurs esclaves, on estime la date moyenne de fin de l'itération $k + 2$. Puis on diffuse cette date à tout les processeurs esclaves. Elle sert d'objectif pour la régulation de la charge au début de l'itération $k + 2$. Nous sommes obligés de prévoir la date de fin de l'itération moyenne deux itérations à l'avance, car les informations de charge circulent, entre le processeur maître et les processeurs esclaves, pendant l'itération $k + 1$, en même temps que les autres informations de contrôle de la simulation.

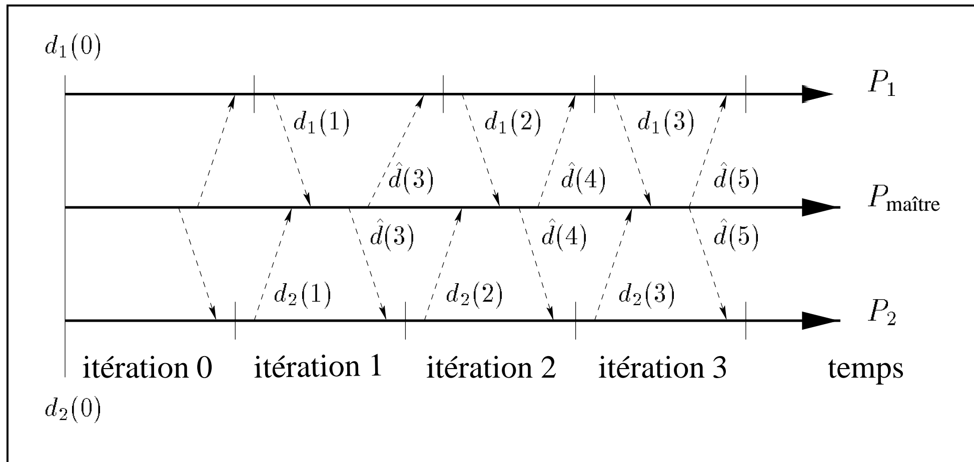


FIG. 6.7: Circulation des informations de charge entre le processeur maître et deux processeurs esclaves.

Le déséquilibre de charge $\delta c_p(k)$, du processeur p, est évalué au début de l'itération courante k. On utilise la prévision, de la charge locale, une itération à l'avance, et à partir de l'itération précédente ($\hat{d}_p((k-1)+1)$). Comme objectif, on utilise la prévision, sur le processeur maître, et deux itérations à l'avance (avec les informations d'il y a deux itérations) de la date moyenne de la fin de l'itération courante ($\hat{d}((k-2)+2)$).

$$\delta c_p(k) = \hat{d}_p((k-1)+1) - \hat{d}((k-2)+2)$$

Pour simplifier la migration de charge de calcul, nous avons utilisé uniquement un mécanisme d'exportation de la charge. C'est-à-dire, qu'un processeur surchargé ($\delta c_p(k) > 0$) peut envoyer des tâches à ses processeurs voisins. Mais un processeur sous-chargé ($\delta c_p(k) < 0$) ne peut pas importer des tâches. Ceci n'est pas trop gênant si on estime correctement la date moyenne de fin de l'itération. En effet, cela signifie que globalement,

sur l'ensemble des processeurs, il y a autant de surcharge que de sous-charge.

$$\sum_{p=0}^{P-1} \delta c_p(k) = 0$$

En fait, les processeurs sous-chargés ont juste à recevoir les tâches exportées par les processeurs surchargés.

Sélection des tâches à déplacer

Les processeurs surchargés diffusent leur charge supplémentaire aux processeurs voisins. Ainsi, pour un processeur p surchargé de $\delta c_p(k)$ et possédant v_p voisins, il exporte une charge de $\delta c_p(k)/v_p$ à chacun de ses voisins. Par exemple, si le processeur q est un voisin de p au sens du rayon de coupure, le mécanisme de sélection des tâches choisit des tâches $T_{\text{nv}}(i, j)$ dans le groupe $G_{\text{nv}}(p, q)$. Ensuite, il les exporte vers le processeur q pour qu'elles soient calculées dans le groupe $G_{\text{nv}}(q, p)$.

Comment sont choisies les tâches $T_{\text{nv}}(i, j)$ dans le groupe $G_{\text{nv}}(p, q)$? Au cours des itérations précédentes, on a mesuré le temps de calcul des tâches $T_{\text{nv}}(i, j)$ du groupe $G_{\text{nv}}(p, q)$. On se sert simplement de ces mesures pour choisir les tâches $T_{\text{nv}}(i, j)$ et exporter une charge correspondant à $\delta c_p(k)/v_p$.

La partie droite de la figure 6.6 illustre comment la charge des processeurs est régulée. Comme le montre le graphique en haut à droite, le mécanisme de régulation force les processeurs à terminer les itérations en même temps. Ainsi, les processeurs restent toujours actifs. Le graphique en bas à droite montre que le temps total moyen (t_{tot}) de calcul d'une itération n'augmente pas au cours de la simulation. De même, le temps d'attente moyen (t_{att}) reste faible.

6.3.2 Résultats

Dans le tableau 6.3, nous avons mesuré la moyenne, sur les 100 premières itérations, des temps de calcul d'une itération. Le tableau du haut donne les mesures sans utiliser l'ajustement dynamique, seulement avec un placement statique initial. Et le tableau du bas donne les mesures avec l'utilisation du mécanisme de régulation de charge.

À première vue, les améliorations sur le temps total moyen ($t_{\text{tot}}(p)$) de calcul des itérations sont très faibles. Les mesures sur 4 processeurs sont mêmes légèrement meilleures sans ajustement dynamique de la charge. En fait, il faut regarder le détail des temps d'exécution. L'ajustement dynamique de la charge essaye de garder les processeurs actifs, il cherche seulement à diminuer le temps d'attente moyen ($t_{\text{att}}(p)$) des processeurs. On observe que pour 8, 12 et 16 processeurs l'ajustement dynamique permet effectivement de faire diminuer ces temps d'attente, et les performances sont effectivement meilleures. On remarque également que les temps de communications et les temps de calcul ne changent pas. Ainsi notre méthode d'ajustement dynamique n'introduit pas de sur-coûts de calcul ou de sur-coûts de communications significatifs.

TAB. 6.3: Détail des temps d'exécution d'une itération de dynamique moléculaire, pour un système de 35 349 atomes en statique et en dynamique, et en utilisant ATHAPASCAN-0B.

Statique (temps en seconde)							
Nb. Proc. (p)	1	4	8	12	16	20	24
$t_{\text{calc}}(p)$	38,21	9,47	4,78	3,20	2,40	1,92	1,61
$t_{\text{comm}}(p)$	0,00	0,20	0,15	0,14	0,11	0,09	0,08
$t_{\text{att}}(p)$	0,00	0,01	0,06	0,19	0,14	0,23	0,27
$t_{\text{tot}}(p)$	38,21	9,68	4,99	3,53	2,65	2,24	1,96

Dynamique (temps en seconde)							
Nb. Proc. (p)	1	4	8	12	16	20	24
$t_{\text{calc}}(p)$	38,21	9,48	4,79	3,20	2,39	1,92	1,60
$t_{\text{comm}}(p)$	0,00	0,20	0,16	0,15	0,11	0,09	0,08
$t_{\text{att}}(p)$	0,00	0,01	0,01	0,06	0,08	0,24	0,25
$t_{\text{tot}}(p)$	38,21	9,69	4,96	3,41	2,58	2,25	1,93

Sur 4 processeurs, le temps d'attente moyen est très faible (seulement 0,01s). Nous avons utilisé un placement BPR-FIN qui équilibre bien la charge de calcul. Mais en plus, en statique, nous avons mis en évidence l'existence d'une phase transitoire pendant laquelle les processeurs se décalent tout en restant actifs. Comme nous effectuons nos mesures sur les 100 premières itérations, on sous-évalue légèrement le temps d'attente moyen. En effet, on n'élimine pas cette phase transitoire dans les mesures. Plus précisément, et toujours sur 4 processeurs, le temps total pour faire les 100 itérations (maximum du temps total de chacun des 4 processeurs) sans ajustement dynamique est de 971,33 secondes. Alors que la même mesure, en utilisant l'ajustement dynamique, est de 969,61 secondes, donc légèrement meilleure. Contrairement à un placement statique, avec le mécanisme d'ajustement dynamique, les processeurs terminent tout au même moment. À long terme (beaucoup d'itérations) les performances sont légèrement meilleures avec l'ajustement dynamique.

