Calcul Haute-Performance et Mécanique Quantique : analyse des ordonnancements en temps et en mémoire

Nicolas Maillard

UNIVERSITÉ GRENOBLE 1 – JOSEPH FOURIER

T H È S E

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Études Doctorale : « Mathématiques Appliquées, parallélisme »

présentée par

Nicolas MAILLARD

Calcul Haute-Performance et Mécanique Quantique : analyse des ordonnancements en temps et en mémoire

---

Directeur de thèse :

Pierre VALIRON

co-Directeur de thèse :

Jean-Louis ROCH

Composition du Jury :

Pierre	MANNEBACK, *Rapporteur*
Yves	ROBERT, *Rapporteur*
Jean-Louis	ROCH, *Co-directeur de thèse*
Pierre	VALIRON, *Directeur de thèse*
Marie-Madeleine	ROHMER, *Examinatrice*
Laurent	DESBAT, *Président*

Thèse préparée au Laboratoire Informatique et Distribution
dans le cadre de l'École Doctorale
« MATHÉMATIQUES, SCIENCES ET TECHNOLOGIE de l''INFORMATION, INFORMATIQUE »

Remerciements

Je croyais ne jamais arriver à cette phase de la thèse : l'écriture des remerciements. Il aurait pourtant été dommage de ne pouvoir le faire, tellement il y a de gens qui m'ont aidé au cours de ces cinq années de travail.

Je voudrais commencer par remercier les membres de mon jury : Yves Robert et Pierre Manneback ont donné leur confiance à mon travail et j'ai beaucoup apprécié leurs conseils précieux pour l'améliorer ; Laurent Desbat, qui fut le premier à me parler de "supercalculateur" à l'Ensimag, m'a fait très plaisir en participant à mon jury, de même que Marie-Madeleine Rohmer, qui a accepté d'apporter sa caution de physicienne à mon travail pluridisciplinaire.

Pierre Valiron a toujours su trouver des points intéressants quels qu'aient été les errements dans mon travail : ce fut une aide réelle de sa part, sans même évoquer les innombrables choses que cet esprit universel aura su m'apprendre, scientifiquement et humainement parlant. Jean-Louis Roch est certainement la personne qui m'a le plus appris dans les disciplines scientifiques que j'ai choisies : comme enseignant passionné et passionnant d'abord, comme directeur de thèse ensuite, comme collègue un jour peut-être, depuis 1993 il a plus contribué que tout autre à m'apprendre le métier d'ingénieur, puis de chercheur. Malgré les difficultés inévitables, il a — je crois — réussi dans sa démarche… et je l'en remercie.

Une chose est certaine : même si le travail avec ces deux directeurs de thêse n'a pas pris le tour que chacun aurait parfois voulu, ce n'est certainement pas par manque d'idée de leur part dans leurs domaines respectifs.

Cette thèse n'aurait tout simplement pas pu avoir lieu si un certain nombre de personnes n'avaient, un jour ou un autre, donné un coup de pouce, parfois sans même se douter combien leur rôle aura pu m'être précieux ! Merci en particulier à Khadija et à Hélène pour leurs aides administratives toujours efficaces ; à Joelle pour le soutien technique dans le labo ; et à Stellio dont la bonne humeur et la disponibilité remontent bien le moral quand les soirées se font longues dans le bureau.

J'ai eu la chance de participer à un travail très enrichissant avec l'équipe "top-500" : à un moment particulièrement difficile de mon travail, Stéphane, Céline (qui en plus a relu ce document !), Simon, Bruno et Philippe m'ont vraiment offert une oportunité unique de retrouver l'envie d'avancer et je les en remercie… sans même parler de leur bonne humeur, des footings avec les uns et des discussions avec les autres.

En parlant de discussions me vient aussitôt à l'esprit le nom de Gregory. Merci pour l'éternel sourire, les TD préparés ensemble, les années de thèse et le bon exemple qu'il me donne. Je ne peux non plus, à ce sujet, manquer d'évoquer tous les confrêres qui

sont passés au cours de cette thèse : dans le désordre, Alexandre, Marcelo, Ilan, Paulo, Martha, Olivier, Ekbal, Christophe, Alfredo, Benhur, Alex et Seb, . . . il y a un peu de vous tous dans la fin de ce travail. Merci aussi aux thésards plus "jeunes" du labo. Hélas je profiterai moins de vos travaux que je ne l'ai fait de ceux de vos prédecesseurs, mais j'ai bien apprécié la bonne humeur dont vous m'avez fait bénéficier.

À ID et ailleurs un certain nombre de chercheurs m'ont témoigné de la confiance et cela m'a aussi beaucoup aidé : merci à Jean-Marc Vincent pour les enseignements partagés et les riches discussions. Merci à Brigitte Plateau, Denis Naddef et Denis Trystram pour les conversations autour des TD pour l'Ensimag. Merci à Guy Fishman pour les cours de mécanique quantique.

Au sein d'ID il y a un groupe de travail P3. Avant qu'il n'existe on parlait plus simplement des "A1 boys". J'ai eu très peur d'être bon dernier mais heureusement, Rémi est arrivé. François, Mathias, Gerson, Thierry, Jean-Guillaume, Rémi. . . C'est avec vous que j'ai peut-être le plus interagi — merci pour tout. Je suis certain que nous n'avons pas fini d'entendre parler les uns des autres.

Chaque année universitaire les étudiants en fin de thèse vivent quelque chose de très spécial, entre les nuits au labo, les horaires décalés, les mêmes angoisses et les mêmes retards. Dans mon cas nous avons été quatre à enfoncer tous les délais et donc à avoir eu peut-être encore un peu plus de mal que la moyenne à terminer — et ainsi un peu plus de moments forts partagés. Au moment pour moi de terminer, j'ai une grosse pensée pour Roberta et Andréa et les remercie de leur soutien. Quant à Gustavo. . . son tour va venir.

Deux amis très proches ont entremêlés leurs parcours au mien, depuis l'Ensimag. Je pense ici à Sylvie et à Claude-Pierre. Je peux difficilement résumer 6 ou 7 ans d'amitié, de voyages, de discussions, de sorties, d'activités. . . et même de travail scientifique !. . . que nous avons eus ensemble. En tous cas sans vous je ne serais pas venu à bout de tout ça.

Je connais Mauricio, Tati et Kelly depuis un peu moins d'années mais le temps ne fait rien à l'affaire. . . Dans la dernière et si difficile année de thèse leur présence, leur joie de vivre quotidienne, leur accueil quand j'en avais besoin, m'ont toujours soutenu sans faille. On a même partagé nos langues respectives ! Sans prendre trop de risque, je crois que la suite des évènements devrait encore renforcer ces liens précieux. . . Et c'est très bien comme ça.

Il reste encore une personne. Gustavo, não me lembro da metade do quê que a gente falou durante esses quatro anos. . . mas com certeza foi a melhor parte da tese. "Procura-se um amigo para gostar dos mesmos gostos, que se comova, quando chamado de amigo. Que saiba conversar de coisas simples, de orvalhos, de grandes chuvas e das recordações da infância. Precisa-se de um amigo para não se enlouquecer, para contar o que se viu de belo e triste durante o dia, dos anseios e das realizações, dos sonhos e da realidade." (Vinícius) — e o melhor é que a gente vai continuar conversar aí !

Table des matières

Table des figures

Index

1

Introduction

L’informatique a historiquement été toujours employée par les scientifiques pour résoudre des problèmes de natures aussi variées que leurs disciplines. Les langages de programmation ont suivi leurs besoins pour permettre à la fois de formuler et de résoudre les problèmes. Ainsi Fortran fut développé par la communauté pour traiter facilement et efficacement les calculs, souvent matriciels, des physiciens. Le problème pour un programmeur est double : d’abord il s’agit de passer de l’application qu’il veut traiter, exprimée selon le formalisme propre à sa discipline, à un algorithme permettant de résoudre le problème de façon “efficace”, la notion d’efficacité étant définie selon les théories de complexité. Ensuite il faut passer de la formulation de l’algorithme choisi à sa programmation, son implantation sur une machine cible, dans un langage donné, pour atteindre les meilleures performances possibles sur la machine.

L’évolution des langages de programmation a été motivée par la recherche d’un meilleur cadre pour l’expression de l’algorithme choisi. Dernièrement les langages orientés objet ont commencé à s’imposer grâce aux fonctionnalités qu’ils offrent en terme d’abstraction des données, ce qui permet à la fois de faciliter la mise au point, la maintenance et la réutilisation du programme ; et en même temps de concevoir le programme plus directement selon les termes des algorithmes implantés : entre les structures de données manipulées par les algorithmes et leur implémentation au niveau de la machine, le langage offre une interface de plus en plus "naturelle" : il en va ainsi de la manipulation des tableaux depuis Fortran-90 ou des "containers" C++ qui fournissent en standard des types tels que les listes, les tables de hachage ou les piles. Le travail du compilateur et son efficacité permettent en théorie de générer du code sans surcoût dû à l’encapsulation de ces types abstraits.

Parallèlement aux évolutions dans les langages, les machines cibles se sont énormé-

ment modifiées. À la fin des années 70 les processeurs sont devenus vectoriels, puis parallèles. Les systèmes d'exploitation permettant le multitâche ou la concurrence de plusieurs processus sont apparus. Actuellement les machines sont constituées de plusieurs noeuds, chacun étant lui-même constitué de plusieurs processeurs à mémoire partagée. Les noeuds, de types éventuellement différents, sont interconnectés selon des réseaux éventuellement eux-mêmes différents, l'ultime étape actuelle étant l'utilisation d'Internet pour relier des grappes distinctes (projet Grille).

La programmation de telles machines, à plusieurs degrés de parallélisme, ajoute de nouveaux problèmes à ceux évoqués ci-dessus : cohérence des données en milieu distribué, accès concurrents dans une mémoire partagée, entrelacement des calculs et des communications garantissant la performance du programme parallèle, ordonnancement des tâches sur une machine hétérogène... Ces écueils sont d'autant plus cruciaux que les machines parallèles sont avant tout destinées à des applications demandant de gros volumes de mémoire et de temps CPU : la performance des programmes est donc prioritaire pour les utilisateurs qui cherchent du calcul haute performance.

Les solutions, en terme de modèles de programmation parallèle, ne sont aujourd'hui pas encore unanimement acceptées. Un standard utilisé *de facto* repose sur le paradigme de l'échange de messages. Les librairies PVM ou MPI offrent des fonctions de communication et de synchronisation entre les processeurs qui s'ajoutent à un langage séquentiel classique (C, C++, Fortran). Fortran, depuis ses versions 90 et 95, a évolué vers une approche "parallélisme de données", en offrant des primitives de distribution des données, les instructions séquentielles s'exécutant sur chaque processeur sur les données localement présentes (HPF). L'efficacité de cette approche semble limitée à certaines classes d'applications. En ce qui concerne la mémoire partagée, des solutions comme Open-MP permettent une programmation relativement simple de certaines applications grâce à une annotation du code séquentiel.

La difficulté réside également dans la complexité des machines employées : le nombre de paramètres les caractérisant augmente (temps d'accès aux différents niveaux de mémoire, temps d'accès aux données distantes, puissances différentes selon les noeuds de la machine...). La recherche d'un modèle assez simple pour permettre de résoudre les problèmes posés et le rendre universel, qui soit en même temps réaliste, reste une question ouverte. Les modèles PRAM ou BSP tentent d'apporter des réponses à ce problème. Cette complexité des machines est non seulement un écueil pour définir un modèle de programmation, mais également une difficulté pour l'obtention de performances.

Une solution pour traiter efficacement la parallélisation d'une application semble être de découpler *l'expression du parallélisme* dans l'algorithme de *l'ordonnancement* des tâches de calcul, pour une machine cible, selon les critères imposés par le programmeur (mémoire, temps de calcul,...). Ainsi il dispose à la fois d'un programme directement à l'image de son algorithme et d'un mécanisme (l'ordonnancement) pour l'adapter aux machines cibles. Les environnements de programmation parallèles qui permettent ce découplage ne sont malheureusement pas encore nombreux et la tâche revient souvent au

programmeur de coder à la fois l'algorithme et son ordonnancement, dans le même programme, limitant ainsi sa portabilité et éventuellement sa performance.

Organisation du document : Ce travail étudie dans ce contexte le traitement d'applications types rencontrées en mécanique et chimie quantique, en vue d'obtenir des programmes parallèles performants. Ce document est divisé en deux parties. La première présente la problématique et son contexte. Le premier chapitre introduit rapidement la théorie quantique et ses applications en mécanique et en chimie. Le problème type qui se présente est la résolution de l'équation de Schrödinger dans des conditions adaptées à chaque phénomène physique étudié. Cette équation est une équation aux valeurs propres et différents algorithmes numériques peuvent être envisagés pour la résoudre de façon approchée. Cette étude focalisera plus précisément sur des méthodes dites itératives, de type de Lanczos/Arnoldi. L'un des calculs de mécanique quantique où ce type d'algorithme est intéressant est celui de la "boîte quantique" BQ. En chimie également, la théorie quantique a de nombreuses applications. L'un des algorithmes de calcul en chimie, cible de ce travail, est la méthode des perturbations de deuxième ordre, MP2.

Pour les deux problèmes types identifiés dans le chapitre 2, il existe des solutions parallèles. Le chapitre 3 les introduit. Dans le cadre des méthodes itératives, la librairie mathématique Arpack fournit une implémentation parallèle basée sur l'échange de messages (MPI). MPI sert aussi de modèle de programmation pour des implémentations parallèles d'algorithmes de chimie quantique, SCF et MP2, comme dans le programme Asterix par exemple. Un autre environnement parallèle est par contre employé dans le logiciel commercial Gaussian : Linda, qui offre une mémoire virtuellement partagée au programmeur. Enfin, Open-MP peut être utilisé pour paralléliser des calculs de chimie quantique.

Tous ces environnements permettent l'obtention de performances, mais imposent des contraintes sur le programmeur : en terme de facilité de programmation ; de portabilité du code ; d'adaptabilité de l'ordonnancement. Le chapitre 4 présente l'environnement parallèle ATHAPASCAN et le modèle de programmation basé sur le flot de données qu'il propose. L'exemple de BQ est présenté en ATHAPASCAN . Enfin est étudié le problème des synchronisations nécessaires au contrôle de la convergence d'un algorithme itératif tel que celui utilisé pour BQ. Le calcul d'une norme sur un résidu, typiquement effectué à chaque itération, impose une synchronisation régulière aux tâches de calcul. Pour augmenter le parallélisme et donner de la liberté aux ordonnanceurs, il faut tenter de désynchroniser au maximum ces tâches. Dans certains cas, le changement de norme mathématique permet d'éviter le problème. Quand ce n'est pas possible, il faut envisager de relâcher le contrôle de l'algorithme.

La modélisation de l'exécution d'un programme (éventuellement) parallèle sous forme de flot de données et son interprétation par un environnement du type d'ATHAPASCAN permettent à la fois de découpler la programmation de l'algorithme de son ordonnancement sur une machine ; et de fournir à l'environnement les informations à même de

calculer un ordonnancement optimal en fonction des critères choisis. La deuxième partie de ce document traite de l'ordonnancement d'une application décrite sous forme de graphe de flot de données.