En revanche, au-delà de 16 processeurs, l'ajustement dynamique est incapable de diminuer le temps d'attente moyen. Nous avons naturellement commencé par croire à un problème de granularité. Nous avons pensé que, au-delà de 20 processeurs, le temps de calcul des tâches $T_{\text{nv}}(i,j)$ que l'on peut déplacer est supérieur au déséquilibre de charge possible. En effet, pour qu'une tâche puisse migrer d'un processeur à l'autre il faut que son temps de calcul $t(T_{\text{nv}}(i,j))$ soit inférieur à la charge à exporter. Ainsi, pour un processeur p possédant v_p voisins et ayant une surcharge de $\delta c_p(k)$, pour exporter $T_{\text{nv}}(i,j))$ il faut que :

$$t(T_{\text{nv}}(i,j)) < \delta c_p(k)/v_p$$

Or avec notre système de mesure, le déséquilibre de charge maximum d'un processeur est majoré par le décalage maximum possible entre les processeurs. Si le décalage maximum possible est trop faible, il n'est pas possible d'exporter des tâches. Mais ici, diminuer la granularité ne change rien. En effet, nous avons observé au chapitre précédent (cf. 5.5), que pour un rayon de coupure trop faible (ici 10 Å), diminuer la granularité dégrade les performances. Mais une granularité trop grosse n'est pas la seule explication.

Problèmes de réactivité

Nous avons effectué les mesures précédentes avec un système comportant un nombre d'atomes constant, indépendamment du nombre de processeurs. Et en fait plusieurs facteurs se combinent pour empêcher le régulateur de charge de fonctionner correctement pour ce système, au-delà de 16 processeurs.

La figure 6.8 compare avec et sans l'ajustement dynamique de la charge, le décalage des processeurs et le temps d'exécution de chacune des itérations.

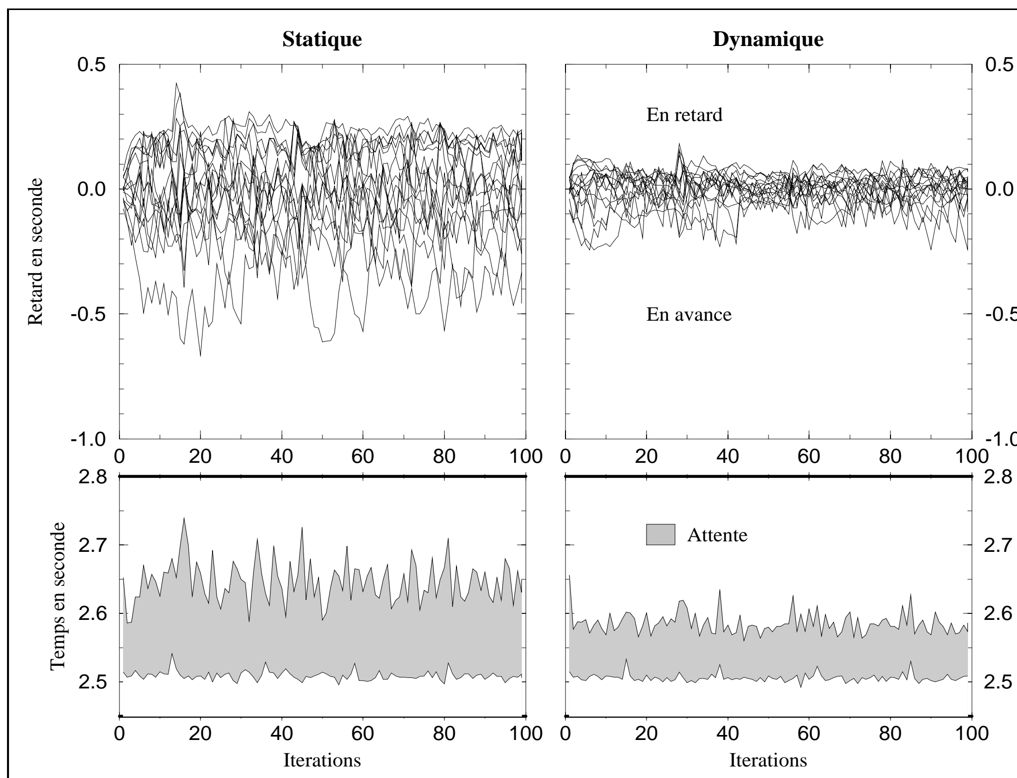


FIG. 6.8: Comparaisons de l'évolution des temps d'exécution en statique et en dynamique, pour la structure de 35 349 atomes sur 16 processeurs en utilisant ATHAPASCAN-0B. Les graphiques du haut indiquent le retard ou l'avance relative de chacun des processeurs. Les graphiques du bas indiquent la moyenne, sur les 16 processeurs, des temps d'exécutions de l'itération courante, La zone en gris indique la part du temps d'attente.

Sur les deux graphiques de gauche, on observe que la phase de transition est beaucoup plus courte que sur 8 processeurs (cf. figure 6.6), elle dure seulement 4 ou 5 itérations,

contre plus de 20 itérations sur 8 processeurs. Nous pouvons avancer essentiellement deux explications à cela :

- Le décalage maximum possible entre les processeurs est plus faible sur 16 processeurs (seulement 0,8 secondes) que sur 8 processeurs où il est d'environ 3 secondes. Il est normal que ce décalage soit au moins deux fois plus faible puisque l'on a doublé le nombre de processeurs.

  Mais en fait il ne représente plus que 31% du temps total de l'itération contre 60% sur 8 processeurs. En augmentant le nombre de processeurs, on diminue également la localité. Ainsi, la part des calculs locaux aux processeurs, qui permet le décalage, diminue quand on augmente le nombre des processeurs.
- Toujours pour une taille de problème constante, le déséquilibre de charge entre les processeurs augmente avec le nombre de processeurs. Ainsi le décalage entre les processeurs d'une itération à l'autre est plus rapide.

En résumé, quand on augmente le nombre de processeurs, pour une taille de problème constante, les processus légers offrent moins de souplesse à l'algorithme. Le déséquilibre de charge entre les processeurs est également proportionnellement plus important. Donc, le décalage entre les processeurs est plus rapide. C'est-à-dire que le nombre d'itérations, avant que le décalage entre les processeurs soit trop important pour les garder actifs, diminue. Si ce nombre d'itérations est trop petit, le régulateur, qui a besoin d'au moins 3 itérations pour réagir, est incapable de compenser le déséquilibre de charge à temps. Nous dirons que le régulateur n'est pas assez réactif.

La partie droite de la figure 6.8 montre comment le mécanisme de régulation de charge arrive à réduire le décalage entre les processeurs. Ainsi, il permet d'améliorer les performances en réduisant le temps d'attente moyen des processeurs. La régulation de la charge des processeurs n'est cependant pas parfaite, le régulateur ne réagit pas tout à fait assez rapidement.

6.4 Déplacement des atomes et ajustement dynamique

Nous en avons très peu discuté jusqu'à présent, mais pour réaliser des simulations longues, il faut déplacer les atomes sur les processeurs. En effet, les atomes sont placés dans des boîtes en fonction de leur position. Or les atomes bougent au cours de la simulation et peuvent changer de boîte. Comme les boîtes sont distribuées sur les processeurs, certains atomes qui changent de boîte peuvent également avoir à changer de processeur. Attention, le placement des boîtes sur les processeurs ne change pas, les processeurs s'occupent toujours de la même région de l'espace de simulation. Seul les atomes qui bougent dans l'espace de simulation sont amenés à changer de boîte, et donc à changer de processeur.

Nous avons vu au chapitre 4 (cf. 4.2.2) que grâce à la distance de tolérance, un mouvement limité des atomes est possible sans avoir à les replacer dans les boîtes. Ceci permet de mettre à jour le placement des atomes dans les boîtes seulement toutes les 50 itérations environ. Remarquons que les atomes se déplacent uniquement vers une boîte adjacente à

leur boîte courante. Les interactions géométriques utilisent des pointeurs sur les atomes, pour pouvoir connaître leur position et mettre à jour leur vecteur de force. Déplacer les atomes implique qu'il faut également mettre à jour les pointeurs des interactions géométriques. De plus, pour pouvoir calculer une interaction géométrique sur un processeur, il faut qu'au moins un de ses atomes soit sur le processeur en question (cf. 5.3.4). Donc le déplacement des atomes implique également le déplacement des interactions géométriques.

6.4.1 Principe

Mettre à jour les pointeurs des interactions géométriques après le déplacement des atomes n'est pas une opération triviale. Pour résoudre ce problème, nous avons utilisé un mécanisme de suivi d'adresses. Quand un atome change de boîte, il indique sa nouvelle adresse à son ancienne adresse. Ensuite les interactions géométriques modifient leurs pointeurs en cherchant la nouvelle adresse des atomes à leur ancienne adresse. L'algorithme complet comporte les étapes suivantes :

1. on calcule la nouvelle boîte de chacun des atomes. Si cette nouvelle boîte change mais qu'elle est sur le même processeur, on place l'atome dans une file locale. Si cette nouvelle boîte est sur un processeur différent, on envoie l'atome au processeur concerné ;
2. on place les atomes de la file locale dans leurs nouvelles boîtes. Conjointement, on construit une file contenant les nouvelles adresses des atomes déplacés ;
3. de même avec les atomes que l'on reçoit des autres processeurs, on les place dans leurs nouvelles boîtes. Et conjointement, on renvoie la nouvelle adresse de l'atome au processeur émetteur ;
4. on met à jour les nouvelles adresses des atomes à leur ancienne adresse, en utilisant la file locale des nouvelles adresses ;
5. de même avec les adresses reçues des processeurs voisins, on les utilise pour mettre à jour les nouvelles adresses des atomes, à leur ancienne adresse. Après cette étape, les atomes ont changé de boîte et les nouvelles adresses des atomes sont à la position de leurs anciennes adresses ;
6. on échange les nouvelles adresses des atomes entre les processeurs voisins (comme on échange les coordonnées des atomes pour le calcul des forces). En effet, les interactions géométriques locales à un processeur n'utilisent pas uniquement les atomes des boîtes du processeur. Elles utilisent également les atomes des boîtes voisines de celles du processeur. Donc pour changer localement les pointeurs des interactions géométriques, on a aussi besoin des nouvelles adresses des atomes des boîtes voisines, d'où cette communication ;
7. ensuite, on parcourt les interactions géométriques pour mettre à jour les nouvelles adresses des atomes ;
8. la dernière étape consiste à échanger entre les processeurs voisins, les interactions géométriques dont tous les atomes ont changé de processeur.

Cet algorithme distribué, pour le déplacement des atomes, est extensible. Son temps de calcul est négligeable par rapport au temps de calcul d'une itération. Mais il a tout de même un inconvénient important, il synchronise les processeurs entre eux. En effet, les nombreuses communications ne sont pas ou très peu recouvertes par des calculs.

6.4.2 Résultats

La partie en haut à gauche de la figure 6.9 montre le décalage relatif entre les processeurs pour un placement statique seul. Le graphique en bas à gauche indique le temps d'exécution de l'itération courante avec la part du temps d'attente en gris, et toujours pour un placement statique. La figure de droite montre ce qui se passe avec un ajustement dynamique de la charge. Sur la figure en haut à gauche, on observe une synchronisation des

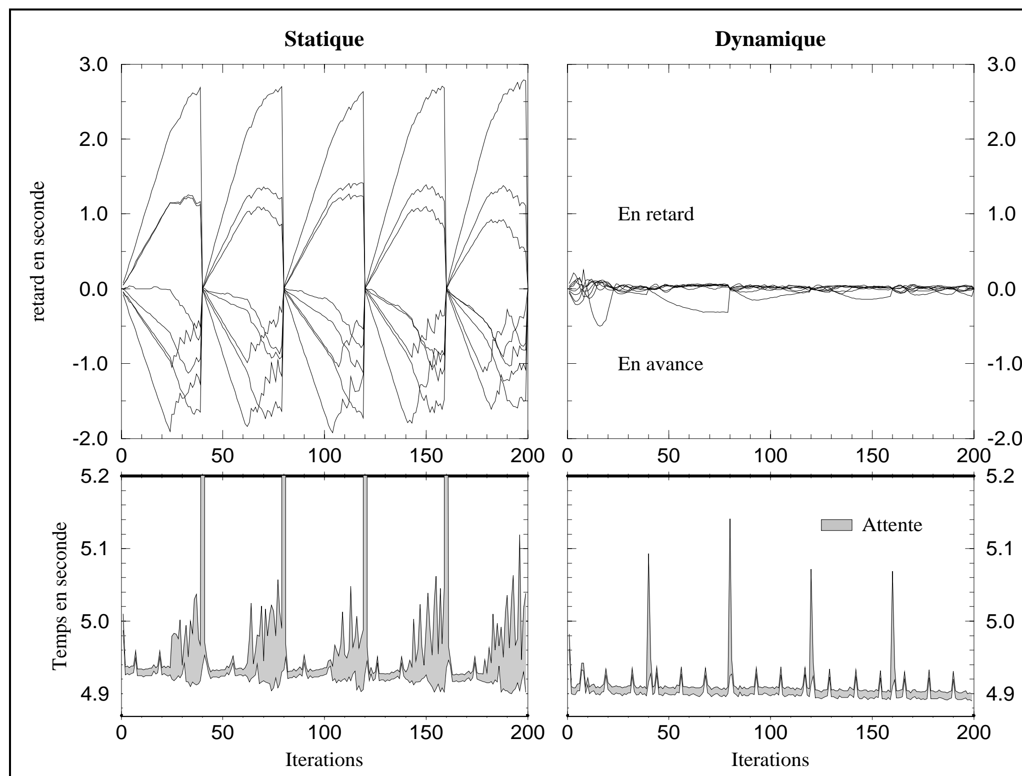


FIG. 6.9: Comparaisons de l'évolution des temps d'exécution en statique et en dynamique avec une redistribution des atomes tout les 40 pas, pour la structure de 35 349 atomes sur 8 processeurs, et en utilisant ATHAPASCAN-0B. Les graphiques du haut indiquent le retard ou l'avance relative de chacun des processeurs. Les graphiques du bas indiquent la moyenne, sur les 8 processeurs, des temps d'exécution de l'itération courante. La zone en gris indique la part du temps d'attente.

processeurs, régulièrement toutes les 40 itérations, quand on met à jour les atomes dans les boîtes. Conjointement sur la figure du bas, ces itérations se traduisent par un pic dans le temps total de l'itération qui dépasse l'échelle de la figure (ces itérations durent environ

8 secondes en réalité). Ceci est également dû à la synchronisation, les processeurs les plus en avance attendent le processeur le plus chargé. Entre ces synchronisations, on observe le même phénomène transitoire que précédemment.

Les graphiques de droite montrent tout l'intérêt de réguler la charge en obligeant les processeurs à terminer en même temps. En effet, lors de la redistribution des atomes sur les processeurs, les pics sont nettement plus faibles. Les processeurs, qui terminent les itérations en même temps du fait de la régulation de charge, sont déjà synchronisés. On peut également remarquer que le déplacement des atomes n'implique pas de modifications importantes de la charge de calcul. Le déplacement des atomes d'une boîte I vers une boîte J implique également des mouvements d'atomes de la boîte J vers la boîte I. Globalement, le déplacement des atomes modifie très peu le nombre d'atomes par boîte, il modifie donc aussi très peu l'équilibre de charge.

Globalement, en tenant compte du déplacement des atomes, sans ajustement dynamique, le temps moyen d'une itération sur 8 processeurs est de 5,03 secondes (dont 0,11 secondes de temps d'attente). Grâce à l'ajustement dynamique, le temps moyen d'une itération est de 4,91 secondes (dont 0,01 secondes de temps d'attente). Ces deux dernières mesures sont des moyennes sur les 300 premières itérations.

Quand on fait des mesures sur des structures plus importantes, l'équilibre dynamique permet également d'améliorer les performances. Ainsi, sur l'IBM-SP1, nous avons calculé la dynamique du système de 413 039 atomes sur 200 itérations et sur 16 processeurs, et toujours en mettant à jour les atomes dans les boîtes toutes les 40 itérations. Sans ajustement dynamique, le temps moyen de calcul d'une itération est de 47,2 secondes (dont 0,65 secondes de temps d'attente). Avec l'ajustement dynamique, le temps moyen de calcul d'une itération est de 46,63 secondes (dont 0,05 secondes de temps d'attente).

6.5 Extensibilité, mesures sur le CRAY-T3E

L'objectif de ce paragraphe est de vérifier l'extensibilité de notre algorithme. Le CRAY-T3E du CEA possède 256 processeurs et permet ainsi de faire ce type d'étude. Sur cette machine, nous avons utilisé un portage de ATHAPASCAN-0B. Dans cet environnement, la plate-forme d'exécution est construite à l'aide de la bibliothèque de communication MPI disponible sur cette machine et du noyau de processus légers MARCEL [100] [101]. Le tableau 6.4 donne les moyennes des temps de calcul, l'accélération et l'efficacité d'une itération de dynamique moléculaire sur les 50 premières itérations, pour trois structures de tailles différentes. Nous avons utilisé l'équilibrage dynamique de la charge de calcul. Nous avons également indiqué les temps de calcul sans ajustement dynamique entre parenthèses. L'accélération (Acc) et l'efficacité (Eff) sont calculées à partir de l'exécution du programme sur un processeur, sauf pour la structure de 413 039 atomes où elles sont calculées à partir de l'exécution sur 8 processeurs. En effet, cette structure est trop grosse pour être exécutée sur un processeur.

TAB. 6.4: Temps d'exécution (t_{tot}), accélération (Acc) et efficacité (Eff) du calcul d'une itération de dynamique moléculaire, pour un système de 11 615 atomes, un système de 35 349 atomes et un système de 413 039 atomes. Les chiffres entre parenthèses indiquent les temps d'exécution sans ajustement dynamique de la charge de calcul.