Étant donné le graphe qui modélise le flot de données, l'environnement ATHAPASCAN permet son interprétation afin d'en calculer des ordonnancements. La première grandeur que l'on peut chercher à optimiser est le temps d'exécution. Selon le modèle de machine et le type d'information que l'on a sur le graphe, diverses stratégies peuvent être appliquées avec des garanties de performance. On peut ainsi adapter, selon l'algorithme d'ordonnancement, l'exécution à la machine. Le chapitre 5 rappelle quelques modèles classiques de machine parallèle (PRAM pour les SMP, délai pour les machines à mémoire distribuée). Dans le cas où le graphe est connu dynamiquement, au cours de l'exécution, les meilleurs algorithmes sont de type liste. Dans la cas où l'on a une connaissance du graphe à la compilation, on montre que l'environnement ATHAPASCAN permet de retrouver des ordonnancements et des performances tout à fait similaires à ceux d'un programme MPI ou Open-MP. On montre enfin quelles performances différents ordonnancements permettent d'obtenir sur BQ, grâce à ATHAPASCAN .

L'autre critère fondamental pour les applications que l'on peut vouloir optimiser est la mémoire requise par l'exécution. Dans le cas de MP2 par exemple, le compromis entre espace mémoire et temps de calcul est critique car on peut diminuer le premier en recalculant certaines des données en entrée, au prix du temps d'exécution. Le graphe de flot de données peut être annoté pour tenir compte des allocations et libérations de mémoire effectuées par chaque tâche. Ceci permet de calculer des ordonnancements à la fois efficaces en temps et qui limitent le surcoût mémoire que peut générer le parallélisme des tâches. C'est l'objet du chapitre 6.

Contributions : les contributions principales de ce travail portent sur les aspects suivants :

- un travail de reformulation algorithmique de deux problèmes types de mécanique et chimie quantique (BQ et MP2), particulièrement adaptées à leur programmation efficace (chapitre 6 pour MP2 et 3 pour BQ) ;
- l'étude comparative de l'adéquation des environnements parallèles de calcul performant à ces deux problèmes, pour les machines parallèles actuelles. En particulier, un travail de benchmarking grâce au test Linpack, sur une grappe de PC, est présenté dans le chapitre 3 et dans l'annexe A ;
- l'identification des limitations de ces environnements et la présentation de l'environnement ATHAPASCAN qui permet de pallier ces limitations en découplant l'algorithme de son ordonnancement ;
- une proposition de modélisation de la mémoire utilisée par des algorithmes tels MP2 afin de pouvoir les ordonnancer en ayant pour objectif d'être performant à la fois en temps de calcul et en espace memoire utilisé (chapitre 6).

Ces contributions ont fait l'objet de quelques publications (référencées éventuellement par ailleurs dans le document) :

– [45], N. Maillard, J.-L. Roch et P. Valiron, *Parallélisation du calcul ab-initio de l'énergie de corrélation électronique par la méthode MP2* évoque le travail algorithmique effectué sur le calcul MP2 (*cf.* chapitre 6) ;
– [32], C.-P. Jeannerod et N. Maillard, *Using Computer Algebra to Diagonalize some Kane Matrices* résume un travail plus mathématique sur les méthodes de perturbations appliquées à des Hamiltoniens usuels en physique des semi-conducteurs (*cf.* section 2.3.1) ;
– [19], G. Fernandes, N. Maillard et Y. Denneulin, *Parallelizing a Dense Matching Region Growing Algorithm for an Image Interpolation Application* présente un travail de parallélisation sur un algorithme d'interpolation d'images. Ce travail n'apparaît pas dans ce document ;
– [56], B. Richard, P. Augerat, N. Maillard, S. Derr, S. Martin et C. Robert, *I-Cluster : Reaching TOP500 Performance Using Mainstream Hardware* effectue un compte-rendu du travail réalisé sur le benchmark Linpack pour l'entrée dans le top-500 de la grappe Icluster du laboratoire ID-IMAG (*cf.* le chapitre 3 ainsi que l'annexe A).

I

Mécanique quantique et parallélisme

La mécanique et la chimie quantique font appel à des algorithmes numériques spécialisés souvent très exigeants en temps de calcul et en mémoire, qui nécessitent donc du calcul à haute performance. Ce travail présente des techniques d'ordonnancement, en vue d'exécutions efficaces en temps et en mémoire sur machines parallèles, appliquées à ces algorithmes numériques pour la mécanique et chimie quantique.

Cette première partie positionne les problèmes à la fois en ce qui concerne la mécanique quantique et du point de vue du parallélisme.

Le premier chapitre introduit rapidement la théorie quantique, son formalisme et les problèmes qu'elle tente de résoudre. Les méthodes numériques utilisées sont brièvement présentées. Deux problèmes types sont présentés plus précisément : celui de la boîte quantique, qui fait appel selon notre analyse à des méthodes itératives de calcul de valeurs propres ; et MP2, un calcul de chimie quantique qui présente un intéressant problème de compromis calcul/mémoire.

Dans les deux cas, on trouve des environnements qui offrent des solutions intéressantes pour des implémentations parallèles efficaces. Le chapitre 2 présente tout d'abord une contribution personnelle sur le type de performances que l'on peut attendre d'une machine parallèle actuelle (grappe de 225 noeuds classée dans le top-500) sur le benchmark Linpack. Ensuite sont présentés quatre exemples de problèmes de mécanique/chimie quantique et des solutions parallèles : l'environnement Arpack/MPI pour BQ ; les calculs de SCF en MPI ; le calcul de MP2 en MPI de Nielsen et de Gaussian/Linda ; et enfin le calcul de Coupled-Cluster basé sur Open-MP.

Même si on trouve des solutions déjà efficaces sur certaines architectures, elles sont donc à la fois très peu portables en terme de performances et très contraignantes pour le programmeur en lui imposant des surcoûts de programmation au-delà de ce qui est requis par l'algorithme séquentiel. De plus, aucune garantie de performance n'est offerte par ces environnements.

Le chapitre 3 propose un autre environnement de programmation parallèle, Athapascan, qui offre une vision différente de l'application parallèle grâce à l'analyse du flot de données que constitue l'algorithme. C'est sous cet environnement que les techniques d'ordonnancement seront étudiées dans la partie suivante. Nous présentons également à la fin de ce chapitre un algorithme asynchrone de contrôle des synchronisations pour un algorithme itératif du type de celui utilisé pour BQ.

2

Physique et modélisation mathématique du problème

Ce chapitre introduit le contexte physique de ce travail : la mécanique et la chimie quantique. La mécanique quantique a été développée pendant quasiment tout le vingtième siècle, de façon théorique pendant sa première moitié, avant de se voir validée par les observations et les expériences. Les applications à la chimie ont rapidement été nombreuses à cause de l'échelle des phénomènes traités par la théorie quantique. La première section introduit le formalisme et les postulats de la théorie quantique. Résoudre l'équation de Schrödinger apparaît comme le problème central qui se pose au physicien ou au chimiste qui utilise cette théorie : nous exposons cette équation en 2.2 et nous l'illustrons sur deux cas simples (atome d'hydrogène et puits quantique) à la fois pour mieux faire comprendre ce dont il s'agit et parce que ces problèmes sont à la base des approximations utilisées dans la suite. Les deux sections suivantes focalisent la théorie sur les deux cas que ce travail traitera plus dans la suite : la résolution de l'équation de Schrödinger dans le cas de la boîte quantique (section 2.3) et les algorithmes spécialisés SCF et MP2 dans le cas de la chimie quantique *ab-initio* (section 2.4).

2.1 Postulats de la mécanique quantique

La mécanique quantique remplace la notion de position classique d'une particule par sa probabilité de présence en un point $r \stackrel{\text{def}}{=} (x, y, z)^t$ de l'espace. L'école de Copenhague a proposé une série de postulats fondant la modélisation mathématique du comportement d'un système de particules. Selon les auteurs (et les éléments de la théorie qui sont utilisés)

ces postulats sont en nombre variable. Nous suivrons dans ce document l'approche de Cohen-Tannoudji qui en propose six dans [13].

Postulat 1 (Fonctions d'onde) *Les états d'un ensemble de N particules ponctuelles sont décrits par une fonction à valeurs complexes $\psi(r_1, r_2, \ldots, r_N, t) = \psi(r, t)$, où $\int_{Espace} \psi^*(r,t)\psi(r,t)d\tau_1 d\tau_2 \ldots d\tau_N$ représente la probabilité de trouver chaque particule $k \in [1 \ldots N]$ dans l'élément de volume $d\tau_k$ centré sur $\overrightarrow{r_k}$, à l'instant t.*

La fonction ψ s'appelle *fonction d'onde* du système de particules. La fonction d'onde contient toute l'information possible sur l'ensemble de particules étudié : la calculer, ou en extraire des informations, sera l'objectif constant des algorithmes étudiés dans ce travail. Les fonctions d'onde doivent donc être au minimum de carré sommable, et en corollaire du postulat 1, leur norme L_2 vaut 1 (puisque cette norme représente la probabilité de trouver les particules dans tout l'espace à un instant t donné).

Dans toute la suite du document on notera $< . \mid . >$ le produit scalaire L_2 et on a donc $< \Psi \mid \Psi > \; = 1$.

Postulat 2 (Grandeur mesurable) *A chaque grandeur physique A mesurable du système de particules on peut faire correspondre un opérateur linéaire et hermitique $\mathcal{A}$, agissant dans l'espace des fonctions d'onde, et tel que la valeur moyenne $< A >$ de A, mesurée quand le système est dans l'état $\psi(r)$, a pour expression :*

$$< A >= \int_{Espace} \Psi^* \mathcal{A} \Psi d\tau.$$

La notation traditionnelle (de Dirac) pour la valeur moyenne d'une observable A est : $< \Psi \mid \mathcal{A} \mid \Psi >$, qui est donc égal selon le postulat 2 à $< \Psi \mid \mathcal{A}\Psi >$ $(=< \mathcal{A}\Psi \mid \Psi >$ puisque $\mathcal{A}$ est hermitique).

Exemple d'associations observables/opérateurs : la mécanique quantique fournit également une "table de correspondance" entre les grandeurs physiques mesurables (encore appelées *observables*) et les opérateurs mathématiques qui permettent de les calculer.

- la position q_i selon la composante i de l'espace (projection sur l'axe i du repère) est associée à l'opérateur "multiplication par la i-ième coordonnée" (par exemple : la position selon x est associée à l'opérateur linéaire $\Psi(x,y,z,t) \longrightarrow x.\Psi(x,y,z,t)$) ;
- la quantité de mouvement p_j selon la composante j de l'espace est associée à l'opérateur
$$\pi_j = \frac{\hbar}{i}\frac{\partial}{\partial x_j},$$
où $\hbar = h/2\pi$ ($h \approx 6,62618.10^{-34}$ J.s est la constante de Planck) ;

– les autres observables s'expriment comme des combinaisons des précédentes et on en déduit les opérateurs associés.

On peut donner l'exemple de l'observable "énergie cinétique" E_c d'un ensemble de particules. On sait que E_c s'exprime, pour une particule i, à partir de p la quantité de mouvement : $E_c(i) = p^2/2m_i$, où m_i est la masse de la particule. A l'aide de l'opérateur associé à p, on déduit que l'opérateur associé à E_c sera

$$\mathcal{E}_c(i) = \frac{1}{2m_i}(-\hbar^2)\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right),$$

ce qui s'écrit encore $\mathcal{E}_c(i) = -\frac{\hbar^2}{2m_i}\Delta$.

L'énergie cinétique de l'ensemble des N particules sera la somme des énergies de chacune d'entre elles, et donc l'opérateur $\mathcal{E}_c$ vaudra

$$\mathcal{E}_c = \sum_{i=1}^{N} -\frac{\hbar^2}{2m_i}\Delta_i.$$

Le i indiçant l'opérateur Laplacien Δ est là pour rappeler qu'il doit être calculé, pour chaque particule i, en exprimant les coordonnées dans un repère d'origine i.

Postulat 3 (observable et valeur propre) *La mesure d'une observable A ne peut donner comme résultat que des valeurs propres de $\mathcal{A}$, l'opérateur correspondant à A.*

Exemple d'application de ce postulat : dans le cas où l'observable recherchée est l'énergie totale d'un ensemble de N particules placées dans un potentiel $V(r)$ qui ne dépend que des coordonnées où l'on se trouve mais non du temps (potentiel Coulombien par exemple), cette énergie est la somme de E_c et de V. V ne dépendant que des positions q_i, l'opérateur qui lui est associé est $\Psi(r,t) \longrightarrow V(r)\Psi(r,t)$ et l'on peut donc écrire que l'opérateur linéaire asssocié à l'observable Energie est le Hamiltonien :

$$\mathcal{H} = \sum_{i=1}^{N} -\frac{\hbar^2}{2m_i}\Delta_i + V(r_1, r_2, \ldots, r_N). \tag{2.1}$$

Le postulat 3 exprime donc que l'énergie observable de cet ensemble de particules est forcément une valeur propre du Hamiltonien (2.1). L'étude du spectre de cet opérateur sera développée dans les parties suivantes.

Postulat 4 (Décomposition spectrale dans le cas d'un spectre discret) *Soit λ_n une valeur propre de l'observable A, de multiplicité g_n. Soit $u_n^i, i = 1 \ldots g_n$ une base orthonormée de l'espace propre associé à λ_n. Alors la mesure d'une observable A dans l'état Ψ de norme 1 vaut λ_n avec la probabilité :*

$$\mathcal{P}(\lambda_n) = \sum_{i=1}^{g_n} | < u_n^i \mid \Psi > \quad |^2.$$

Ceci est possible car les fonctions propres de $\mathcal{A}$ forment une base orthonormée complète de l'espace total (puisque $\mathcal{A}$ est hermitique).

Le postulat 3 indique que le résultat d'une mesure ne peut être qu'une valeur propre de l'opérateur associé à l'observable mesurée et le postulat 4 donne la probabilité avec laquelle on l'obtient.

Néanmoins dès que la mesure est effectuée, on ne peut plus être dans l'état Ψ de départ et on n'est plus dans le cas où l'on "peut avoir la valeur λ_n avec la probabilité $\mathcal{P}(\lambda_n)$". Le postulat suivant précise ce que l'on attend dans ce cas :

Postulat 5 (Réduction de la fonction d'onde) *Si le résultat de la mesure de l'observable A sur le système dans l'état Ψ donne le résultat λ_n, l'état du système immédiatement après la mesure est la projection orthogonale de Ψ dans l'espace propre associé à λ_n.*

En particulier si λ_n est de multiplicité 1, le système se trouve après mesure dans l'état propre normé associé à λ_n.

Ce dernier postulat est associé à l'image du "chat de Schrödinger" : un chat, c'est-à-dire un ensemble de particules, est complètement isolé dans une boîte sans interaction avec l'extérieur. Un dispositif automatique est capable de briser à tout instant une capsule de cyanure dans la boîte. Le "chat quantique" est-il mort ou vivant ? Tant qu'aucune mesure n'est effectuée (donc tant que personne ne regarde dans la boîte, par exemple), il n'est ni l'un ni l'autre : sa fonction d'onde est une combinaison linéaire des états "mort" et "vivant" selon le postulat 2, et une mesure pourrait déboucher sur l'une ou l'autre des deux issues avec la probabilité du postulat 4. Par contre le postulat 5 précise que dès que la mesure est effectuée, le chat prend bel et bien l'un des deux états. Cette "expérience de pensée" pose les problèmes de la transmission de l'information (comment le chat "sait-il" quel état prendre selon que la mesure est effectuée ou non ?) ; et également la question de savoir ce qu'est une mesure : une caméra qui filmerait le chat serait-elle "capable" de le projeter dans l'un des deux états ? Ou doit-on postuler que seule la "conscience" d'un observateur donne sa valeur à la mesure ? [1]

Il manque encore un postulat pour compléter la théorie exposée : celui qui gouverne l'évolution temporelle de la fonction d'onde d'un système de particules.