Structure de 11 615 atomes (temps en seconde)							
Nb. Proc. (p)	1	2	4	8	16	32	64
$t_{\text{tot}}(p)$	6,87	3,51	1,79	0,94	0,52	0,34	0,23
Acc(p)	1,00	1,96	3,84	7,31	13,21	20,21	29,87
Eff(p) (%)	100	97,8	95,9	91,3	82,6	63,1	46,7

Structure de 35 349 atomes (temps en seconde)								
Nb. Proc. (p)	1	2	4	8	16	32	64	128
$t_{\text{tot}}(p)$	21,80	10,77	5,39	2,76	1,43	0,79	0,56	0,30
Acc(p)	1,00	2,02	4,04	7,89	15,24	27,59	38,93	72,67
Eff(p) (%)	100	101,2	101,1	98,7	95,2	86,2	60,8	56,8

Structure de 413 039 atomes (temps en seconde)						
Nb. Proc. (p)	8	16	32	64	128	256
$t_{\text{tot}}(p)$	65,21	27,70	11,69 (12,11)	5,88 (6,14)	3,07 (3,22)	1,69 (1,83)
Acc(p)	8,00	18.83	44,63	88,72	169,93	308,69
Eff(p) (%)	100	117,7	139,5	138,6	132,7	120,6

Pour la petite structure de 11 615 atomes, la granularité est trop grosse pour l'exécuter

sur 128 processeurs. Nous avons diminué la granularité (cf. 5.5) mais le coût de gestion des tâches est trop important pour obtenir de bonnes performances (0,228 secondes sur 128 processeurs). Cela n'est pas trop étonnant, sur 128 processeurs il y a en moyenne moins de 100 atomes par processeur. En réalité, au-delà de 16 processeurs, l'efficacité chute rapidement. En effet, à partir de 32 processeurs, l'algorithme de régulation dynamique n'arrive plus à équilibrer la charge de calcul et à diminuer le temps d'attente. Le phénomène est identique pour la structure de 35 349 atomes à partir de 64 processeurs.

Pour simuler le mouvement de la structure de 413 039 atomes, il faut au moins huit processeurs du CRAY-T3E. En dessous, la mémoire disponible sur chacun des noeuds est insuffisante. Nous avons donc calculé l'accélération et l'efficacité de l'algorithme à partir des résultats sur huit processeurs. Cette structure est très intéressante pour montrer l'extensibilité de notre algorithme. En effet, il est, vu sa taille, impossible actuellement de simuler une telle structure sur une machine séquentielle. Nous avons également essayé de simuler cette structure sur deux autres programmes parallèles de dynamique moléculaire, mais sans y parvenir.

Pour accélérer les calculs, nous avons construit les listes d'interactions (cf. 4.2.2) localement sur chacun des processeurs, et à l'aide du découpage en boîtes de l'espace de simulation. Mais pour la grosse structure, il n'est pas possible, sur 8 processeurs, de construire toute la liste des interactions (il n'y a pas assez de mémoire). On construit donc la liste partiellement en fonction de la place mémoire disponible. Pour les tâches pour lesquelles on ne peut pas construire cette liste, on utilise l'algorithme traditionnel plus lent. Quand on augmente le nombre de processeurs, on augmente la mémoire globale disponible pour le programme. On peut donc construire une plus grande partie de la liste des interactions. C'est ce qui explique le comportement super-linéaire de l'accélération jusqu'à 32 processeurs pour cette structure.

6.6 Conclusion

Dans ce chapitre, nous avons montré comment la multiprogrammation permet d'introduire de la souplesse dans la description des algorithmes parallèles irréguliers. En particulier, nous avons vu les avantages d'un ordonnancement dynamique entrelaçant les tâches de calcul et les tâches de communications. Enfin, la souplesse offerte par la multiprogrammation s'est avérée être très utile pour notre mécanisme d'ajustement dynamique de la charge.

Bien que le déséquilibre de charge soit généralement très faible après un placement BPR-FIN, notre mécanisme d'ajustement dynamique a réussi à améliorer les performances globales de notre programme. Cependant pour une taille de problème donnée, si on augmente trop le nombre de processeurs, le régulateur de charge devient inefficace. En effet, en augmentant le nombre de processeurs, on diminue la souplesse offerte par la multiprogrammation. Le régulateur réagit trop lentement pour éviter que les processeurs les moins chargés attendent. Nous pensons qu'il est possible d'améliorer légèrement la réactivité de notre régulateur, notamment en modifiant la façon de diffuser la surcharge des processeurs, mais le problème de réactivité subsistera quand on augmentera le nombre

des processeurs.

Pour notre problème, un placement initial au début est suffisant pour obtenir des bonnes performances. Même si notre algorithme d'ajustement dynamique permet d'approcher une utilisation optimale des processeurs, cela reste un mécanisme optionnel pour l'utilisateur final.

Enfin, les mesures sur le CRAY-T3E montrent que notre algorithme de découpage géométrique de l'espace de simulation est assez extensible.

Quatrième partie

Conclusion

Chapitre 7

Conclusions et perspectives

Dans cette thèse, nous nous sommes intéressés à la parallélisation d'une application de dynamique moléculaire pour la simulation du mouvement des atomes des protéines. Nous nous sommes également attachés à montrer l'intérêt de la multiprogrammation pour la parallélisation de cette application.

Dans la première partie, nous avons résumé les principes de base du parallélisme. Nous avons ensuite présenté notre méthodologie de parallélisation. Dans le deuxième chapitre, nous avons défini les principes de la multiprogrammation, puis nous avons décrit l'environnement de programmation ATHAPASCAN-0.

Dans la deuxième partie, nous avons commencé par isoler les éléments du modèle indispensables à la simulation du mouvement des protéines. Puis dans le chapitre 4, après une description des méthodes d'approximation des forces non-liées, nous avons détaillé la méthode du rayon de coupure. En particulier, nous avons vu que la méthode du rayon de coupure était nécessaire aux autres méthodes d'approximation.

Dans la troisième partie, consacrée à la parallélisation de la méthode du rayon de coupure, nous avons commencé par chercher un algorithme extensible. Nous avons ensuite réalisé une première maquette de programme de dynamique moléculaire afin de préciser les hypothèses de travail. Nous avons pu observer ainsi l'importance de l'équilibre de charge et la faible évolutivité de cette charge.

Fort des connaissances acquises, nous avons développé un programme de dynamique moléculaire parallèle plus complet dans l'environnement de programmation parallèle ATHAPASCAN-0. Sur ce programme, nous avons testé et évalué différentes stratégies de découpage en tâches et de placements statiques. Nous avons montré que l'algorithme de placement par bi-partition récursive donne le meilleur compromis entre l'équilibre de charge et la minimisation des communications. Nous avons également montré l'intérêt de pouvoir adapter la granularité en fonction des paramètres du programme et du nombre de processeurs disponibles.

Dans le chapitre 6, nous nous sommes attachés à montrer l'utilité de la multiprogrammation pour entrelacer les calculs et les communications et pour introduire de la souplesse dans l'ordonnancement local des tâches. Enfin nous avons étudié un mécanisme d'ajustement de la charge pour compenser les légères imperfections du placement initial. Nous

avons mis en évidence les limites de notre mécanisme d'ajustement dynamique quand on augmente le nombre de processeurs. Nous avons terminé ce chapitre en montrant l'extensibilité de notre algorithme.

En résumé, nous avons grâce au parallélisme réussi à ramener à quelques heures le calcul d'une structure de 30 000 atomes. Nous sommes également capables de simuler des systèmes plus gros (413 000 atomes) en quelques heures, en augmentant le nombre de processeurs. Nous avons également montré l'utilité de la multiprogrammation légère pour entrelacer les calculs et les communications, et pour la gestion des communications non bloquantes. La programmation de cette application a également contribué à l'amélioration de ATHAPASCAN-0, notamment dans la version ATHAPASCAN-0B par l'introduction des primitives de communication entre les processus légers, et par l'implémentation des tubes de communications.

Perspectives

Si notre programme permet actuellement de traiter des systèmes de grande taille, l'utilisation du rayon de coupure ne permet pas de prendre en compte les forces électrostatiques à longue distance. L'implémentation de la méthode des moments multipolaires pour prendre en compte ces forces, constitue la suite logique de ce travail.

Notre algorithme de placement des calculs suppose que les processeurs soient homogènes, mais notre algorithme de calcul des itérations est extensible, il est possible de simuler des problèmes encore plus importants. Pour cela il peut être nécessaire d'utiliser plusieurs machines parallèles différentes en même temps pour résoudre un même problème. L'environnement de programmation ATHAPASCAN-0 est conçu pour fonctionner dans un environnement hétérogène, mais notre algorithme de placement doit être adapté pour tenir compte de la vitesse relative de calcul entre les processeurs. Actuellement une étude de faisabilité est menée pour faire fonctionner le IBM-SP1 de l'IMAG et le CRAY-T3E du CEA en même temps.

Nous avons essentiellement utilisé la multiprogrammation pour partager les ressources du processeur entre les tâches, mais c'est aussi un bon outil pour décrire le parallélisme dans les ordinateurs à mémoire physiquement partagée. Ici aussi notre programme nécessite des améliorations pour supporter ce type d'architecture.

Nous avons montré l'intérêt de la multiprogrammation pour la parallélisation de notre problème irrégulier. Mais cette approche nécessite une analyse très fine du problème. En offrant une meilleure abstraction de la machine-cible ATHAPASCAN-1 [49] [48] doit permettre de simplifier cette phase.

Enfin notre technique d'ajustement de la charge, par déplacement du calcul des frontières, mérite également d'être développée. Des travaux sont actuellement menés pour la généraliser à la décomposition de maillages non structurés. Nous avons restreint notre étude aux ordinateurs dédiés, mais la régulation peut également être utilisée sur une machine partagée. Mais cela nécessite encore beaucoup de travail théorique pour définir l'efficacité dans ce type d'environnement.

Bibliographie

[1] M.-P. ALLEN ET D.-J. TILDESLEY, *Computer Simulation of Liquids.*, Clarendon Press, Oxford, 1987.

[2] E. BAMPIS, J.-C. KÖNIG ET D. TRYSTRAM, *Optimal Parallel Execution of Complete Binary Trees and Grids into most Popular Interconnection Networks.*, Theoretical Computer Science, 147 (1995), pp. 1–18.

[3] B. BANKO ET H. HELLER, *User Manual for* EGO.

[4] J. BARNES ET P. HUT, *A Hierarchical* $O(N \log N)$ *Force-Calculation Algorithm.*, Nature, 324 (1986), pp. 446–449.

[5] D.-M. BEAZLEY ET P.-S. LOMDAHL, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5.*, Parallel Computing, 20 (1993), p. 173.

[6] P.-E. BERNARD, *Dynamique Moléculaire et Calcul Parallèle.* Rapport de D.E.A., juin 1993.

[7] P.-E. BERNARD, B. PLATEAU ET D. TRYSTRAM, *Using Threads for developing Parallel Applications: Molecular Dynamics as a case study.*, dans Parallel Numerics, Trobec, ed., Gozd Martuljek, Slovenia, sept. 1996, pp. 3–16.

[8] P.-E. BERNARD ET D. TRYSTRAM, *Algorithme Parallèle de Dynamique Moléculaire*, Rapport APACHE 20, LMC-IMAG Grenoble France, juin 1996. ftp://ftp.imag.fr/imag/APACHE/RAPPORTS/APACHE_RR20.ps.gz.

[9] P.-E. BERNARD, D. TRYSTRAM ET Y. CHAPRON, *Parallélisation d'un Algorithme de Dynamique Moléculaire.*, dans Actes RenPar8, Bordeaux, juin 1996, pp. 97–100.

[10] J.-A. BOARD, J.-W. CAUSEY ET J.-F. LEATHRUM, *Accelerated Molecular Dynamics Simulation with the Parallel Fast Multipole Algorithm.*, Chemical Physics Letters, 198 (1992), pp. 89–94.

[11] J.-A. BOARD, Z.-S. HAKURA, W.-D. ELLIOTT, D.-C. GRAY, W.-J. BLANKE ET J.-F. LEATHRUM, *Scalable Implementations of Multipole-Accelerated Algorithms for Molecular Dynamics.*, Scalable Hight Performance Computing Conference (SHPCC94), (1994), p. 87. IEEE Computer Society Press. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-002.ps.

[12] J.-A. BOARD, Z.-S. HAKURA, W.-D. ELLIOTT ET W.-T. RANKIN, *Scalable Variants of Multipole-Accelerated Algorithms for Molecular Dynamics Applications.*, Rapp. Tech. 94-006, Duke University, Departement of Electrical Engineering, P.O.

Box 90291 Durham, NC 27708-0291, 1994. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-006.ps.

[13] K. BOEHMCKE, H. HELLER, H. GRUBMÜLLER ET K. SCHULTEN, *Molecular Dynamics Simulations on a Systolic Ring of Transputers.*, Transputer Research and Applications 3, (1990), pp. 83–94. Washington DC 1990 NATUG, IOS Press.

[14] J. BRIAT, J. CHASSIN DE KERGOMMEAUX, B. PLATEAU, J.-L. ROCH, D. TRYSTRAM, G. VILLARD ET J.-M. VINCENT, *APACHE : Algorithmique Parallèle et pArt age de CHargE*, dans Actes des 5^{iemes} Rencontres sur le Parallél isme - RenPar5, Brest, Mai 1993.

[15] J. BRIAT, I. GINZBURG, M. PASIN ET B. PLATEAU, *Athapascan runtime: efficiency for irregular problems.*, dans EURO-PAR'97, août 1997.

[16] B.-R. BROOKS, R.-E. BRUCCOLERI, B.-D. OLAFSEN, D.-J. STATES, S. SWAMINATHAN ET M. KARPLUS, *CHARMM : A Program for Macromolecular Energy, Minimization, and Dynamics Calculations.*, Journal of Computational Chemistry, 4 (1983), pp. 187–217.

[17] C.-L. BROOKS III ET M. KARPLUS, *Deformable stochastic boundaries in molecular dynamics.*, Journal of Chemical Physics., 79 (1983), pp. 6312–6325.

[18] D. BROWN ET J.-H.-R. CLARKE, *A Loose-coupling, Constant-pressure, Molecular Dynamics Algorithm for Use in the Modelling of Polymer Materials.*, Computer Physics Communications, 62 (1991), pp. 360–369.

[19] D. BROWN, J.-H.-R. CLARKE, M. OKUDA ET T. YAMAZAKI, *A domain decomposition parallelization strategy for molecular dynamics simulations on distributed memory machines.*, Computer Physics Communications., 74 (1993), pp. 67–80.

[20] ——, *A domain decomposition parallel processing algorithm for molecular dynamics simulations of polymers.*, Computer Physics Communications., 83 (1994), pp. 1–13.

[21] D. BROWN, H. MINOUX ET B. MAIGRET, *A domain decomposition parallel processing algorithm for molecular dynamics simulations of systems of arbitrary connectivity.*, Computational Physics Communications., 103 (1997), pp. 170–186.

[22] M. BRUCHNER ET B.-M. LADANYI, *Molecular Dynamics Algorithm for Flexible Molecules Using Normal Coordinates.*, Molecular Physics, 73 (1991), pp. 1127–1143.

[23] A.-T. BRÜNGER, *X-PLOR A System for X-ray Crystallography and NMR*, Yale University Press, 1992.

[24] A.-T. BRÜNGER, C.-L. BROOKS III ET M. KARPLUS, *Stochastic Boundary Conditions for Molecular Dynamics Simulations of ST2 Water.*, Chemical Physics Letters, 105 (1984), pp. 495–500.

[25] M. BUBAK, J. MOSCINSKI, M. POGODA ET R. SLOTA, *Load Balancing for Lattice Gas and Molecular Dynamics Simulations on Networked Workstations.*, Hight-Performance Computing and Networking, (1995), pp. 329–334.

[26] C. CALVIN, *Minimisation du sur-coût des communications dans la parallélisation des algorithmes numériques.*, thèse en mathématiques appliquées, Institut National Polytechnique de Grenoble, juil. 1995.

[27] C. CALVIN ET L. COLOMBET, *Introduction à PVM et MPI.*, Rapport APACHE 12, Rapport APACHE, juil. 1994. Available via ftp://ftp.imag.fr/pub/APACHE/RAPPORTS/APACHE_RR12.ps.gz.

[28] A. CANNING, G. G. ANF F. MAURI, A. DE VITA ET R. CAR, *O(N) Tight-Binding Molecular Dynamics on Massively Parallel Computers: an Orbital Decomposition Approach.*, Computer Physics Communications, 94 (1996), pp. 89–102.

[29] J. CASULLERAS ET E. GUARADIA, *Computer Simulation of Liquid Methanol I. Molecular Dynamics on a Supernode Transputer Array.*, Molecular Simulation, 7 (1991), pp. 155–169.

[30] C. CHANG, A. SUSSMANN ET J. SALTZ, *Support for Distributed Dynamic Data Structures in C++.*, Rapp. Tech. CS-TR-3416 and UMIACS-TR-95-19, University of Maryland, Department of Computer Science and UMIACS, College Park, MD 20742, jan. 1995. Available via ftp://hpsl.cs.umd.edu /pub/papers/chaos++.ps.

[31] M. CHRISTALLER, ATHAPASCAN-0A *control parallelism approach o n top of pvm*, dans Proc PVM User's group meeting, University of Tennessee, Oak Ridge, juin 1994.

[32] ——, ATHAPASCAN-0A *sur PVM 3 : définition et mode d'emploi.*, Rapp. Tech. Apache TR-11, IMAG, juin 1994.

[33] ——, *Vers un Support d'Exécution Portable pour Applications Parallèles Irrégulières : Athapascan-0.*, thèse en informatique, Université Joseph Fourier, Grenoble I, nov. 1996.

[34] M. CHRISTALLER, M.-R. CASTANEDA RETIZ ET T. GAUTIER, *Control parallelism on top of PVM: The* ATHA PASCAN *environment*, dans Proc. Second European PVM User's Group Meeting, J. Dongarra, M. Gengler, B. Tourancheau et X. Vigouroux, eds., Ecole Nationale Supérieure, Lyon, France, sept. 1995, Hermes, pp. 71–76.