Postulat 6 *Un système de particules dans l'état $\Psi(r,t)$ évolue selon l'équation de Schrödinger dépendante du temps :*

$$i\hbar\frac{d}{dt}\Psi(r,t) = H(t)\Psi(r,t). \tag{2.2}$$

[1] Pour une réflexion sur les problèmes philosophiques que soulèvent les postulats de la mécanique quantique et les résultats scientifiques qui les renforcent ou non, on pourra consulter par exemple [51].

L'équation de Schrödinger dépendante du temps est la plus générale qui soit. Les cas que nous étudierons en chimie quantique à la section 2.4, ainsi que ceux de la section 2.3 se simplifieront par rapport à cette équation très générale. Nous commençons dans la section suivante par regarder le cas du système stationnaire.

2.2 Equation de Schrödinger stationnaire

Dans le cas géneral l'opérateur associé à l'observable étudiée dépend du temps et des trois coordonnées de l'espace. Cependant beaucoup d'applications peuvent apparaître dans des cas plus simples : la première section montre comment on peut simplifier le calcul dans le cas stationnaire ; ensuite est expliqué comment on peut passer d'un cas à une dimension d'espace au cas géneral à trois dimensions. Les parties suivantes explicitent deux cas où l'on peut résoudre entièrement l'équation de Schrödinger (atome de l'hydrogène et puits de potentiel carré).

2.2.1 Cas d'une particule soumise à un potentiel V indépendant du temps

Dans ce cas, l'équation de Schrödinger (2.2) s'écrit

$$i\hbar\frac{d}{dt}\Psi(r,t) = -\frac{\hbar^2}{2m}\Delta\Psi(r,t) + V(r)\Psi(r,t).$$

On peut rechercher des solutions sous la forme à variables séparées $\Psi(r,t) = \varphi(r)\chi(t)$. En réinjectant dans l'équation ci-dessus et en regroupant les variables on obtient que $\varphi(r)$ et $\chi(t)$ doivent chacune vérifier :

$$\begin{cases} \frac{i\hbar}{\chi(t)}\frac{d\chi(t)}{dt} = \hbar\omega, \\ -\frac{\hbar^2}{2m}\Delta\varphi(r) = \hbar\omega\varphi(r), \end{cases} \tag{2.3}$$

où ω est une constante. La première équation s'intègre trivialement et on en déduit que les fonctions

$$\Psi(r,t) = \varphi(r)e^{-i\omega t}$$

sont solutions de (2.2) si et seulement si $\varphi(r)$ est solution de l'équation de Schrödinger simplifiée suivante :

$$-\frac{\hbar^2}{2m}\Delta\varphi(r) = \hbar\omega\varphi(r). \tag{2.4}$$

Ainsi, on a séparé la partie temporelle de la partie spatiale de la fonction d'onde. Une telle fonction d'onde est appelée fonction d'onde *stationnaire*, et l'équation (2.4) est l'équation de Schrödinger stationnaire.

Cette forme spécifique des fonctions d'onde est valable dans tous les cas où le Hamiltonien ne dépend pas du temps. Ce sera en particulier le cas pour le calcul des orbitales de l'atome d'hydrogène présenté en section (2.2.3.1).

2.2.2 Cas où le potentiel V se décompose selon les coordonnées spatiales

Une autre simplification courante [34] est possible quand le potentiel s'écrit comme une combinaison linéaire des variables d'espace x, y et z. Par exemple, si $V(x,y,z,t) = V_1(x) + V_2(y) + V_3(z)$, on peut montrer ([13] p. 61) que les fonctions d'onde stationnaires $\varphi(x,y,z)$ s'écrivent sous la forme $\varphi_1(x)\varphi_2(y)\varphi_3(z)$, avec $\varphi_i(x)$ vérifiant $\left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V_i(x)\right]\varphi_i(x) = E_i\varphi_i(x)$, et que la valeur propre E associée à $\varphi_1(x)\varphi_2(y)\varphi_3(z)$ égale $E_1 + E_2 + E_3$.

On est donc ramené à la résolution de l'équation de Schrödinger stationnaire monodimensionnelle. A partir de celle-ci on peut "remonter" à l'équation tridimensionnelle. Ceci justifie par exemple le traitement du cas monodimensionnel dans la section 2.2.3.2, puisqu'on peut en déduire le cas général en trois dimensions.

2.2.3 Résolution analytique de l'équation de Schrödinger stationnaire

Le problème est donc de calculer les valeurs et fonctions propres de l'opérateur Hamiltonien pour un ensemble de N particules (de masses $m_i, i = 1 \ldots N$) :

$$\mathcal{H} = \sum_{i=1}^{N} -\frac{\hbar^2}{2m_i}\Delta_i + V(r_1, r_2, \ldots, r_N). \tag{2.5}$$

Selon les cas traités la fonction potentiel V peut varier. Dans les cas les plus simples on aboutit à une équation différentielle entièrement soluble analytiquement. Le cas de l'atome d'hydrogène est présenté dans la section suivante, puis c'est l'exemple d'une particule piégée dans un puits de potentiel que nous discuterons.

2.2.3.1 Atome d'hydrogène

Cet atome, le premier de la classification périodique, sert d'exemple car il permet à la fois de mieux comprendre les notions de la mécanique quantique et de fournir des fonctions d'onde de base qui fondent la plupart des méthodes numériques présentées dans la suite du document.

Le cas de l'atome d'hydrogène est le plus simple puisqu'il ne met en jeu qu'un unique électron de masse m_e en rotation autour d'un noyau de masse $M \gg m_e$ supposé fixe. On peut alors se ramener au cas à une seule particule de masse réduite $\mu = m_e M/(m_e + M)$.

Le potentiel auquel est soumis l'électron est limité au potentiel Coulombien qui est (on note ici $\|.\|$ la norme euclidienne d'un vecteur) :

$$V(\overrightarrow{r}) = \frac{e^2}{4\pi\epsilon_0\|r\|},$$

où ϵ_0 est une constante physique (permitivité du vide).

L'équation de Schrödinger (2.5) s'écrit donc dans ce cas :

$$-\frac{\hbar}{2\mu}\Delta\psi - \frac{e^2\psi(r,\theta,\phi)}{4\pi\epsilon_0\|r\|} = E\psi.$$

La géométrie complètement sphérique de l'hydrogène permet de passer en coordonnées sphériques pour la résoudre. On peut alors effectuer une séparation des variables pour trouver exactement le spectre de cet opérateur $\mathcal{H}$ et une base orthonormée de ses vecteurs propres Φ.

On obtient les valeurs propres (ordonnées) de $\mathcal{H}$ et ses fonctions propres, indicées par n, l et m :

$$E_n = -\frac{\mu e^4}{\hbar^2(4\pi\epsilon_0)^4}\frac{1}{2n^2}, \quad n = 1, \ldots, +\infty$$
$$\Phi_l^m(r,\theta,\phi) = R_{n,l}(r)Y_l^m(\theta,\phi), \quad l = 0, 1, \ldots, n-1\,; m = -l, -l+1, \ldots, l-1, l$$

La partie radiale des fonctions d'onde ne dépend pas de m et est du type :

$$R_{n,l}(r) = KQ_l(r)e^{-K'r}$$

où $Q_l(r)$ est un polynôme de degré $n-1$ dépendant de l.

La partie angulaire des fonctions propres, les $Y_l^m(\theta,\phi)$, sont des harmoniques sphériques :

$$Y_l^m(\theta,\phi) = \frac{1}{\sqrt{2\pi}}S_{l,m}(\theta)e^{im\phi},$$
$$S_{l,m}(\theta) = \left[\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}\right]^{\frac{1}{2}} P_l^{|m|}(\cos\theta)$$

où $P_l^{|m|}(X)$ est la *fonction de Legendre*.

On retrouve ici le phénomène de quantification de l'énergie (prédit par Planck) car le spectre de $\mathcal{H}$ est discret et caractérisé par 3 nombres entiers n, l et m : les nombres

quantiques. L'énergie peut prendre donc plusieurs valeurs discrètes (les valeurs propres E_n ci-dessus). À chacune de ces valeurs correspondent plusieurs fonctions propres (sauf pour n = 1), caractérisées par les paramètres l et m.

On appelle une telle fonction d'onde une *orbitale*.

Convention de vocabulaire : On désigne les différentes orbitales correspondant aux différentes valeurs possibles du paramètre l selon le tableau suivant :

l	0	1	2	3	4	5	6	7	...
lettre	s	p	d	f	g	h	i	k	...

(s, p, d et f signifient "sharp", "principal", "diffuse" et "fundamental" ; ensuite on utilise l'ordre alphabétique, la lettre j exceptée.)

On pourra donc parler par exemple de l'orbitale p ($l = 1$) de l'état de première excitation ($n = 2$, l'état $n = 1$ étant l'état *fondamental*, ou état de repos de l'atome) de l'hydrogène, pour désigner $\Phi_2^m(r, \theta, \phi) = R_{2,1}(r)Y_1^m(\theta, \phi)$.

Les orbitales de l'atome d'hydrogène serviront comme première approximation pour fonder les méthodes numériques décrites en 2.4.

La section suivante propose un autre cas où l'on sait résoudre exactement l'équation de Schrödinger .

2.2.3.2 Puits carré

On se place dans cette section en dimension 1. L'exemple d'une particule unique de masse m soumise à un potentiel constant par intervalles peut être entièrement résolu dans certaines cas. Ceux-ci, simples, sont également à la base de nombreux modèles en physique, par exemple pour les défauts dans un potentiel périodique comme on en trouve dans les semi-conducteurs (*cf.* section 2.3).

Si l'on se place sur un intervalle où $V(x) = V_0$, l'équation de Schrödinger (2.5) se ré-écrit simplement :

$$\left(\frac{-\hbar^2}{2m}\frac{d}{dx^2}\right)\Psi(x) = (E - V_0)\Psi(x). \tag{2.6}$$

On montre facilement que les solutions de cette équation différentielle sont :

– si $E > V_0$, en posant k le réel positif tel que $E - V_0 = \frac{\hbar^2 k^2}{2m}$,

$$\Psi(x) = Ae^{ikx} + Be^{-ikx},$$

où A, B sont des constantes complexes.

– si $E < V_0$ en posant ρ le réel positif tel que $V_0 - E = \frac{\hbar^2\rho^2}{2m}$,

$$\Psi(x) = Ae^{\rho x} + Be^{-\rho x},$$

où A, B sont des constantes complexes.

– si $E = V_0$, Ψ est une fonction affine.

On montre de plus que les solutions sont continues et à dérivée première continue.

Le cas du puits de potentiel carré infini est celui où $V(x) = -\infty < 0$ pour $x \in [0, a]$ et $V(x) = 0$ partout ailleurs. En tenant compte des conditions aux bords dans ce cas on aboutit aux solutions bien connues, en posant $k = \frac{n\pi}{a}$ et dans le cas où k est entier :

$$E_n = \frac{\pi^2\hbar^2}{2ma^2}n^2 \; ; \tag{2.7}$$

$$\Psi_n(x) = \sqrt{\frac{2}{a}}\sin\left(\frac{n\pi x}{a}\right). \tag{2.8}$$

On a donc également dans ce cas quantification des niveaux d'énergie.

Le cas du puits de potentiel carré fini est celui où $V(x) = -V_0 < 0$ pour $x \in [-a/2, a/2]$ et $V(x) = 0$ partout ailleurs. En exprimant les conditions aux bords et le fait que les solutions doivent être C^1 on obtient que les solutions $-V_0 < E < 0$ de l'équation de Schrödinger sont solutions d'une équation implicite. En posant :

$$\rho = \sqrt{-\frac{2mE}{\hbar^2}}, \tag{2.9}$$

$$k = \sqrt{\frac{2m(E+V_0)}{\hbar^2}}, \tag{2.10}$$

il vient ([13]) que E doit vérifier :

– si $(\rho - ik)/(\rho + ik) = -e^{ika}$,

$$\begin{cases} |\cos\left(\frac{ka}{2}\right)| = \frac{k}{k_0}, \text{ où } k_0 = \sqrt{k^2+\rho^2}, \\ \tan\left(\frac{ka}{2}\right) > 0. \end{cases}$$

– si $(\rho - ik)/(\rho + ik) = e^{ika}$,

$$\begin{cases} |\sin\left(\frac{ka}{2}\right)| = \frac{k}{k_0}, \text{ où } k_0 = \sqrt{k^2+\rho^2}, \\ \tan\left(\frac{ka}{2}\right) < 0. \end{cases}$$

Dans l'un et l'autre cas les valeurs possibles de E sont les intersections entre une droite de cœfficient directeur $1/k_0$ et des arcs de sinusoïde : on a à nouveau quantification de l'énergie.

Ce dernier exemple montre la limite, très vite atteinte, des cas analytiques, puisque déjà on ne peut plus trouver de formule explicite pour calculer les valeurs de E. Il va de soi que dès que l'on veut travailler sur un puits de potentiel un tant soit peu plus compliqué, ou en plusieurs dimensions sans symétrie, le recours au numérique s'impose.

2.3 Le problème de la boîte quantique (BQ)

Dans le cas tridimensionnel, le puits de potentiel décrit ci-dessus devient une “boîte” où la particule est confinée. Le problème de la boîte quantique, noté BQ dans la suite, consiste donc en la détermination des valeurs propres les plus petites (énergie de l’état fondamental et des premiers états excités) ainsi que des fonctions d’onde associées du Hamiltonien $\mathcal{H}$ associé au puits de potentiel.

Cette section décrit rapidement un exemple de mécanique du solide où l’on aboutit directement sur ce problème (2.3.1), puis une première méthode numérique pour le résoudre (méthode variationnelle) utilisée de façon standard par les physiciens du solide. Nous proposons un autre type d’algorithmes dans la section 2.3.3.

2.3.1 Méthode kp et masse effective

2.3.1.1 Méthode kp

Dans le cadre de la physique des semi-conducteurs, le Hamiltonien $\mathcal{H} = p^2/(2m_0) + V$ est tel que la fonction potentiel $V = V_\sim(x,y,z) + V_\sqcup(x,y,z)$, où $V_\sim$ est périodique afin de refléter la structure régulière du cristal formant le matériau et $V_\sqcup$ est un potentiel confinant.

Dans la suite de cette section on se placera dans le cadre de la dimension 1 pour simplifier la présentation. Les résultats de la théorie se généralisent sans mal à trois dimensions. Dans ces conditions, $V_\sqcup$ s’écrit typiquement $V_\sqcup = V_\sqcup(x) = 0$ si $x \notin [-x_0, x_0]$ et $-V_0$ sinon, avec x_0 et V_0 des constantes positives. On note $\mathcal{H}_0$ le Hamiltonien $\frac{p^2}{2m_0} + V_\sim(x)$ ($p = -i\hbar\frac{d}{dx}$).

$V_\sim$ étant périodique, le théorème de Bloch [13] implique alors que les fonctions propres de $\mathcal{H}_0$ sont sous la forme

$$v_k(x) = e^{ikx}\psi(x),$$

où k est un réel quelconque et $\psi(x)$ une fonction ayant la même périodicité que le cristal. Avec la fonction d’onde sous cette forme, on aboutit en ré-écrivant l’équation de Schrödinger à une séparation des variables et on se ramène à la résolution de l’équation de Schrödinger au Hamiltonien "perturbé" :

$$\mathcal{H}_k \stackrel{\text{def}}{=} \mathcal{H}_0 + k_0^2 + \frac{\hbar}{m_0}kp, \qquad k_0^2 = \frac{\hbar^2}{2m_0}k^2.$$

L’opérateur $\mathcal{H}_k$ peut donc être vu comme une perturbation de l’Hamiltonien $\mathcal{H}_0$ par kp. Pour calculer les valeurs et vecteurs propres de $\mathcal{H}_k$, on part donc de ceux de $\mathcal{H}_0$ et on observe l’influence de la perturbation sur eux. En notant $E_i, i \geq 0$ les valeurs propres du

Hamiltonien $\mathcal{H}_0$ non perturbé, (Ψ_i) une base orthonormale de vecteurs propres associés et

$$V_{ij} = \frac{\hbar}{m_0} < \psi_i \mid p \mid \psi_j > ,$$

on obtient la matrice de $\mathcal{H}_\sim$ dans la base des Ψ_i :

$$H(k) = \begin{bmatrix} E_1 + k_0^2 & & (kP_{ij}) \\ & \ddots & \\ (k\bar{P}_{ij}) & & E_n + k_0^2 \end{bmatrix}. \tag{2.11}$$

En utilisant les formules de Rayleigh-Schrödinger on peut calculer une approximation au deuxième ordre des solutions de ce Hamiltonien perturbé $\mathcal{H}_k$

$$E_i(k) = E_i + \sum_{j \neq i} \frac{H_{ij}^2}{E_i - E_j} k^2 + o(k^2). \tag{2.12}$$

(Dans [32] nous proposons une autre approche mathématique pour le traitement des perturbations dans le Hamiltonien, basée sur le polygône de Newton).

Sous cette forme, le développement de l'énergie au deuxième ordre en k permet d'assimiler la particule dans un potentiel périodique à une particule libre (potentiel constant) puisque dans ce cas trivial les solutions de l'équation de Schrödinger sont $E' = -\hbar^2 k^2 / 2m_0$. Le développement (2.12) calculé, on peut donc calculer une masse effective m_0^* qui caractérise la physique de la particule dans le potentiel périodique.

2.3.1.2 Boîte quantique

Pour prendre en compte le potentiel confinant $V_\sqcup$ une technique classique ([20]) est alors de résoudre à nouveau l'équation de Schrödinger pour la particule de masse effective m_0^* plongée dans le potentiel $V_\sqcup$. La perturbation dûe à la périodicité du cristal est prise en compte par la masse effective de la particule ; le confinement est calculé par la résolution numérique du Hamiltonien avec un potentiel "en forme de boîte".

Les techniques numériques de résolution peuvent être diverses. L'une des plus employées est de type variationnel à cause des propriétés mathématiques du Hamiltonien. La section suivante introduit celles-ci.

2.3.2 Méthode variationnelle

La théorie de la mécanique quantique prévoit que l'énergie est la valeur minimale du quotient de Rayleigh

$$R(\psi) = \frac{< \psi \mid H \mid \psi >}{< \psi \mid \psi >},$$

où ψ parcourt tout l'espace des fonctions d'onde. Cela revient à dire que le Hamiltonien $\mathcal{H}$ vérifie le théorème du min-max. Nous l'exposons ici en dimension finie (*cf.* [58, 53]). Soit H la matrice hermitienne de taille $n \times n$ qui représente $\mathcal{H}$ dans l'espace de dimension n et $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ ses n valeurs propres réelles.

Théorème 1 (Formule du min-max) *La k-ième valeur propre λ_k de l'opérateur H est donnée par la formule*

$$\lambda_k = \min_{V, dim(V)=k} \left\{ \max_{u \in V, u \neq 0} \frac{(Hu, u)}{(u, u)} \right\}, \tag{2.13}$$

$$= \max_{V, dim(V)=n-k+1} \left\{ \min_{u \in V, u \neq 0} \frac{(Hu, u)}{(u, u)} \right\}. \tag{2.14}$$

Dans le cas particulier de $k = 1$, l'équation (2.14) implique bien que la plus petite valeur propre λ_1 est le $\min$, sur les vecteurs u de tout l'espace, du quotien de Rayleigh.

Au delà de ce cas particulier de la plus petite valeur propre, le théorème du min-max permet de caractériser toute valeur propre comme un optimum dans un ensemble particulier de sous-espaces de l'espace total.

Cela permet de déduire immédiatement une classe de méthodes numériques très employées par les physiciens : les méthodes dites variationnelles.

Le principe est de se placer en dimension finie en choisissant une base de fonctions d'onde particulières pour projeter le Hamiltonien dans un espace de dimension finie. La physique (ou la chimie) du problème intervient dans le choix de la base car les fonctions d'ondes seront choisies selon les connaissances *a priori* pour être le plus proche possible de la solution cherchée. L'idée est ensuite de définir les fonctions d'onde de la base en fonction d'un ou de plusieurs paramètres caractéristiques du problème. Comme on cherche un minimum, il suffira ensuite de faire varier les paramètres afin de minimiser le quotient de Rayleigh.

C'est ainsi que [20] traite le cas des puits quantiques à géométrie irrégulière (non carrée). La base choisie est un ensemble de fonctions d'onde solutions du cas à puits carré. Elles sont paramétrées par la largeur a du puits. Le Hamiltonien est exprimé dans cette base puis on peut en calculer numériquement la plus petite valeur propre, ce qui sert de fonction objectif à un algorithme de minimisation.

Cet algorithme a plusieurs avantages : le premier est de lier la recherche de la valeur propre à un critère de minimisation d'énergie qui est purement physique. Le second est de permettre au physicien d'exprimer sa connaissance du problème à travers le choix de la base. Cela permet de plus *a posteriori* de "donner du sens physique" aux énergies et fonctions d'onde trouvées par l'optimisation.

L'inconvénient est la limitation justement dûe au choix de la base : que faire quand on n'a aucune idée *a priori* ? Pire, ne risque-t-on pas de chercher la solution dans un espace

qui n'est pas représentatif en fait du phénomène étudié ? Un autre élément limitatif est la taille des espaces que l'on peut ainsi traiter. La matrice H construite dans ces espaces est souvent dense, ce qui limite la capacité de traitement.

2.3.3 Méthodes itératives

Une autre idée est donc de discrétiser l'espace de façon indépendante des solutions recherchées, par exemple par différences finies (ou par éléments finis). La matrice H, de taille $n \times n$, du Hamiltonien aura le même profil que celle d'un Laplacien (tridiagonale), avec des cœfficients diagonaux en plus dûs à la fonction potentiel. Elle sera donc très creuse, et hermitienne. On peut y appliquer une autre classe de méthodes numériques : les méthodes itératives.

La première des méthodes itérative est celle de la puissance itérée.

Etant donné un vecteur initial u_0, la méthode de la puissance itérée consiste en le calcul successif des $H^k u_0, k = 1, 2, 3, \ldots$. Cette suite de vecteurs converge vers le vecteur propre associé à la plus grande valeur propre en module.

Une généralisation directe est l'itération des sous-espaces, qui part d'une matrice X de taille $n \times m$ (constituée de m vecteurs de départ, linéairement indépendants, pour les itérations) et calcule les itérations $H^k X$. Néanmoins la matrice ainsi calculée à l'étape k voit son rang diminuer car toutes ses colonnes convergent vers le vecteur propre associé à la valeur propre de plus grand module. Pour éviter cela, il faut ré-orthogonaliser régulièrement la matrice : toutes les $l > 1$ itérations, on effectue la factorisation $QR = H^l X$. On pose $X = Q$, et on recommence l itérations. Le nombre l d'itérations à effectuer avant la ré-orthogonalisation QR dépend de la vitesse de convergence (donc de la matrice H). Cette méthode dûe à Bauer permet d'obtenir m vecteurs (et valeurs) propres grâce à l'emploi d'un sous-espace vectoriel de taille m, celui engendré par les m colonnes de X, noté $\overrightarrow{X}$ dans la suite.

Néanmoins on converge encore vers les valeurs propres les plus grandes en module. Afin de sélectionner celles que l'on désire et également d'accélérer la convergence vers la meilleure approximation possible dans l'espace de taille m utilisé, on emploie des techniques de projection orthogonale ([58]) sur le sous-espace. Étant donnée une base orthonormale $\{Z_1, Z_2, \ldots Z_m\}$ du sous-espace $\overrightarrow{X}$, la matrice $Z^* H Z$, de taille $m \times m$, représente la projection de l'opérateur associé à H, restreint au sous-espace $\overrightarrow{X}$. On peut calculer le spectre entier de cette matrice $m \times m$ par des méthodes directes (méthode QR par exemple). On peut extraire de ce spectre les vecteurs propres qui nous intéressent (par exemple ceux associés aux plus petites valeurs propres), les exprimer dans l'espace de départ grâce à la matrice de passage Z et recommencer les itérations à partie de ces vecteurs sélectionnés.

On dispose donc de méthodes itératives efficaces pour calculer des valeurs propres

localisées dans le spectre de H. La section 3.2 illustrera plus en détail l'utilisation de l'algorithme de Arnoldi avec redémarrage implicite. Ce qui est important dans cette classe de méthodes est qu'elles ne font intervenir H qu'à travers sa multiplication par un vecteur u_0 (ou par plusieurs dans le cas des itérations de sous-espaces). Ce produit matrice-vecteur, avec une matrice très creuse, peut être programmé de manière très performante sur machine vectorielle ou parallèle. De plus elles sont économes en mémoire. Cette façon de traiter l'équation de Schrödinger est donc une alternative sérieuse à la méthode variationnelle.

2.4 Chimie quantique et méthodes d'approximation

Malheureusement, la forme de $\mathcal{H}$ dépend des potentiels auxquels est soumise la particule considérée et la résolution analytique de l'équation de Schrödinger est dans la grande majorité des cas impossible. D'une part, les problèmes mathématiques à résoudre sont du type "problème à n corps", qui est l'exemple même, dans son équivalent en mécanique classique, du phénomène chaotique ; d'autre part, l'espace de Hilbert où l'on travaille est *a priori* de taille infinie ; pour calculer, on est donc amené à tronquer cet espace pour se ramener en dimension finie, suffisamment grande pour assurer la précision des calculs.

L'expression du Hamiltonien fait apparaître des dérivées secondes et (2.2) se ramène à une équation différentielle aux variables en général non séparables que l'on peut résoudre de manière approchée. Deux voies sont possibles pour cela : les techniques *semi-empiriques* modifient le Hamiltonien pour le simplifier en introduisant des paramètres obtenus expérimentalement ; elles sont rapides à mettre en œuvre, mais ne fonctionnent que pour des phénomènes déjà connus. La chimie théorique "*ab initio*" permet le calcul des propriétés chimiques des molécules (structure, énergie de liaison, moment dipolaire, fréquences de vibration, potentiels d'interaction inter-moléculaires…) en faisant appel aux fondements théoriques de la mécanique quantique. L'absence de tout paramètre ajustable ou empirique rend cette approche *prédictive*, d'où son intérêt dans toute situation où des mesures expérimentales sont difficiles.

Une première approximation en chimie quantique *ab-initio* consiste à moyenner l'effet de $n-1$ particules sur la n-ième : c'est l'objet de la méthode SCF (section 2.4.1). Pour aller au delà, une méthode standard est celle des perturbations. La méthode MP2 a fait l'objet d'une étude particulière dans le travail de parallélisation et sera exposée ensuite.

2.4.1 Champ moyen — méthode SCF

Les orbitales de l'atome d'hydrogène sont bien connues (*cf.* 2.2.3.1). Mais dès l'atome d'hélium, le deuxième de la classification périodique, apparaissent dans l'expression de

l'hamiltonien des termes de répulsion électronique (*cf.* par exemple des termes Coulombiens en $1/\|r_i - r_j\|$ entre deux particules i et j) qui empêchent toute résolution analytique des équations différentielles.

Comment faire donc dans le cas des atomes à plusieurs $(n > 1)$ électrons pour trouver une "bonne" fonction d'onde ? Hartree a proposé en 1928 une méthode dite SCF (Self Consistent Field) pour trouver une approximation acceptable de la fonction d'onde.

Comme le problème de calcul vient des termes d'interactions électroniques, on peut partir en première approximation d'une fonction d'onde $\psi^{(0)}$ calculée en négligeant complètement ces termes. Mathématiquement (*cf.* la séparation des variables en 2.2.2), cela revient à écrire la fonction d'onde comme un produit d'orbitales mono-électroniques, du type hydrogènoïde donc, comme celles décrites précédemment en (2.2.3.1).

$$\psi^{(0)} = \Phi_1(r_1, \theta_1, \phi_1)\Phi_2(r_2, \theta_2, \phi_2) \ldots \Phi_n(r_n, \theta_n, \phi_n) \tag{2.15}$$

Dans l'ensemble des orbitales $\Phi_i(r_i, \theta_i, \phi_i)$, certaines seront dites "occupées", c'est-à-dire que l'on aura effectivement des électrons dans cette orbitale (au maximum deux par orbitale) ; et les autres seront dites "virtuelles" : elles permettent de modéliser l'espace complet des états excités possibles.

Le calcul des $\Phi_i\,, i = 1, \ldots, n$ est effectué de manière itérative : on considère l'électron 1 et on assimile les $(n-1)$ autres à un "champ" moyen pour calculer un terme de répulsion entre l'électron 1 et l'ensemble des autres, corrigeant ainsi l'oubli total de la répulsion inter-électronique dans $\psi^{(0)}$. On résout alors à nouveau l'équation de Schrödinger avec un potentiel Coulombien issu de ce champ moyen pour obtenir une nouvelle orbitale monoélectronique $\Phi_1^{(1)}$ que l'on ré-injecte dans l'expression de $\psi^{(0)}$ (*cf.* (2.15)) à la place de Φ_1. Puis on itère en refaisant la même chose avec l'électron 2, ce qui donne ψ_2, etc... En répétant l'opération, on obtient une expression moyenne de la fonction d'onde.

La méthode SCF — Self-Consistent Field — (le champ électronique trouvé est dit "self-consistant" car après p itérations il ne varie plus) a des limites, l'une d'entre-elles étant qu'elle ne tient pas compte d'une exigence physique de base : il existe un quatrième nombre quantique, le *spin*, qui interdit une telle combinaison d'orbitales (un produit) pour obtenir une fonction d'onde car elle ne satisfait pas à des règles de symétrie (principe de Pauli). On est obligé donc de passer à un déterminant de spin-orbitales (orbitales couplées avec un terme de spin prenant deux valeurs possibles), le déterminant de Slater, au lieu d'un produit d'orbitales. C'est Fock qui a proposé cette solution en 1930, nommée méthode de Hartree-Fock.

L'erreur sur les résultats obtenus pour l'énergie totale par les méthodes SCF est typiquement de l'ordre de 0.5 % à 1 %. Dans beaucoup de domaines une telle précision est suffisante. Mais pour les applications en chimie, cette précision est insuffisante. En effet, l'ordre de grandeur d'une énergie est la centaine d'électrons-Volts : l'erreur est donc de la taille d'un eV, ce qui est l'ordre de grandeur d'une énergie de liaison chimique (*cf.* [42] page 290 pour plus de détails sur ces ordres de grandeurs). On retrouve ici le fait que l'on

néglige dans ce calcul les interactions fines entre électrons pour se contenter d'un champ moyen.

On est donc amené à chercher à prendre en compte non plus un champ moyen pour les répulsions électroniques, mais des interactions plus fines qui vont fournir une correction au deuxième ordre pour l'énergie. C'est tout le but de la méthode des perturbations.

2.4.2 Méthodes des perturbations

Pour avoir plus de précision que SCF, un certain nombre de théories ont été avancées. Celle que l'on utilisera ici est celle dite *méthode des perturbations*. Le principe est à la fois simple mathématiquement parlant et naturel en physique : il consiste à dire que ce que l'on connaît est le premier terme d'un développement limité dont les termes suivants apparaissent comme des perturbations, afin de corriger les imprécisions.