[35] T.-W. CLARK, R. V. HANXLEDEN, K. KENNEDY, C. KOELBEL ET L.-R. SCOTT, *Evaluating Parallel Languages for Molecular Dynamics Computations.*, Scalable Hight Performance Computing Conference (SHPCC92), (1992), pp. 26–29. IEEE Computer Society Press.

[36] T.-W. CLARK, R. V. HANXLEDEN, J.-A. MCCAMMON ET L.-R. SCOTT, *Parallelizing Molecular Dynamics Using Spatial Decomposition.*, Scalable Hight-Performance Computing Conference (SHPCC94), (1994), p. 95. IEEE Computer Society Press. Available via ftp://softlib.rice.edu /pub/CRPC-TRs/reports/CRPC-TR93356-S.ps.

[37] E.-G. COFFMAN, *Bounds on the Performance of Scheduling Algorithms, Computer and Job/Shop Scheduling Theory.*, John Wiley, New York, 1976. Chap. 5.

[38] L. COLOMBET, *Parallélisation d'applications pour les réseaux de processeurs homogènes ou hétèrogènes.*, thèse en informatique, Institut National Polytechnique de Grenoble, oct. 1994.

[39] M. COSNARD ET Y. ROBERT, *Algorithmique Parallèle : une étude de Complexité.*, Technique et Science Informatiques, 6 (1987), pp. 115–125.

[40] M. COSNARD ET D. TRYSTRAM, *Algorithmes et Architectures Parallèles*, Inter-Editions, Collection IIA, 1993.

[41] R. DAS ET J. SALTZ, *Parallelizing Molecular Dynamics Codes Using Parti Software Primitives.*, dans Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, mars 1993, pp. 187–192. Available via ftp://hpsl.cs.umd.edu /pub/papers/siam93.ps.

[42] R. DAS, M. UYSAL, J. SALTZ ET Y.-S. HWANG, *Communication Optimisations for Irregular Scientific Computations on Distributed Memory Architectures.*, Journal of Parallel and Distributed Computing, 22 (1994), pp. 462–479. Available via ftp://hpsl.cs.umd.edu /pub/papers/scalability.ps.

[43] K.-M. DECKER, *A Methodology for Design and Implementation of Efficient Algorithms for Scalable Parallel Architectures.*, rapp. tech., IAM, University of Berne, Länggassstrasse 51, CH-3012 Berne SWITZERLAND, nov. 1991.

[44] R. DIEKMANN, D. MEYER ET B. MONIEN, *Parallel Decomposition of Unstructured FEM-Meshes.*, Lecture Notes in Computer Science, 980 (1995), p. 199.

[45] H.-Q. DING, N. KARASAWA ET W.-A. GODDARD III, *Atomic Level Simulation on a Million Particles : The Cell Multipole Method for Coulomb and London Nonbond Interactions.*, Journal of Chemical Physics, 97 (1992), pp. 4309–4315.

[46] ——, *Optimal Spline Cutoffs for Coulomb and Van der Waals Interactions.*, Chemical Physics Letters, 193 (1992), pp. 197–201.

[47] J. DONGARRA ET AL., *MPI: A Message-Passing Interface Standard.*, University of Tennessee, Knoxville, Tennessee., mai 1994. Available via http://www.netlib.org/mpi/mpi-report.ps.

[48] M. DOREILLE, F. GALILÉE ET J.-L. ROCH, *Athapascan-1b : Présentation.*, Rapport APACHE, LMC-IMAG Grenoble France, 1996. Available via http://navajo.imag.fr/ath1/.

[49] ——, *Construction dynamique du graphe de flot de données en Athapascan.*, dans Actes RenPar9, Lausanne, mai 1997.

[50] W.-D. ELLIOTT, *Revisiting the Fast Multipole Algorithm Error Bounds.*, Rapp. Tech. 94-008, Duke University, Departement of Electrical Engineering, P.O. Box 90291 Durham, NC 27708-0291, 1994. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-008.ps.

[51] W.-D. ELLIOTT ET J.-A. BOARD, *Fast Fourier Transform Accelerated Fast Multipole Algorithm.*, Rapp. Tech. 94-001, Duke University, Departement of Electrical Engineering, P.O. Box 90291, Durham, NC 27708-0291, mars 1994. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-001.ps.

[52] ——, *Fast Multipole Algorithm for the Lennard-Jones Potential.*, Rapp. Tech. 94-005, Duke University, Departement of Electrical Engineering, P.O. Box 90291, Durham, NC 27708-0291, août 1994. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-005.ps.

[53] A. FAGOT ET J. CHASSIN DE KERGOMMEAUX, *Optimized execution replay mechanism for RPC-based parallel programming models.*, Rapport APACHE 18, LMC-IMAG Grenoble France, 1995.

[54] A. FERRARI ET V.-S. SUNDERMAN, *TPVM: Distributed concurrent computing with lightweight processes.*, dans IEEE High Performance Computing, I. Press, ed., Washington D.C., 1995, pp. 211–218.

[55] D. FINCHAM, *Parallel Computers and Molecular Simulation.*, Molecular Simulation, 1 (1987), pp. 1–45.

[56] M.-J. FLYNN, *Somme Computer Organization and their effectiveness.*, IEEE Transaction on Computer, (1979), pp. 948–960.

[57] S. FORTUNE ET J. WYLLIE, *Parallelism in random access machines.*, dans STOC, 1978, pp. 114–118.

[58] I. FOSTER, C. KESSELMAN, R. OLSON ET S. TUECKE, *Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems.*, Rapp. Tech. ANL/MCS-TM-189, Argonne National Laboratory, USA, 1994.

[59] I. FOSTER, C. KESSELMANN ET S. TUECKE, *The* NEXUS *Approach to Integrating Multithreading and Communication.*, dans Parallel Programming Environments for High Performancs Computing, ESPPE'96, avr. 1996, pp. 53–67.

[60] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK ET V. SUNDERA, *PVM A User Guide, and Tutorial for Networked Parallel Computing.* Available via ftp://netlib2.cs.utk.edu/pvm3/book/pvm-book.ps.

[61] R.-L. GRAHAM, *Bounds on Multiprocessing Timing Anomalies.*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[62] D.-C. GRAY, *Load Balancing the Parallel Fast Multipole Algorithm.*, Rapp. Tech. 94-003, Duke University, Departement of Electrical Engineering, P.O. Box 90291, Durham, NC 27708-0291, avr. 1994. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-003.ps.

[63] D.-G. GREEN, K.-E. MEACHAM ET F. VAN HOESEL, *Parallelization of the Molecular Dynamics Code GROMOS87 for Distributed Memory Parallel Architectures.*, Hight-Performance Computing and Networking, (1995), pp. 875–879.

[64] L. GREENGARD ET V. ROKHLIN, *A Fast Algorithm for Particule Simulations.*, Journal of Computational Physics, 73 (1987), pp. 325–348.

[65] ——, *Rapid Evaluation of Potential Fields in Three Dimensions.*, Research Report YALEU/DCS/RR-515, Yale University, Departement of Computer Science, 51 Prospect Street, P.O. Box 2158 Yale Station, New Haven, Connecticut 06520, fév. 1987.

[66] ——, *On the Efficient Implementation of the Fast Multipole Algorithm.*, Research Report YALEU/DCS/RR-602, Yale University, Departement of Computer Science, 51 Prospect Street, P.O. Box 2158 Yale Station, New Haven, Connecticut 06520, fév. 1988.

[67] H. GRUBMÜLER, H. HELLER, A. WINDEMUTH ET K. SCHULTEN, *Generalized Verlet Algorithm for Efficient Molecular Dynamics Simulations with Long-Range Interactions.*, Molecular Simulation, 6 (1991), pp. 121–142.

[68] H. GRUBMÜLLER, *Predicting Slow Structural Transitions in Macromolecular Systems: Conformational Flooding.*, Physical Review E, 52 (1995), pp. 2893–2906.

[69] F. GUINAND, *Ordonnancement avec Communications pour Architecures Multiprocesseurs dans Divers Modèles d'Exécution.*, thèse en informatique, Institut National Polytechnique de Grenoble, juin 1995.

[70] I. GUINZBURG, *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications.*, thèse en informatique, Institut National Polytechnique de Grenoble, sept. 1997.

[71] H. HEKKER, E.-J. DIJKSTRA ET H.-J.-C. BERENDSEN, *Mapping Molecular Dynamics Simulation Calculation on a Ring Architecture.*, Parallel Computing: From Theory to Sound Practice, (1992), pp. 269–279.

[72] H. HELLER, H. GRUBMÜLLER ET K. SCHULTEN, *Molecular Dynamics Simulation on a Parallel Computer.*, Molecular Simulation, 0 (1989), pp. 0–33.

[73] B. HENDRICKSON ET S. PLIMTON, *Parallel Many-Body Simulations Without All-to-All Communication.*, rapp. tech., Sandia National Laboratories, Albuquerque, NM 87185, 1994.