Notation "bracket" :
On utilisera dans la suite le formalisme dit de "bras" et de "kets" classique en mécanique quantique ; on écrit une fonction d'onde ψ sous la forme d'un ket : $\mid \psi >$; le "bra" noté $< \psi \mid$ est l'opérateur adjoint de ψ. Si l'on se donne une matrice A et deux fonctions d'onde $\mid \varphi >$ et $\mid \psi >$, on note :

$$< \varphi \mid A \mid \psi > = \iiint \varphi^* A\psi \; dx\, dy\, dz$$

On retrouve ainsi par le pur jeu syntaxique sur les notations $< \psi \mid$ et $\mid \psi >$ les opérations dues au produit hermitien classique dans L_2 et l'on peut manipuler simplement les expressions de manière purement formelle.

La méthode des perturbations pose donc, compte tenu de ce que l'on vient de dire :

$$\mathcal{H} = H^0 + \lambda W \tag{2.16}$$

H^0 est un hamiltonien de valeurs propres $E_n^{(0)}$ calculables, avec les fonctions propres associées $\mid \psi_n^0 >$ et :

$$E_n = E_n^{(0)} + \lambda E_n^{(1)} + \lambda^2 E_n^{(2)} + \dots \tag{2.17}$$

$$\mid \psi_n > = \mid \psi_n^0 > + \lambda \mid \psi_n^1 > + \lambda^2 \mid \psi_n^2 > + \dots \tag{2.18}$$

En développant alors en série l'équation

$$\mathcal{H} \cdot \mid \psi_n > = E_n \cdot \mid \psi_n >$$

et en utilisant de plus une condition d'orthonormalisation ($< \psi_n \mid \psi_n > \; = 1$), il vient successivement (*cf.* [13] pour plus de détails) :

– à l'ordre 0 :

$$H^0 \mid \psi_n^0 > = E_n^0 \mid \psi_n^0 > \tag{2.19}$$

– à l'ordre 1 :

$$E_n^{(1)} =< \psi_n^0 \mid W \mid \psi_n^0 > \tag{2.20}$$

(l'énergie, au premier ordre, est donc la valeur moyenne de la perturbation W.)
Et :

$$\mid \psi_n^{(1)} > = \sum_{m \neq n} \overset{\text{multiplicite de } \mid \psi_m^0 >}{\sum_{q=1}} \frac{< \psi_{m,q}^0 \mid W \mid \psi_n^0 >}{E_m^0 - E_n^0} \mid \psi_{m,q}^0 > \tag{2.21}$$

– à l'ordre 2 :

$$E_n^{(2)} = \sum_{m \neq n} \overset{\text{multiplicite de } \mid \psi_m^0 >}{\sum_{q=1}} \frac{\mid < \psi_{m,q}^0 \mid W \mid \psi_n^0 > \mid^2}{E_m^0 - E_n^0} \tag{2.22}$$

On peut poursuivre le développement à des ordres supérieurs en espérant améliorer la précision. Ceci suppose cependant une convergence de la série, ce qu'aucune propriété mathématique ne permet en général d'assurer. L'expérience montre que la formule MP2 a pratiquement de bonnes propriétés et améliore significativement SCF. Aller au-delà coûte cher en mémoire et en temps de calcul avec une correction qui n'est pas garantie. C'est pourquoi on s'arrètera ici à l'ordre deux.

2.4.3 Changement de base pour les orbitales

On a donc besoin, pour effectuer le calcul par la méthode des perturbations, des fonctions d'ondes exprimées dans une base orthonormale (*cf.* (2.19)).

Or pour traduire les propriétés chimiques des atomes de la molécule étudiée, on part de la base des *orbitales atomiques* (OA), que l'on sait bien exprimer à partir d'orbitales du type hydrogène (orbitales de Slater). Mais en pratique la base d'OA n'est pas orthonormale (en particulier les OA concernant deux atomes distincts ne sont pas orthogonales).

D'autre part la formule de calcul de l'énergie au second ordre (2.22) fait intervenir :

$$< \psi_{m,q}^0 \mid W \mid \psi_n^0 > \tag{2.23}$$

pour tous les $\psi_{m,q}^0$ possibles (qui sont les fonctions d'onde Hartree-Fock de la molécule considérée). Or on montre (*cf.* [57]) qu'avec les fonctions d'ondes Hartree-Fock exprimées dans la base des *Orbitales Moléculaires* (OM), qui est orthogonale par construction, toutes ces intégrales sont nulles sauf celles mettant en jeu un $\psi_{m,q}^0$ égal à $\tilde{\psi}_n^0$, où $\tilde{\psi}_n^0$ est ψ_n^0 dont deux électrons sont passés d'orbitales occupées à des orbitales virtuelles. On peut donc réécrire la sommation de (2.22) en une sommation sur les orbitales virtuelles et occupées, double à chaque fois puisque ne restent que les intégrales portant sur des doublets

d'électrons : on n'a donc pas besoin de N^4 OM mais seulement de O^2V^2 pour la méthode des perturbations. On utilise donc systématiquement la base des orbitales moléculaires pour la méthode des perturbations. Avant de calculer l'énergie MP2, il faut donc effectuer des changements de bases pour passer des intégrales sur les fonctions d'onde exprimées dans la base des OA aux intégrales exprimées dans la base des OM.

On note traditionnellement $(\mu\nu|\lambda\sigma) \stackrel{\text{def}}{=} < \varphi_\mu\varphi_\nu \mid W \mid \varphi_\lambda\varphi_\sigma >$ les intégrales exprimées dans la base des OA, et $(ia|jb)$ les intégrales après changement de base. Avec ces notations et la remarque ci-dessus sur l'annulation des cœfficients, la sommation s'écrit dans la base des OM simplement :

$$E^{(2)} = \sum_{ijab} \frac{(ia|jb)^2 + \frac{1}{2}((ia|jb) - (ib|ja))^2}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b}. \tag{2.24}$$

Pour obtenir les O^2V^2 orbitales moléculaires, orthonormales, dont on a besoin, on utilise le calcul SCF qui permet d'obtenir justement les cœfficients $C_{\mu,i}$ des OM dans la base des OA, qui ont ces propriétés. En effet, cette orthogonalité est nécessaire pour la méthode SCF, puisqu'elle permet de simplifier les $n!$ cœfficients du déterminant de Slater (qui tient compte de l'antisymétrie de la fonction d'onde, comme expliqué en 2.4.1).

Remarque : comme expliqué ci-dessus, pour le calcul MP2 on a donc besoin de transformer des OA en OM uniquement au nombre de O^2V^2, et jamais d'effectuer des transformations sur toutes (les $N = O + V$) orbitales. Par contre, pour les méthodes allant au-delà de MP2, on ne peut faire l'économie d'une transformation globale. En ce sens, il peut être intéressant d'étudier dans toute sa généralité la transformation des N^4 orbitales même si on doit garder à l'esprit que dans ce cas se pose fatalement le problème de leur stockage en mémoire.

2.4.4 Méthode des perturbations de Møller-Plesset (MP2)

En 1932, Møller et Plesset ont proposé un traitement basé sur la méthode des perturbations, avec pour fonction d'onde non-perturbée (de degré 0) celle résultant d'un calcul SCF. L'hamiltonien non-perturbé est donc celui obtenu à l'issue des itérations de SCF (avec les remplacements successifs des potentiels de répulsion électronique).

La perturbation va donc être ici la différence entre la répulsion électronique exacte et la moyenne utilisée par SCF.

Avec ce choix de perturbation, l'énergie au premier ordre est égale à l'énergie Hartree-Fock. Au deuxième ordre, il vient la formule (2.22) :

$$E^{(2)} = \sum_{ijab} \frac{(ia|jb)^2 + \frac{1}{2}((ia|jb) - (ib|ja))^2}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{2.25}$$

i et j sont des indices d'*orbitales occupées* au nombre de O ; a et b sont des indices d'*orbitales virtuelles*, au nombre de V ; $O + V = N$ est le nombre d'orbitales dans la base considérée et c'est le paramètre qui mesure la complexité algorithmique du calcul.

On appelle MP2 (Møller-Plesset d'ordre 2) le calcul de l'énergie par cette méthode.

2.4.5 Intérêt pratique de MP2

MP2 est à la fois une méthode de chimie quantique simple à mettre en oeuvre et le premier moyen pour tenir compte de la corrélation électronique. Pour donner une idée de l'importance de celle-ci dans les calculs *ab-initio*, la figure 2.1 présente les résultats comparés d'un calcul SCF, d'un calcul MP2 (effectués par P. Valiron [46]) avec l'énergie expérimentalement mesurée par Aziz et Slaman [48]. Le système traité est He_2 et le graphe représente l'énergie d'interaction (μHa) entre les deux atomes en fonction de la distance ($\mathring{A}$) qui les sépare. La taille de la base choisie est telle que l'on peut estimer que son choix n'influe pas sur la méthode numérique.

Ce système, qui ne possède pas de liaison, est représentatif des interactions de Van der Waals, qui jouent un grand rôle dans la modélisation des gaz et des liquides.

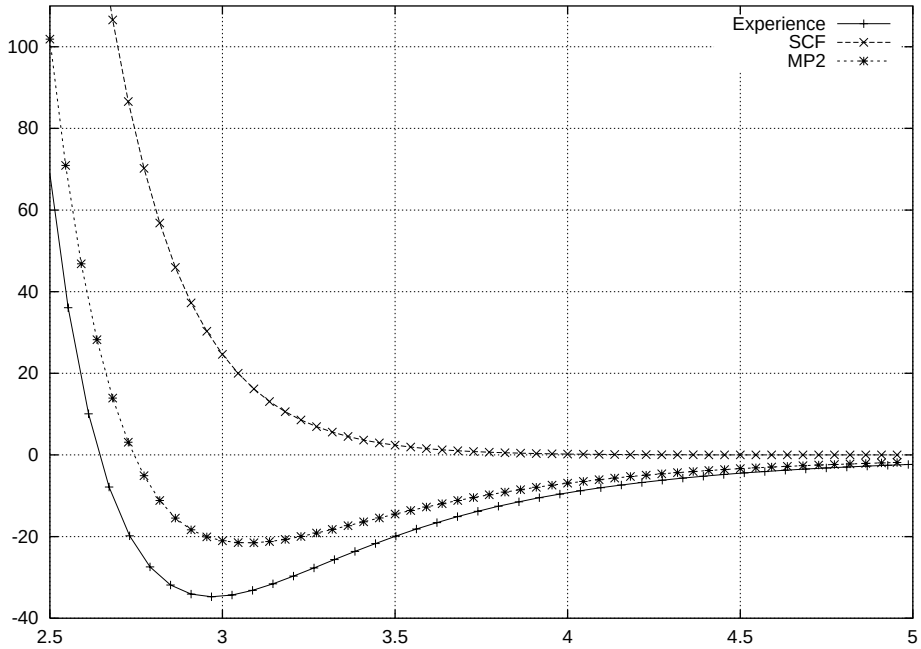


Figure 2.1 *Énergie d'interaction en fonction de la distance pour une molécule He_2, calculée selon SCF, MP2, et mesurée expérimentalement.*

On observe immédiatement la différence quantitative entre les valeurs calculées par les deux méthodes *ab-initio* d'une part ; et également entre la meilleure des deux (MP2)

et la réalité expérimentale. Au-delà de l'écart quantitatif constaté même avec MP2, on observe surtout que SCF ne rend pas même compte qualitativement d'un phénomène chimiquement crucial : le minimum observé autour de $3\mathring{A}$ signifie l'existence d'une interaction inter-atomique, or la courbe obtenue avec SCF ne présente même pas ce minimum. D'autres méthodes fréquemment utilisées depuis peu, telles Density Functional Theory, demeurent aujourd'hui également inadaptées pour le traitement précis des interactions de Van der Waals. Des méthodes de corrélation électronique telles que Coupled-Cluster, qui vont au-delà de MP2, permettent de retrouver pratiquement les valeurs expérimentales.

2.5 Conclusion

Ce chapitre a introduit les principaux concepts de la mécanique quantique et ses applications en chimie. Deux problèmes types ont été décrits plus particulièrement : le problème du Hamiltonien dans une boîte quantique, motivé par des calculs sur les semiconducteurs ; et le problème MP2, nécessaire en chimie quantique.

3

Calcul haute-performance et mécanique quantique : étude de cas

Le chapitre précédent a introduit un certain nombre de besoins en calcul intensif pour la mécanique quantique, au moins pour deux problèmes types : la boîte quantique BQ et la méthode des perturbations (MP2). Les algorithmes nécessaires sont souvent exigeants en temps d'exécution et/ou en espace mémoire. Par exemple, l'algorithme de Pople pour résoudre MP2 sur un problème de taille $N = O + V$ nécessite un espace mémoire $\mathcal{M}$ $\mathcal{O}(OVN)$ *mots* et un nombre d'opérations flottantes $\mathcal{T}$ en $\mathcal{O}(ON^4 + kON^4)$ (le premier terme est le coût de l'algorithme MP2 lui même ; le second est le coût de calcul des intégrales en entrée. Le cœfficient k représente le coût de calcul d'une intégrale, réputé élevé). Typiquement N vaut quelques centaines et O quelques dizaines. Par exemple pour $N = 500, O = 50$, on aboutit à :

$$\mathcal{M} = 8 \times 50 \times 450 \times 500 = 85 Mmots$$
$$\mathcal{T} = (1+k)50 \times 450^4 = (1+k)3125000\,\mathrm{Mflops} = (1+k)3.125 Tflops.$$

Sur un processeur de type Pentium III à 733 MHz, supposé calculer un flop par cycle donc 733 MFlops en une seconde, cela donne un temps d'exécution en $(1+k)\times$1h10. Réduire le temps d'exécution est possible à condition de ré-ordonner les calculs (*cf.* 6.4) mais cela fait passer à un espace mémoire quartique : on a alors $\mathcal{T} = ON^4 + kN^4$ mais $\mathcal{M} = (1/2)O^2VN$, ce qui fait dans notre exemple 268 Mmots.

On voit sur cet exemple simple que l'on atteint très vite les limites de praticabilité sur un processeur usuel. Le recours aux machines parallèles apparaît comme le prolongement logique pour les utilisateurs de ces algorithmes. En effet des benchmarks standards comme Linpack montrent la puissance que l'on peut obtenir avec des architectures parallèles, avec un effort de programmation non négligeable parfois. La section 3.1 présente

les résultats obtenus sur une grappe de 225 PC, en vue d'intégrer le classement top-500 des 500 machines les plus performantes du monde.

La programmation des algorithmes pour la mécanique et chimie quantique, et leur exécution sur machine séquentielle ou parallèle, nécessite également un contrôle fin si l'on veut tirer profit de ces architectures. Beaucoup de solutions ont déjà été proposées par un certain nombre d'environnements de programmation. Ce chapitre en présente quelques-uns : la librairie numérique, éventuellement parallèle, Arpack permet d'utiliser des méthodes itératives efficacement (section 3.2) en se basant sur MPI. Le logiciel Gaussian, écrit au-dessus de Linda (section 3.4), fournit des outils pour la parallélisation de certains algorithmes de chimie quantique. Des programmes parallèles, utilisant MPI, ont été proposés également pour la chimie quantique, notamment pour SCF et MP2. Enfin le standard Open-MP permet également de paralléliser certains algorithmes, au moins sur multi-processeurs (section 3.5).