[74] P.-A.-J. HILBERS ET K. ESSELINK, *Parallel Molecular Dynamics.*, Parallel Computing: From Theory to Sound Practice, (1992), pp. 288–299.

[75] D. HUTCHESON ET J. HUTCHESON, *Le Futur de la Micro-électronique.*, Pour la science, (1996), pp. 50–57.

[76] Y.-S. HWANG, R. DAS, J. SALTZ, M. HODOSCEK ET B.-R. BROOKS, *Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support Library.*, Rapp. Tech. CS-TR-3374 and UMIACS-TR-94-125, University of Maryland, Department of Computer Science and UMIACS, College Park, MD 20742, nov. 1994. To appear in IEEE Computational Science and Engineering. Available via ftp://hpsl.cs.umd.edu /pub/papers/charmm.ps.

[77] Y.-S. HWANG, B. MOON, S.-D. SHARMA, R. PONNUSAMY, R. DAS ET J. SALTZ, *Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines.*, SP&E, 25 (1995). To appear. Available via ftp://hpsl.cs.umd.edu /pub/papers/spe95.ps.

[78] A. JAIN, N. VAIDEHI ET G. RODRIGUEZ, *A Fast Recursive Algorithm for Molecular Dynamics Simulation.*, Journal of Computational Physics, 106 (1993), pp. 258–268.

[79] D. JANEZIC ET F. MERZEL, *An Efficient Sympletic Integration Algorithm for Molecular Dynamics Simulations.*, Journal of Chemical Information and Computer Sciences, 35 (1995), pp. 321–326.

[80] J.-F. JONAK ET P.-C. PATTNAIK, *Protein Calculations on Parallel Processors. I. Parallel Algorithm for the Potential Energy.*, Journal of Computational Chemistry, 13 (1992), pp. 533–538.

[81] ——, *Protein Calculations on Parallel Processors. II. Parallel Algorithm for the Forces and Molecular Dynamics.*, Journal of Computational Chemistry, 13 (1992), pp. 1098–1102.

[82] D.-M. JONES ET J.-M. GOODFELLOW, *Parallelization Startegies for Molecular Simulation Using the Monte Carlo Algorithm.*, Journal of Computational Chemistry, 14 (1993), pp. 127–137.

[83] S. KRISHNAN ET L.-V. KALÉ, *A Parallel Adaptive Fast Multipole Algorithm for N-Body Problems.*, dans Internatianal Conference on Parallel Processing, vol. III Algorithms & Applications, 1995, pp. 46–50.

[84] V. LAKAMSANI, L.-N. BHUYAN ET D. SCOTT LINTHICUM, *Mapping Molecular Dynamics Computations on to Hypercubes.*, Parallel Computing, 21 (1995), pp. 993–1013.

[85] C.-G. LAMBERT, *Multipole-Based Algorithms for Efficient Calculation of Forces and Potentials in Macroscopic Periodic Assemblies of Particles.*, Rapp. Tech. 94-004, Duke University, Department of Electrical Engineering, P.O. Box 90291, Durham, NC 27708-0291, mai 1994. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR94-004.ps.

[86] R. LAVERY, *Junctions and Bends in Nucleic Acids : A New Theoretical Modeling Approch.*, Structure & Expression, 3 (1988), pp. 191–211.

[87] J.-F. LEATHRUM ET J.-A. BOARD, *The Parallel Fast Multipole Algorithm in Three Dimensions.*, rapp. tech., Duke University, Departement of Electrical Engineering, P.O. Box 90291 Durham, NC 27708-0291, avr. 1992. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/PFMA_TR.ps.

[88] J. LI, A. BRASS, D.-J. WARD ET B. ROBSON, *A Study of Parallel Molecular Dynamics Algorithms for N-Body Simulations on a Transputer System.*, Parallel Computing, 14 (1990), pp. 211–222.

[89] L.-K. LIEBROCK ET K. KENNEDY, *Modeling Parralel Computation.*, Rapp. Tech. CRPC TR94499, Rice University Center for Research on Parallel Computation, 6100 South Main Street, Houston, TX 77005-1892, mai 1994. Available via ftp://softlib.rice.edu /pub/CRPC-TRs/reports/CRPC-TR94499.ps.

[90] S.-L. LIN, J. MELLOR-CRUMMERY, B.-M. PETTITT ET G. PHILLIPS, JR., *Molecular Dynamics on a Distributed-Memory Multiprocessor.*, Journal of Computational Chemistry, 13 (1992), pp. 1022–1035.

[91] P.-S. LOMDAHL, D.-M. BEAZLEY, P. TAMAYO ET N. GRONBECH-JENSEN, *Multi-Million Particle Molecular Dynamics on the CM-5.*, International Journal of Modern Physics., 0 (1992). LBTG92lll.

[92] P.-S. LOMDAHL, P. TAMAYO, N. GRONBECH-JENSEN ET D.-M. BEAZLEY, *50 GFlops Molecular Dynamics on the Connection Machine 5.*, 1993.

[93] E. MAILLET, *Le Traçage Logiciel d'Applications Parallèles : Conception et Ajustement de Qualité.*, thèse en informatique, Institut National Polytechnique de Grenoble, sept. 1996.

[94] A.-M. MATHIOWETZ, A. JAIN, N. KARASAWA ET W.-A. GODDARD III, *Protein Simulations Using Techniques Suitable for Very Large Systems: The Cell Multipole Method for Nonbond Interactions and the Newton-Euler Inverse Mass Operator Method for Internal Coordinate Dynamics.*, Proteins: Structure, Function, and Genetics, 20 (1994), pp. 227–247.

[95] A.-K. MAZUR ET R.-A. ABAGYAN, *New Methodology for Computer-Aided Modelling of Biomolecular Structure and Dynamics. 1. Non-Cyclic Structures.*, Journal of Biomolecular Structure & Dynamics, 6 (1989), pp. 815–832.

[96] J.-E. MERTZ, D.-J. TOBIAS, C.-L. BROOKS III ET U.-C. SINGH, *Vector and Parallel Algorithms for the Molecular Dynamics Simulation of Macromolecules on Shared-Memory Computers.*, Journal of Computational Chemistry, 12 (1991), pp. 1270–1277.

[97] B. MOON, G. PATNAIK, R. BENNETT, D. FYFE, A. SUSSMAN, C. DOUGLAS, J. SALTZ ET K. KAILASANATH, *Runtime Support and Dynamic Load Balancing Strategies for Structured Adaptive Applications.*, dans Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, SIAM, fév. 1995, pp. 575–580. Available via ftp://hpsl.cs.umd.edu /pub/papers/siam95.ps.

[98] F. MUELLER, *A library implementation of POSIX threads under UNIX.*, dans Proc. of the Winter USENIX Conference, San Diego, CA, jan. 1993, pp. 29–41.

[99] F. MÜLLER-PLATHE, W. SCOTT ET W.-F. VAN GUNSTEREN, *Molecular Dynamics on Supercomputers : Implementations and Applications.*, SPEEDUP, 6 (1992), pp. 33–38.

[100] R. NAMYST, *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.*, thèse en informatique, Université des Sciences et Technologies de Lille, 1997.

[101] R. NAMYST ET J.-F. MEHAUT, *PM2 Parallel Multithreaded Machine A multithreade d environement on top of PVM*, dans EuroPVM'95, J. Dongarra et al., eds., Hermes, sept. 1995, pp. 179–184.

[102] C. NIEDERMEIER ET P. TAVAN, *A Structure Adapted Multipole Method for Electrostatic Interactions in Protein Dynamics.*, Journal of Chemical Physics, 101 (1994), pp. 734–748.

[103] *Special Issue: High Performance Fortran Language Specification.*, ACM Fortran Forum, 13 (1994).

[104] OBSERVATOIRE FRANÇAIS DES TECHNIQUES AVANCÉES, *Ordinateurs et calcul parallèles.* ARAGO 19, 1997.

[105] M.-R.-S. PINCHES ET D.-J. TILDESLEY, *Large Scale Molecular Dynamics on Parallel Computers Using the Link-Cell Algorithm.*, Molecular Simulation, 6 (1991), pp. 51–87.

[106] S. PLIMPTON, *Fast Parallel Algorithms for Short-Range Molecular Dynamics.*, Journal of Computational Physics, 117 (1995), pp. 1–19.

[107] S. RAINA, *Virtual Shared Memory: A Survey of Techniques and Systems.*, Available by http: www.pact.srf.ac.uk /DDM/ps/cstr-92-36.ps CSTR-92-36, University of Bristol, Dept. of Computer Science, Bristol BS8 1TR, U.K., déc. 1992.

[108] A.-R.-C. RAINE, *Systolic Loop Methods for Molecular Dynamics Simulation, Generalised for Macromolecules.*, Molecular Simulation, 7 (1991), pp. 59–69.

[109] S. RANKA, J.-C. WANG ET G. FOX, *Static and Runtime Algorithms for All-to-Many Personalised Communication on Permuion Networks.*, Rapp. Tech. CRPC TR94501, Rice University Center for Research on Parallel Computation, 6100 South Main Street, Houston, TX 77005-1892, juin 1994. To apear in IEEE Transactions on Parallel and Distributed Systems. Available via ftp://softlib.rice.edu /pub/CRPC-TRs/reports/CRPC-TR94501.ps.

[110] W.-T. RANKIN ET J.-A. BOARD, *A Portable Distributed Implementation of the Parallel Multipole Tree Algorithm.*, Rapp. Tech. 95-002, Duke University, Departement of Electrical Engineering, P.O. Box 90291 Durham, NC 27708-0291, 1995. Available via ftp://ftp.ee.duke.edu /pub/SciComp/papers/TR95-002.ps.

[111] L.-M. RICE ET A.-T. BRÜNGER, *Torsion Angle Dynamics : Reduced Variable Conformational Sampling Enhances Crystallographic Structure Refinement.*, Proteins : Structure, Function, and Genetics, 19 (1994), pp. 277–290.

[112] M. RIVIERE, *Concepts structurants pour la mise en oeuvre d'applications irrégulières : Application au support exécutif parallèle Athapascan0$_{mp}$.*, thèse en informatique, Institut National Polytechnique de Grenoble, sept. 1997.

[113] J.-L. ROCH ET D. TRYSTRAM, *Méthodologies pour la Programmation Efficace d'Applications Parallèles.*, Calculateurs Parallèles, 6 (1994), pp. 133–138.

[114] J.-L. ROCH, A. VERMEERBERGEN ET G. VILLARD, *A new load-prediction scheme based on arithmetic cost functions.*, dans CONPAR 94, Linz Austria, Springer-Verlag, ed., LNCS 854, sept. 1994.

[115] J. SALTZ, R. PONNUSAMY, S.-D. SHARMA, B. MOON, Y.-S. HWANG, M. UYSAL ET R. DAS, *A Manual for the CHAOS Runtime Library.*, Rapp. Tech. CS-TR-3437 and UMIACS-TR-95-34, University of Maryland, Department of Computer Science and UMIACS, College Park, MD 20742, mars 1995. Available via ftp://hpsl.cs.umd.edu /pub/papers/chaos-manual.ps.

[116] D.-B. SHMOYS, J. WEIN ET D.-P. WILLIAMSON, *Scheduling Parallel Machines On-Line*, SIAM J. Comput., 24 (1995), pp. 1313–1331.

[117] J.-P. SINGH, C. HOLT, T. TOTSUKA, A. GUPTA ET J. HENNESSY, *Load Balancing and Data Locality in Adaptive Hierarchical N-Body Methods: Barnes-Hut, Fast Multipole and Radiosity.*, Journal of Parallel and Distributed Computing, 27 (1995), pp. 118–141.

[118] D.-B. SKILLICORN, J.-M.-D. HILL ET W.-F. MCCOLL, *Questions and Answers about BSP.*, Rapp. Tech. PRG-TR-15-96, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1996.

[119] M. SPRIK, *Running Molecular Dynamics Code in Parallel on a Cluster of Workstations.*, SPEEDUP, 6 (1992), pp. 57–61.

[120] P. STEINBACH ET B.-R. BROOKS, *New Spherical-Cutoff Methods for Long-Range Forces in Macromolecular Simulation.*, Journal of Computational Chemistry, 15 (1994), pp. 667–683.

[121] W.-B. STREETT, D.-J. TILDESLEY ET G. SAVILLE, *Multiple time step methods in molecular dynamics.*, Molecular Physics, 35 (1978), pp. 639–648.

[122] D. SUGIMOTO, Y. CHIKADA, J. MAKINO, T. ITO, T. EBISUZAKI ET M. UMEMURA, *A Special-purpose Computer for Gravitational Many-body Problems.*, Nature, 345 (1990), pp. 33–35.

[123] F. TEODORESCU ET J. CHASSIN DE KERGOMMEAUX, *On Correcting the Intrusion of Tracing non Deterministic Programs by Software.*, dans Proceedings of EuroPar'97, LNCS, Springer Verlag, août 1997.

[124] Y. TRÉMOLLET, *Parallélisation d'algorithmes variationnels d'assimilation de données en météorologie.*, thèse en mathèmatiques appliquées, Université Joseph Fourier, Grenoble I, nov. 1995.

[125] R. TROBEC, I. JEREBIC ET D. JANEZIC, *Parallel Algorithm for Molecular Dynamics Integration.*, Parallel Computing, 19 (1993), pp. 1029–1039.

[126] C. TRON, *Modèles quantitatifs de machines parallèles : les réseaux d'interconnexion.*, thèse en informatique, Institut National Polytechnique de Grenoble, nov. 1994.

[127] L.-G. VALIANT, *A Bridging Model for Parallel Computation.*, Communications of the ACM, 33 (1990), pp. 103–111.

[128] R. VAN DRIESSCHE ET D. ROOSE, *Dynamic Load Balancing with a Spectral Bisection Algorithm for the Constrained Graph Partitioning Problem.*

[129] W.-F. VAN GUNSTEREN ET H.-J.-C. BERENDSEN, *Algorithms for Macromolecular Dynamics and Contraint Dynamics.*, Molecular Physics, 34 (1977), pp. 1311–1327.

[130] M.-H. WILLEBEEK-LEMAIR ET A.-P. REEVES, *Strategies for Dynamic Load Balancing on Highly Parallel Computers.*, IEEE Transactions on Parallel and distributed systems, 4 (1993), pp. 979–993.

[131] A. WINDEMUTH, *Advanced Algorithms for Molecular Dynamics Simulation: The Program* PMD., dans Parallel Computing in Computational Chemistry, A. Symposium, ed., 1995. Available via ftp://cumbnd.bioc.columbia.edu/pmd/pmd.ps.

[132] A. WINDEMUTH ET K. SCHULTEN, *Molecular Dynamics on the Commection Machine.*, Molecular Simulation, 5 (1991), pp. 353–361.

Résumé :

De nombreuses méthodes de calcul numérique parallèle sont développées pour utiliser les super-ordinateurs d'aujourd'hui, mais ces méthodes utilisent rarement les mécanismes de régulation et perdent en efficacité sur des problèmes non structurés. En particulier, la simulation numérique, par dynamique moléculaire (DM), du mouvement des atomes des protéines dans les structures biologiques est un de ces problèmes irréguliers qui demande beaucoup de puissance de calcul.
Cette thèse, à travers la réalisation d'un programme parallèle de DM opérationnel pour l'étude des protéines, s'intéresse à montrer l'apport des processus légers pour la parallélisation de ce type d'application. Ce travail s'inscrit dans le projet INRIA-IMAG APACHE de réalisation du support d'exécution ATHAPASCAN pour les applications parallèles irrégulières et en collaboration avec le laboratoire BMC du CEA-GRENOBLE.
Après une introduction sur la parallélisation d'applications, nous présentons les concepts de base de la parallélisation par des processus légers et des échanges de messages. La deuxième partie du document propose une synthèse des éléments du modèle de DM pour l'étude des protéines et présente la méthode du rayon de coupure pour l'approximation des forces. Dans la suite nous proposons une parallélisation de cette méthode du rayon de coupure par décomposition du domaine de simulation. Nous étudions plusieurs stratégies de placement des calculs afin de trouver le meilleur compromis entre l'équilibre de charge et la minimisation des communications. Puis nous montrons comment la multiprogrammation permet de recouvrir les communications par des calculs. Enfin nous terminons en proposant un mécanisme d'équilibre dynamique de la charge de calcul. De nombreuses mesures sur le IBM-SP1 et le CRAY-T3E sont présentées et montrent l'extensibilité de nos algorithmes parallèles.

Mots clés : parallélisme, dynamique moléculaire, processus légers, placement, répartition de charge.

Abstract:

Many numerical parallel methods are implemented to use the modern super computer, but those methods rarely use the technics of regulation and lose their efficiency on unstructured problems. In particular, the numerical simulation of the motion of atoms inside biological structures is one of the irregular problems that need many computational power.
The subject concerned in this thesis, through the achievement of a parallel molecular dynamics (MD) software to study proteins, show how threads (or light-weight processes) are useful to parallelize such an application. This work take part of the INRIA-IMAG APACHE project that carry out ATHAPASCAN, a runtime system for the irregular parallel applications, and in collaboration with the laboratory BMC CEA-GRENOBLE.
After an introduction on parallelization technics for applications, we present basic concept of a parallelization technics using threads and message passing. The second part of this document give a synthetic view of the MD models used to study proteins and presents the cut off radius method to approximate forces. Next we propose a parallel implementation of this cut off method using a spatial decomposition of the domain. We study some mapping strategies to find the better compromise between load balancing and communication minimization. Then, we show how multiprogramming is able to overlap communication with computation. Finally, we propose a dynamic load balancing technics of computational load. Many performance measurement on the IBM-SP1 and on the CRAY-T3E are done, and they show scalability of our parallel algorithms.

Keywords: parallelism, molecular dynamics, threads, mapping, load balancing.