3.1 Performances sur grappe du benchmark Linpack avec MPI

Les applications numériques alliant une librairie "bas-niveau" comme MPI à des opérations d'algèbre linéaire dense hautement vectorisables permettent aujourd'hui d'obtenir d'excellentes performances sur machines parallèles à noeuds identiques [17]. C'est par exemple le cas du calcul de la factorisation LU d'une matrice dense, parallélisé grâce à MPI, tel que le benchmark HPL (High Parallel Linpack) le propose[1]. Linpack est un benchmark standard pour les processeurs vectoriels qui implémente la résolution d'un système linéaire, *via* une factorisation LU (pivot de Gauss partiel). Les performances, en terme de puissance de calcul (flops/s), de la version parallèle, servent de test d'entrée au top-500, le classement mondial des supercalculateurs (http ://www.top500.org). La factorisation LU est en effet considérée comme un exemple représentatif des calculs effectués par toute une classe de codes de simulation numérique.

MPI permet d'obtenir d'excellentes performances sur grappe de PC pour ce type d'applications. L'article [56] détaille le travail d'optimisation du programme linpack réalisé sur la grappe de 225 PC HP e-Vectra du laboratoire ID-IMAG[2]. L'annexe A détaille de façon plus technique toutes les étapes et les résultats de ce travail. La figure 3.1 résume les performances obtenues en parallèle sur I-Cluster.

Les noeuds de la grappe sont des Pentium III (HP eVectra) cadencés à 733 MHz, disposant de 256 Moctets de mémoire chacun. La meilleure performance séquentielle mesurée sur cette machine est de 680 MFlop/s (sur un produit matriciel, avec des Blas optimisées). La performance séquentielle du programme Linpack parallèle (exécuté sur

[1] www.netlib.org/hpl

[2] La grappe I-Cluster est financée par un projet commun INRIA/ID-IMAG/H.P. Grenoble.

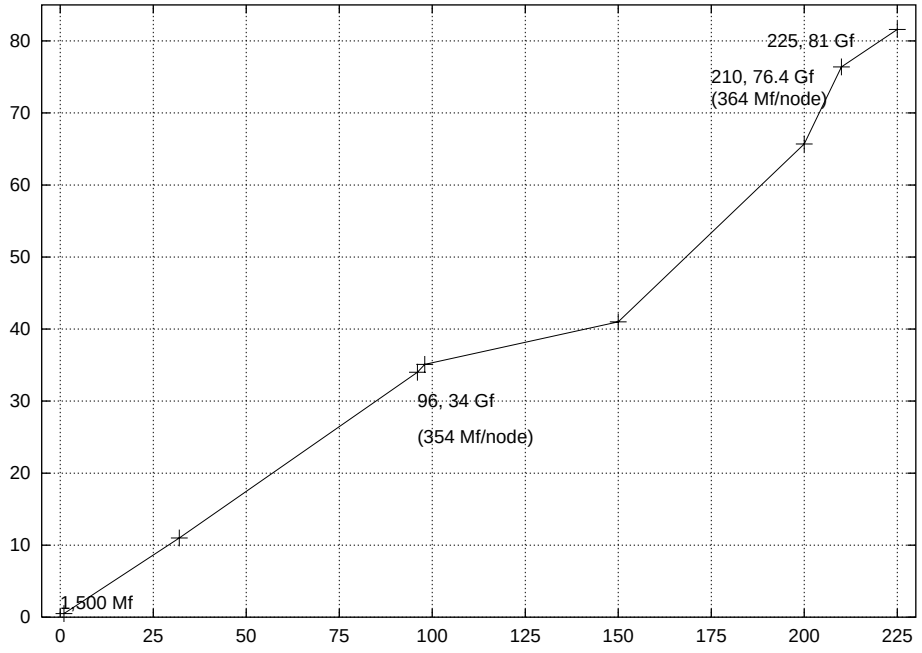


Figure 3.1 *Performances du test Linpack (GFlop/s) vs. nombre de noeuds sur la grappe I-Cluster*

un nœud) est de 500 MFlop/s. Lors des meilleures performances mesurées sur 225 nœuds qui sont de l'ordre de 81.6 GFlop/s, on obtient 368 MFlop/s nœud. On est donc en deçà des 500 MFlop/s séquentiels, mais on reste à une efficacité de $368/500 = 73.6\%$, cela sur une machine massivement parallèle et dont le réseau de communication (Fast-ethernet) n'est pas excellent.

Le résultat à 81.6 GFlop/s sur le test Linpack place la grappe I-Cluster en position 385 au top-500.

Le travail d'optimisation mené a touché presque tous les niveaux, de l'architecture du cluster à l'implémentation du benchmark.

- La topologie du réseau a été testée dans plusieurs configurations. Elle s'est organisée autour de 5 switches (HP ProCurve 4000), interconnectés par des liens Ethernet 100. Les topologies testées ont été le réseau complet, un arbre de switches et un anneau.
- Les pilotes du réseau et le noyau Linux utilisés ont été optimisés en fonction des cartes de I-Cluster (taille des buffers TCP).
- Les librairies mathématiques (Blas) ont été extensivement testées et modifiées. La meilleure solution retenue a été un mélange des implémentations d'Atlas [67] et des versions proposées par Intel de certaines routines de produit matriciel, directement optimisées pour le Pentium III. Les tests de performance ont été effectués à la fois sur des produits de matrice de petite taille (tenant dans le cache), de grande taille

(de l'ordre de celles manipulées par HPL) et sur la routine de factorisation LU directement.
- Les routines de communications ont dû être adaptées, surtout celle d'échange global, en fonction de la topologie du réseau. En particulier la différence de performance entre les communications intra- et extra-switches a conditionné le choix de l'algorithme de diffusion. Le placement des calculs sur les processeurs a également été imposé par le réseau.
- Enfin le benchmark Linpack propose lui-même un grand nombre de paramètres d'algorithmique numérique, à tester et adapter selon la machine cible (taille de blocs, type d'algorithme appelé en fin de récursion, ...).

Chacun de ces paramètres est très technique et fait appel à des connaissances spécifiques. Le travail a été réalisé par une équipe de 5 personnes. L'apport personnel à ce travail a porté sur la mise au point des librairies mathématiques, des routines de communication (algorithme de diffusion) et sur l'étude des valeurs optimales des paramètres algorithmiques de Linpack. La coordination avec les opérations relevant plus du niveau système et/ou réseau a été très étroite car il faut bien sûr tenir compte des interactions possibles entre tous ces paramètres lors des tests de mise au point. Enfin et surtout, la difficulté principale a été la variabilité extrême, surtout au niveau des performances du réseau, en fonction du nombre de noeuds utilisés : l'ensemble des paramètres optimaux sur une exécution à 100 noeuds a été catastrophique sur 225 (15 GFlops/s, soit plus de 5 fois moins que les 81 GFlops/s obtenus à l'optimal !).

Les performances obtenues sur Linpack, sanctionnées par le classement au top-500, justifient pleinement le choix d'une librarie comme MPI pour obtenir d'excellents programmes parallèles sur des applications d'algèbre linéaire du type linpack. Néanmoins l'intéret de ce travail est à nuancer :
- par le volume de travail que cette performance a demandé : 1 mois de travail pour 5 personnes à temps plein. Le travail a de plus nécessité un investissement très important sur des couches bas-niveaux de la machine (topologie du réseau, drivers et noyau Linux à adapter), des librairies de communication (variantes de MPI, versions ré-écrites des broadcasts) et de calcul numérique (Blas optimisées selon le processeur) ;
- par la portabilité de ce travail : toute modification du réseau par exemple (ou d'une autre caractéristique de la grappe) se traduit par une perte brutale des performances.

La limitation de ce type d'approche apparaît ainsi directement sur ce benchmark, presque comme conséquence même de sa performance.

Après la présentation du type de calcul haute-performance que l'on peut effectuer en parallèle sur une application-test comme Linpack, les sections suivantes reviennent au traitement de BQ et MP2 par des environnements parallèles.

3.2 Arpack/Parpack et MPI

Comme il a été exposé en 2.3.3 l'équation de Schrödinger peut être résolue en projetant $\mathcal{H}$ en dimension finie sur un espace de fonctions d'ondes discrétisé de façon à se ramener à des calculs de valeurs propres sur des matrices creuses et symétriques, ce qui se fait bien à l'aide de méthodes itératives.

L'environnement Arpack[3] [40] fournit une implémentation (entre autres) de la méthode de Lanczos avec redémarrage implicite. Une version parallélisée à l'aide de MPI, Parpack, est également fournie. Nous présentons ici l'algorithme, les principes qui ont guidé les concepteurs d'Arpack et l'implémentation sur MPI. Enfin, cette librairie est évaluée sur le problème BQ.

3.2.1 Principes de Arpack/Parpack

3.2.1.1 Espace de Krylov et méthode de Lanczos

Le chapitre précédent introduisait les méthodes itératives qui visent à construire un espace vectoriel $\overrightarrow{X}$ de taille m contenant des bonnes approximations des vecteurs propres recherchés. Au lieu de fixer *a priori* la taille m de l'espace $\overrightarrow{X}$, on peut aussi chercher à construire itérativement cet espace à partir d'un vecteur de départ u. On note par définition $\mathcal{K}_m(u)$ l'espace vectoriel engendré par $\{u, Hu, H^2u, \ldots, H^{m-1}u\}$ appelé espace de Krylov engendré par u. Un algorithme de Lanczos est basé sur une méthode de projection, qui construit à chaque itération une base orthogonale de $\mathcal{K}_m(u)$.

L'outil de base est la factorisation sous forme de Hessenberg d'une matrice. Une matrice A de taille $n \times n$ est dite sous forme Hessenberg (supérieure) si et seulement si $A_{i,j} = 0 \quad \forall i > j + 1$, c'est-à-dire que les cœfficients non-nuls sont au-dessus de la diagonale, sur la diagonale et sur la première sous-diagonale uniquement. Toute matrice est semblable à une forme de Hessenberg avec un changement de base orthonormée. Dans le cas où la matrice A est hermitienne (ce qui est le cas du Hamiltonien), la forme de Hessenberg d'une matrice est donc tridiagonale.

Comme le montre par exemple [53], l'orthogonalisation de la famille de vecteurs $H^iu, i = 0 \ldots m$ aboutit directement à la factorisation de H sous forme Hessenberg. En supposant que $\mathcal{K}_m(u)$ est de rang m et en notant Q_mR_m sa factorisation QR (Q_m étant donc orthogonale, et R_m triangulaire supérieure), la matrice $m \times m\ T \stackrel{\text{def}}{=} Q_m^* H Q_m$ est sous forme de Hessenberg.

L'intérêt supplémentaire est que l'on peut facilement passer de l'espace de taille j à celui de taille $j+1$ en construisant à la fois la factorisation de Hessenberg et le $j+1$-ième

[3] http ://www.caam.rice.edu/software/ARPACK/

vecteur de la base ortogonale de $\mathcal{K}_m(u)$. L'algorithme 1 qui construit cette base est dû à Lanczos. La notation standard est utilisée :

$$\alpha_i \stackrel{\text{def}}{=} T_{i,i}, i = 1 \ldots m, \tag{3.1}$$

$$\beta_i \stackrel{\text{def}}{=} T_{i+1,i}, i = 1 \ldots m-1. \tag{3.2}$$

Le cas où β_{j+1} s'annule est atteint si $\mathcal{K}_m(u)$ contient exactement une valeur propre re-

Algorithme 1 Algorithme de Lanczos

Entrées : un vecteur q_1 normé ; $\beta_1 \leftarrow 1, q_0 \leftarrow 0$.
Itérations :
Pour $j = 1, 2, \ldots, m$ **faire**
$r_j \leftarrow Aq_j - \beta_j q_{j-1}$
$\alpha_j \leftarrow r_j^* q_j$
$r_j \leftarrow r_j - \alpha_j q_j$
$\beta_{j+1} \leftarrow \|r_j\|_2$
$q_{j+1} \leftarrow r_j / \beta_{j+1}$
Fin pour

cherchée. Dans ce cas le calcul itératif est bien sûr interrompu.

Cet algorithme peut être étendu au cas des matrices non hermitiennes. On parle alors d'algorithme d'Arnoldi.

T représente la projection orthogonale de l'opérateur associé à H restreint à $\mathcal{K}_m(u)$: on est bien dans le cadre des méthodes de projection. La suite du calcul consiste donc en la résolution du problème aux valeurs propres pour $T = Q_m^* H Q_m$ (ce qui est facilité par sa structure tridiagonale), qui fournira des approximations "correctes" des éléments propres.

Le principal inconvénient de cette version de l'algorithme de Lanczos est son manque de stabilité numérique. La perte d'orthogonalité entre les vecteurs q_i construits itérativement oblige à les ré-orthogonaliser fréquemment. Saad [58] envisage le cas où la ré-orthogonalisation est nécessaire à chaque itération.

De plus, pour arriver à une précision correcte, il est souvent nécessaire de choisir m de grande taille. Si l'on veut garder en mémoire des vecteurs q_i (pour les réorthogonaliser par exemple) on aboutit à un algorithme qui requiert $\mathcal{O}(nm)$ en mémoire et augmenter m peut être à terme limitatif.

Une possibilité pour répondre à ces deux difficultés est d'utiliser une méthode de redémarrage.

3.2.1.2 Algorithme de Lanczos avec redémarrage

L'idée est d'utiliser le contrôle de l'espace que l'on a grâce à une méthode par itération de sous-espaces, avec la construction itérative de la forme de Hessenberg de H. On applique donc l'algorithme de Lanczos à partir de u_0 pour construire T jusqu'à une valeur fixée de m. On en calcule les valeurs et vecteurs propres (λ_j, u_j), dont on sélectionne ceux qui répondent à l'objectif (par exemple : la plus petite valeur propre λ_0). Et on se sert de u_0, le vecteur propre associé, pour reconstruire un nouvel espace de Krylov $\mathcal{K}_m(u_0)$ et la matrice T associée.

Redémarrage explicite Afin d'améliorer encore la convergence vers les valeurs propres désirées, au lieu de repartir directement de u_0, on peut utiliser un filtrage polynomial des vecteurs propres $u_j, j = 1 \dots m$ de T. On se donne un polynôme ψ et on pose $u_0 \leftarrow \psi(H)u_0$. En notant $(\tilde{\lambda}, \tilde{u_i}), i = 1 \dots n$ les éléments propres exacts de H, si on a $u_0 = \sum_{j=1}^{n} \alpha_j \tilde{u_i}$, alors $\psi(H)u_0 = \sum_{j=1}^{n} \alpha_j \psi(\tilde{\lambda}_j)\tilde{u_i}$. Le polynôme ψ doit donc être bien choisi pour prendre des valeurs "petites" sur les valeurs propres $\tilde{\lambda}_i$ à éliminer, et "grandes" sur celles que l'on veut calculer. Saad [58] propose par exemple d'utiliser les polynômes de Tchebycheff pour ψ.

Redémarrage implicite Le dernier type de méthode est dit avec redémarrage implicite [62]. Il combine la méthode de Lanczos à une factorisation QR avec translations implicites. La méthode QR est un algorithme direct détaillé par exemple dans [39]. Il consiste en une série de factorisations QR de la matrice $H - \mu \cdot I$, H étant initialement factorisée sous forme de Hessenberg (*cf.* algorithme 2).

Algorithme 2 Algorithme QR avec translations

Entrées : H, V, T, avec $HV = VT$ et $V^*V = I$, T sous forme de Hessenberg (factorisation de H).
Itérations :
Pour $j = 1, 2, \dots,$ jusqu'à convergence **faire**
  Choisir une translation $\mu \leftarrow \mu_j$.
  Factoriser $Q_i = H - \mu_i$ sous forme QR.
  $T \leftarrow Q^*TQ$; $V \leftarrow VQ$.
Fin pour

Les colonnes de V convergent vers les vecteurs propres de H et T converge vers une matrice triangulaire dont la diagonale contient donc les valeurs propres de H.

L'algorithme de Lanczos avec redémarrage implicite est alors le suivant (*cf.* les algorithmes (5) page 73 et (6) page 74 pour leur écriture exacte) : en notant k le nombre de valeurs propres recherchées, on commence par effectuer $m = k + p$ itérations de

Lanczos pour construire les $q_i, i = 1 \dots m$ comme décrit dans l'algorithme 1. A ce moment on dispose d'une factorisation de Hessenberg T_m de H restreint à $\mathcal{K}_m(u)$, telle que : $HQ_m = Q_m T_m + f_m e_m^*$ (avec e_m un vecteur orthogonal à $\mathcal{K}_m(u)$), où T_m est sous forme de Hessenberg et Q_m est orthogonale. On applique ensuite QR avec p translations sur cette factorisation. [41] montre que l'on peut ainsi annuler les $k-1$ premières composantes du vecteur résiduel e_m. On a donc une factorisation de Lanczos exacte $HQ_k = Q_k T_k$ dans un espace de dimension k. On peut ensuite recommencer des itérations de Lanczos pour revenir à un espace de dimension m et continuer le processus. Il y a ainsi alternance de p itérations de Lanczos et de factorisations QR pour se ramener à une matrice $k \times k$.

On a donc bien un algorithme de Lanczos avec redémarrage. Il est dit implicite car le filtrage des valeurs propres recherchées, au lieu de se faire par l'application directe d'un polynôme sur les vecteurs propres approchés calculés avant redémarrage, se fait par la factorisation QR avec translation. On montre que ces translations équivalent mathématiquement à l'application d'un polynôme $\psi(X) = \Pi_{j=1}^{p}(X - \mu_j)$, où μ_j sont les translations appliquées. On a donc bien le même principe de filtrage polynomial, caché sous les translations.

Arpack propose une implémentation, sous forme de librairie, de l'algorithme d'Arnoldi / Lanczos avec redémarrage implicite (IRA). Les concepteurs ont ainsi voulu baser leur librairie sur un algorithme itératif qui présente les avantages suivants :

- être stable numériquement, ce qui n'est pas le cas, comme on l'a vu, des implémentations directes d'un certain nombre de méthodes itératives ;
- être basé sur des produits matrice-vecteur. L'utilisateur est donc sollicité uniquement en devant fournir cette opération. Arpack fournit un harnais de contrôle de l'algorithme (contrôle de la convergence, calcul des itérations de Lanczos, factorisations QR et calculs des translations,...) et demande à l'utilisateur, selon un code précis, d'effectuer des produits matrice-vecteur quand nécessaire ("reverse communication").
  Ceux-ci peuvent de plus être implémentés comme boîte noire, prenant en entrée un vecteur et en fournissant un autre en sortie. En particulier cela ne nécessite pas que la matrice H soit explicitement formée (ce qui est le cas dans certaines librairies iteratives, *cf.* [40]). Le travail pour l'utilisateur est donc minimal ;
- pouvoir utiliser au maximum des routines efficaces et bien implémentées car toutes les opérations intermédiaires des algorithmes (QR, calculs de norme, ...) sont de l'algèbre linéaire dense bien maîtrisée au niveau implémentation dans les librairies séquentielles (Blas, Lapack) ;
- la mémoire requise est en $n(m+4) + 3m^2 + \mathcal{O}(m)$. Le code EB12, qui implémente une méthode de Lanczos avec redémarrage explicite en filtrant par des polynômes de Tchebycheff, requiert un espace mémoire en $3nm + 2m^2 + \mathcal{O}(m)$ légèrement supérieur ($n \gg m$) ([40]).

La librairie Arpack apparaît donc comme une implémentation prometteuse d'une méthode performante pour résoudre l'équation de Schrödinger dans certains cas présentés au cha-

pitre précédent BQ. Une fois un algorithme séquentiel choisi se pose la question de sa parallélisation afin d'employer au mieux l'algorithme le plus adapté sur les machines les plus puissantes à la disposition des utilisateurs.

Arpack propose une version parallèle de l'algorithme IRA (Implicitly Restarted Arnoldi), basée sur le standard MPI. La section suivante présente celui-ci.

3.2.2 Principales caractéristiques de MPI

MPI (Message Passing Interface) est actuellement le standard pour la programmation par passage de message [60]. Les bibliothèques de passage (ou échange) de messages offrent à un langage séquentiel (C ou Fortran) des routines de communication et de synchronisation qui permettent la gestion en parallèle des processeurs utilisés. Le programmeur dispose d'une vision exacte des p processeurs à sa disposition, chacun étant identifié par son rang entre 0 et $p-1$. C'est donc à lui de replier son algorithme sur les p processeurs à sa disposition. La programmation est SPMD (Single Program, Multiple Data), c'est-à-dire que chaque processeur charge en mémoire le même code exécutable et l'exécute sur des données différentes. En pratique on peut en rajoutant des tests sur le rang du processeur faire exécuter des instructions différentes à des groupes prédéfinis de processeurs.

3.2.2.1 Primitives de communication

MPI fournit aussi bien des communications point-à-point (send, receive) que des communications globales de tous types (diffusion, échange global, réduction,...). De plus les communications peuvent être bloquantes ou non. Il existe aussi des variantes de chaque primitive en fonction du type de bufferisation que l'utilisateur souhaite employer.

MPI encapsule les types scalaires usuels. Chaque message est identifié par un *tag* (étiquette) et un statut, qui contient *a posteriori* des informations sur le message acheminé. MPI fournit la notion de *groupe* : un groupe est un sous-espace des processeurs disponibles, ce qui permet de partitionner les noeuds de la machine selon une topologie virtuelle en fonction de l'algorithme programmé. Des "wildcards" sont fournies pour remplacer la source (MPI_ANY_SOURCE) ou le tag (MPI_ANY_TAG) d'un message en cas d'une réception.

Des exemples de primitives parmi les plus usuelles sont :

MPI_SEND(`buffer,n,MPI_TYPE,dest,etiquette,group,status,ierr`)
: effectue un envoi bloquant de `n` éléments de `buffer`, de type `MPI_TYPE`, vers le processeur `dest`. Le message sera étiqueté par `etiquette` (ce qui permet éventuellement de le distinguer d'un autre à la réception) ; le processeur `dest` appartient

au groupe de processeurs `groupe` ; l'opération s'achève en laissant des informations sur son déroulement dans `status` et `ierr` ;

MPI_RECV(`buffer,n,MPI_TYPE,source,etiquette,group,status,ierr`) effectue une réception bloquante de `n` éléments dans `buffer`, de type `MPI_TYPE`, attendus du processeur `source`. le message attendu est étiqueté par `etiquette` ; le processeur `source` appartient au groupe de processeurs `groupe` ; l'opération s'achève en laissant des informations sur son déroulement dans `status` et `ierr` ;

MPI_ISEND, MPI_IRECV sont les analogues non-bloquants de ces opérations : le processeur poursuit son travail sans se bloquer en attente d'un partenaire pour communiquer, ce qui permet du recouvrement calcul/communication.

MPI_BCAST(`buffer, count, MPI_TYPE, source, groupe, ierr`) : le processeur numéro `source` envoie à tous les autres (et à lui-même) `count` données de type `MPI_TYPE`, localisées chez lui à l'adresse pointée par `buffer`.

MPI_GATHER(`sendbuffer, sendcount, MPI_TYPE,recvbuffer, recvcount, MPI_TYPE , source, groupe`) : chaque processeur envoie à `source` le contenu de `sendbuffer`. `source` reçoit de chacun en stockant à partir de l'adresse pointée par `recvbuffer`, dans l'ordre défini par les rangs des émetteurs.

MPI_ALLTOALL(`sendbuffer, sendcount, MPI_TYPE, recvbuffer, recvcount, MPI_TYPE, groupe`) : chaque processeur envoie à tous les autres et reçoit de tous les autres. La donnée i envoyée par le processeur j est reçue par le processeur i et stockée à la j-ième place à partir de l'adresse pointée par `recvbuffer`.

3.2.2.2 Primitives de synchronisation

MPI prévoit aussi la gestion des synchronisations entre les processeurs, entre autres à utiliser dans le cas de messages non bloquants. Deux exemples de telles primitives sont :

MPI_WAIT(`requete, status, ierr`) qui permet de bloquer le processeur jusqu'à ce que `requete` ait été accomplie (typiquement c'est l'attente de la fin d'une opération non-bloquante d'un autre processeur pour utiliser ses données) ;

MPI_BARRIER(`groupe, status, ierr`) qui établit une barrière de synchronisation entre tous les processeurs qui vont se bloquer sur cette instruction jusqu'à ce que tous l'aient atteinte.

3.2.3 Parallélisation avec MPI dans Parpack

L'algorithme IRA repose sur des itérations matrice-vecteur. Arpack propose au programmeur toute la structure de contrôle des itérations, le test de convergence, la construc-

tion de la matrice projetée du Hamiltonien sur l'espace de Krylov, etc... Seul le produit matrice-vecteur est laissé au soin du programmeur.

La version parallèle de Arpack, Parpack, fournit des versions parallèles des structures de contrôle (calcul de normes pour la convergence, initialisations,...), qui reposent au choix sur MPI ou sur les BLACS (Basic Linear Algebra Communication System). Il reste donc au programmeur à fournir sa version parallèle du produit matrice-vecteur pour compléter le programme de calcul de valeurs propres.

3.2.3.1 Performances sur BQ en environnement distribué

Dans le cadre d'une discrétisation par différences finies du Hamiltonien et de la résolution de l'équation de Schrödinger avec Arpack, il suffit de paralléliser le produit matrice-vecteur creux.

Une méthode naïve peut être employée : on découpe le domaine maillé (tridimensionnel) par exemple selon une direction z, sans recouvrement entre les sous-domaines. L'emploi d'un schéma aux différences finies pour discrétiser le Laplacien du Hamiltonien implique que chaque nœud du maillage a besoin de connaître les valeurs du vecteur sur les deux nœuds voisins (dans les 3 directions de l'espace). La découpe selon z impliquera donc des communications entre chaque section de l'espace pour que le processeur calculant le produit matrice-vecteur sur l'intervalle $[z_g, z_d]$ puisse avoir accès aux valeurs en $z_g - 2$ et $z_g - 1$, ainsi qu'en $z_d + 1$ et $z_d + 2$. A chaque fois cela signifie le transfert de vecteurs de données de taille $2N_xN_y$, où N_x et N_y désignent le nombre de nœuds selon x et selon y respectivement.

Avec ces données, chaque processeur peut calculer le produit matrice-vecteur localement sur tous les points (x, y, z) tels que $z \in [z_g, z_d]$, soit un volume de calculs en $N_xN_y|z_d - z_g + 1|$, qui est donc plus important que le volume de communications si et seulement si les sous-domaines sont découpés selon z avec une "épaisseur" très supérieure à 2. La fonction potentiel n'agit que localement sur un nœud (elle n'a pas besoin des valeurs autres que (x, y, z)), mais elle contient la description physique de la géométrie de la boîte quantique.

La figure (3.2), page 54, montre le résultat typique d'un calcul que nous avons effectué avec Parpack, sur Cray T3E. Il s'agit de la fonction d'onde de l'état fondamental (d'énergie la plus basse) d'une particule de type électron plongée dans une "boîte" quantique (un potentiel) en forme de "T". Dans une analogie en mécanique quantique il s'agirait d'une bille placée dans une gouttière horizontale en forme de T : la bille aurait alors une probabilité égale de se trouver en n'importe quel endroit de la gouttière car la pesanteur (le potentiel) la "confine" dans la gouttière dont elle ne peut sortir. La fonction d'onde vaudrait donc une constante non-nulle sur toute la surface de la gouttière en T, et 0 à l'extérieur.

On voit sur la figure ici un effet purement quantique de confinement à l'aide d'une "boîte ouverte", car la fonction d'onde quantique (densité de probabilité) est fortement localisée autour de "l'intersection" des deux branches du "T" : la particule a une probabilité plus grande de se trouver à cet endroit qu'en n'importe quel autre point de l'espace. De plus on constate que la probabilité n'est pas nulle que la particule "sorte" du puits puisque la fonction d'onde "déborde" (prend une valeur non nulle) en dehors du puits de potentiel.

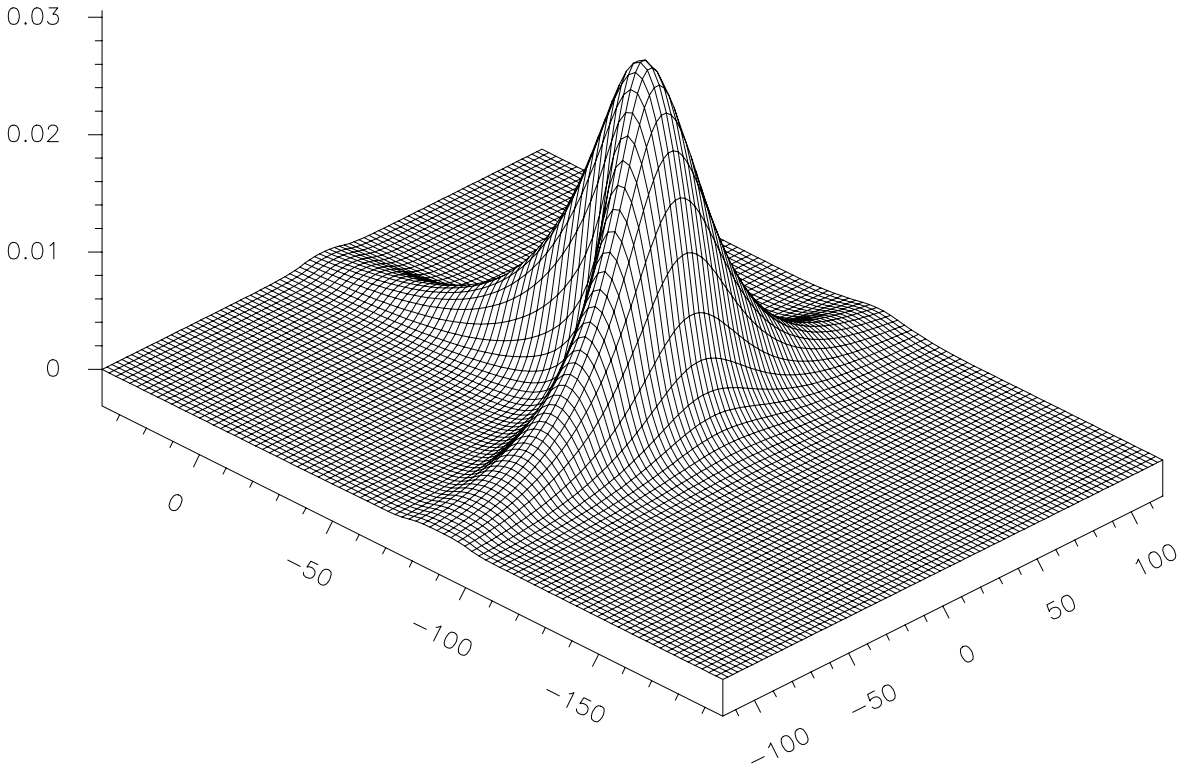


Figure 3.2 *Fonction d'onde de l'état fondamental calculée par Parpack (potentiel en forme de T)*

La figure (3.3), page 55, montre elle l'accélération (temps d'exécution séquentielle divisé par le temps d'exécution parallèle) en fonction du nombre de processeurs, pour le calcul de la plus petite valeur propre avec Parpack, sur la grappe I-cluster du laboratoire ID. L'accélération optimale pour un nombre p de processeurs est p. La baisse de performances quand le nombre de processeurs augmente est dûe à une matrice de taille 25000 qui devient trop petite par rapport aux ressources de calcul disponibles.

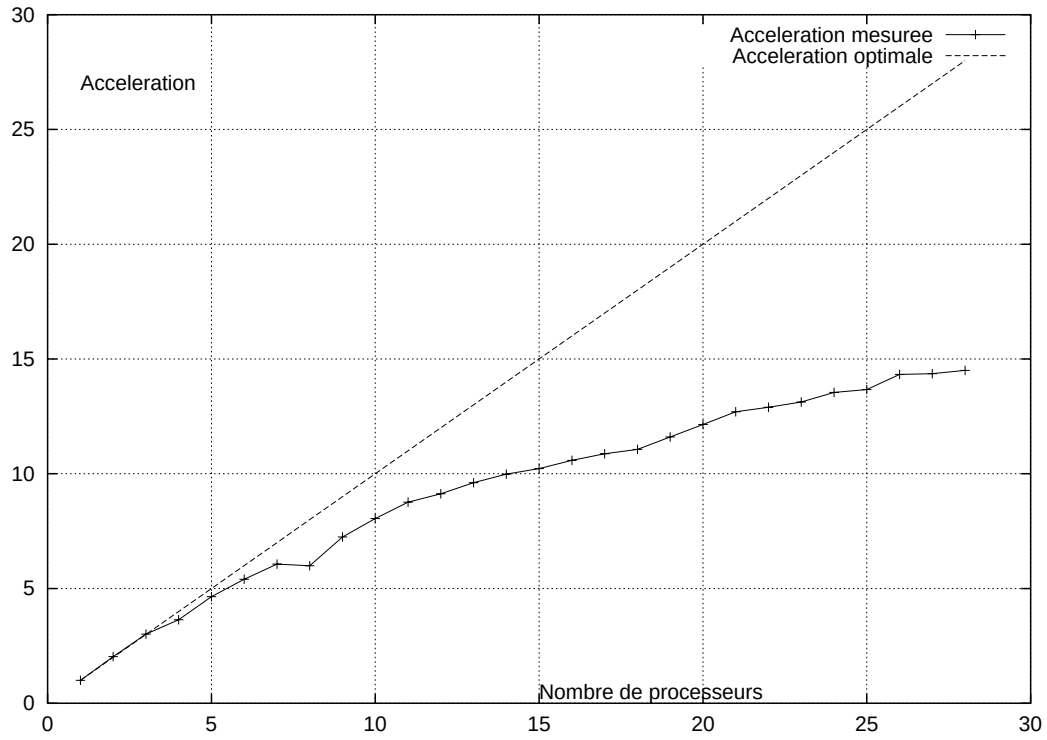


Figure 3.3 *Accélération du calcul de la plus petite valeur propre du Hamiltonien avec Parpack (routine pdnsaupd)*

3.2.4 Arpack/MPI et l'ordonnancement

L'ordonnancement des calculs dans Parpack est purement statique : les vecteurs manipulés sont distribués par blocs et les calculs effectués localement sur ces blocs. Une opération synchronisante comme un calcul de norme est implémenté avec un MPI_REDUCE. Aucun mécanisme n'est prévu pour permettre un ordonnancement dynamique ou même une découpe différente des données. La portabilité sur un environnement hétérogène par exemple est donc compromise. De façon générale la portabilité et les performances de Parpack reposent sur celles de la couche MPI. Le travail présenté sur le benchmark Linpack illustre les problèmes que celles-ci posent.

3.3 SCF et MP2 avec MPI

La section précédente a présenté la parallélisation d'un calcul de mécanique quantique à l'aide de MPI. En chimie quantique également des codes parallèles existent, basés sur l'échange de messages. Nous présentons ici deux implémentations : celle effectuée par Devémy dans le cadre du programme Asterix, puis celle de Nielsen.

3.3.1 SCF : version PVM dans Asterix

En reprenant les notations de la section 2.4.4 et celles de [16], l'algorithme 3 montre comment se fait le calcul SCF. Le calcul consiste donc en l'assemblage de la matrice de

Algorithme 3 Algorithme de calcul SCF

1: **Entrées :** un entier N et une matrice $N \times N$ $c_{\mu,i}$.
2: **Itérations :**
3: **Répète**
4: Calcul de la matrice de densité : $\forall i,j = 1 \ldots N, D_{ij} = \sum_{\mu=1}^{N} c_{i\mu} c_{j\mu}$
5: Calcul de la matrice de Fock : $\forall i,j = 1 \ldots N, F_{ij} = F_{ij} + \sum_{k,l=1}^{N} D_{kl} \left((ij|kl) - \frac{1}{2}(ik|jl) \right)$.
6: Diagonalisation de la matrice de Fock $F = P^T \operatorname{diag}(\epsilon_1, \epsilon_2, \ldots \epsilon_N) P$.
7: $\forall i,\mu$: $c_{\mu i} \leftarrow P_{\mu i}$.
8: **Jusqu'à** Convergence des $c_{\mu i}$

Fock (ligne 5), chaque cœfficient étant obtenu par sommation des intégrales $(ij|kl)$, au nombre total de N^4 qui sont calculées à la volée. Il y a ensuite une diagonalisation de la matrice de Fock (qui est dense) à effectuer (ligne 6).

Deux approches existent pour la parallélisation : celle qui consiste en la distribution exclusive du calcul des intégrales et qui duplique la matrice de Fock sur tous les nœuds de calcul. Et celle qui distribue aussi la matrice de Fock pour effectuer une diagonalisation également parallèle [22]. La première approche se justifie par le coût du calcul des intégrales qui est prépondérant. Néanmoins les processeurs de plus en plus performants ont peu à peu réduit ce coût et transferé le problème sur le stockage de la matrice de Fock, qui se pose lorsqu'elle n'est pas distribuée.

Le logiciel Asterix est développé depuis plus de 30 ans au Laboratoire de Chimie Quantique de Strasbourg. Jérôme Devémy a proposé une version parallèle de SCF basée sur la distribution du calcul des intégrales, en utilisant la librairie de passage de messages PVM. Les intégrales à calculer sont regroupées en sous-ensembles (quartets) respectant les symétries entre les indices, ce qui permet d'éviter des recalculs inutiles. Un schéma de ferme de processeurs est utilisé pour permettre une régulation de charge dynamique et pallier les tailles différentes des quartets. Dans [16], Devémy reporte une efficacité de 97.6% sur 31 noeuds (sur Cray T3D) de son SCF parallèle, sur un test comportant $N = 242$ fonctions de base. Sur cette taille de données, la parallélisation de la diagonalisation ne semble donc effectivement pas un goulot d'étranglement.

Ian Foster et les développeurs de la libraririe ChemIO[4] ont eux proposé une version entièrement distribuée de SCF, basée sur une encapsulation des structures de tableaux parallèles pour les routines de construction de la matrice de Fock, et sur une librairie de passage de messages pour la diagonalisation. Cela leur permet d'attaquer des problèmes

[4] http ://www-fp.mcs.anl.gov/chemio/

plus gros. Dans [22] ils obtiennent 86% d'efficacité sur 512 processeurs, avec le calcul de l'énergie d'une molécule $Si_{16}O_{25}H_{14}$ dans une base de 707 fonctions. L'efficacité de la seule partie "assemblage de la matrice" est 97%, ce qui confirme l'importance de la diagonalisation parallèle pour cette taille de calcul puisque l'on perd 11% d'efficacité sur le total de l'algorithme en ajoutant la phase de diagonalisation pourtant parallélisée. Sur un tel calcul il serait donc inenvisageable d'utiliser une version parallèle de SCF avec une diagonalisation séquentielle de la matrice de Fock.

Il existe donc à l'heure actuelle plusieurs implémentations (massivement) parallèles très efficaces pour l'algorithme SCF, basées essentiellement sur l'échange de messages.

3.3.2 MP2 : version MPI de Nielsen

En ce qui concerne MP2, un calcul qui raffine l'approximation fournie par SCF, on trouve également plusieurs implémentations parallèles. Nous commençons dans cette section par rappeler le calcul MP2, et la façon classique de l'effectuer tel que Pople et Head-Gordon l'ont décrit. Au moins deux logiciels de "Computational Chemistry", GAMESS-UK et Gaussian, proposent des implémentations parallèles de ce calcul, à côté de nombreuses autres méthodes classiques [27]. Nielsen [50], dans le cadre du développement du programme MPQC, fournit une implémentation de MP2 parallélisée en MPI pour machines à mémoire distribuée (d'autres environnements parallèles sont utilisés dans MPQC selon l'architecture cible, par exemple pour SMP).

3.3.2.1 Algorithme et calcul de Pople

Exposé de l'algorithme. Le calcul MP2 a été exposé en section 2.4.4 dans son contexte physique. Nous nous intéressons ici essentiellement à l'algorithme de calcul, que nous détaillons plus.

Etant donné :
- trois entiers N, O et V, tels que $O + V = N$;
- deux matrices C_O de taille $N \times O$ et C_V de taille $N \times V$;
- les N^4 intégrales $< \mu\nu \mid \lambda\sigma >\ , \mu, \nu, \lambda, \sigma = 1 \ldots N$;
- un vecteur $\epsilon_l, l = 1 \ldots N$ précalculé,

MP2 calcule successivement :

$$\forall\ i,j \in 1\ldots O,\ \forall\ a,b \in 1\ldots V \quad : \tag{3.3}$$

$$< ia \mid jb > \ = \sum_{\mu\nu\lambda\sigma=1}^{N} C_{O\mu,i}C_{O\lambda,j}C_{V\nu,a}C_{V\sigma,b} < \mu\nu \mid \lambda\sigma > \ , \tag{3.4}$$

$$= \sum_{\mu} C_{O\mu,i}\left(\sum_{\sigma} C_{V\sigma,b}\left(\sum_{\lambda} C_{O\lambda,j}\left(\sum_{\nu} C_{V\nu,a} < \mu\nu \mid \lambda\sigma > \ \right)\right)\right), \tag{3.5}$$

puis la valeur de l'énergie de corrélation E :

$$E = \sum_{iajb} \frac{< ia \mid jb > \ ^2 + \frac{1}{2}[< ia \mid jb > \ - < ib \mid ja > \ ]^2}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b}. \tag{3.6}$$

La contribution à l'énergie met en jeu les deux termes $< ia \mid jb > $ et $< ib \mid ja > $. Ainsi, à i et j fixés, si on a les valeurs de $< ia \mid jb > $ pour tout a et tout b, on peut sommer leur contribution directement dans E. Cette propriété est utilisée dans l'algorithme de Pople.

Le calcul des transformations $\mu \to i$, $\nu \to a$, $\lambda \to j$ et $\sigma \to b$ s'appelle "transformation à quatre indices". Il correspond à un changement de base, puisque les ERI (Electronic Repulsion Integrals) $< \mu\nu \mid \lambda\sigma > $ sont calculées dans la base des Orbitales Atomiques alors que les ERI finales $< ib \mid ja > $ sont exprimées dans la base des Orbitales Moléculaires.

Méthode de Pople. Pople [28] propose de faire ces calculs dans l'ordre présenté page 59 dans l'algorithme (4)[5]. Pour contrôler l'espace mémoire requis par les intégrales d'OA (il y en a en tout $\mathcal{O}(N^4)$, où N est le nombre d'orbitales dans la base), on recalcule les ERI au lieu de toutes les stocker : ainsi dans les boucles $7-9$, à chaque passage dans la boucle la plus extérieure (sur I), on appelle la routine de calcul des ERI.

Les boucles $6-16$ parcourent tous les couples μ,ν pour un seau donné de couples λ,σ, ce qui permet d'effectuer la transformation $\mu \to i$ pour tous les $\nu = 1\ldots N$ et les λ,σ du seau. La boucle $17-25$ calcule ensuite la transformation $\nu \to 1, \forall\ a = 1\ldots V$ $(18-20)$, puis $\lambda \to j$, du moins pour les valeurs de λ du seau considéré. Au fur et à mesure des parcours de tous les seaux on obtient donc après l'instruction 26 toutes les intégrales triplement transformées $< ia \mid j\sigma > $ pour les i du seau I, tous les $a = 1\ldots V$ et tous les $j = 1\ldots O$. Cela permet de faire la dernière transformation $(28-30)$ $\sigma \to b$ $= 1\ldots V$. A la ligne 31 l'algorithme a finalement tous les $< ia \mid jb > $ pour un couple i,j donné et donc peut calculer la contribution à l'énergie MP2 pour ce couple.

[5]On appelera ici "seau" un groupe d'indices $i \in 1\ldots O$. C'est la traduction adoptée dans ce document pour le terme *batch* employé par Pople.

Algorithme 4 Algorithme de Pople

1: **Entrées :** 3 entiers N, O, V tels que $N = O + V$.
2: Les matrices C_O et C_V de tailles respectives $N \times O$ et $N \times V$, le vecteur ϵ de taille N.
3: **Sortie :** l'énergie au deuxième ordre : E.
4: **Pour tous** les seaux I sur i **faire**
5: **Pour tous** les seaux L sur λ, σ **faire**
6: **Pour tous** les seaux M sur μ, ν **faire**
7: **Pour tous** $\mu, \nu, \lambda, \sigma \in LM$ **faire**
8: Calcule $< \mu\nu \mid \lambda\sigma >$
9: **Fin pour**
10: **Pour tous** $\mu, \nu \in M, i \in I$ **faire**
11: **Pour tous** $\lambda, \sigma \in L$ **faire**
12: $< i\nu \mid \lambda\sigma > \; += < \mu\nu \mid \lambda\sigma > \; *C_{O\mu,i}$
13: $< i\mu \mid \lambda\sigma > \; += < \mu\nu \mid \lambda\sigma > \; *C_{O\nu,i}$ {1ère transformation μ en i.}
14: **Fin pour**
15: **Fin pour**
16: **Fin pour**
17: **Pour tous** $\lambda, \sigma \in L$ **faire**
18: **Pour** $a = 1 \ldots V, \nu = 1 \ldots N$ **faire**
19: $< ia \mid \lambda\sigma > \; += < i\nu \mid \lambda\sigma > \; *C_{V\nu,a}$ {2ème transformation ν en a.}
20: **Fin pour**
21: **Pour** $j \leq i, a = 1 \ldots V$ **faire**
22: $< ia \mid j\sigma > \; += < ia \mid \lambda\sigma > \; *C_{O\lambda,j}$
23: $< ia \mid j\lambda > \; += < ia \mid \lambda\sigma > \; *C_{O\sigma,j}$ {3ème transformation λ en j.}
24: **Fin pour**
25: **Fin pour**
26: **Fin pour**
27: **Pour tous** $i \in I, j \leq i$ **faire**
28: **Pour** $a = 1 \ldots V, b = 1 \ldots V, \sigma = 1 \ldots N$ **faire**
29: $< ia \mid jb > \; += < ia \mid j\sigma > \; *C_{V\sigma,b}$ {4ème transformation σ en b.}
30: **Fin pour**
31: $E +=$ contribution(i, j)
32: **Fin pour**
33: **Fin pour**

Les calculs $12 - 13$ et $22 - 23$ tiennent compte des symétries dans les ERI $\mu \leftrightarrow \nu$ et $\lambda \leftrightarrow \sigma$. En effet on a $< \mu\nu \mid \lambda\sigma > \; =< \nu\mu \mid \lambda\sigma >$ et donc

$$< i\mu \mid \lambda\sigma > \; = \sum_{\nu} < \nu\mu \mid \lambda\sigma > \; C_{O\nu i},$$

ce qui justifie la ligne 13. La ligne 23 provient du même calcul avec la symétrie $\lambda \leftrightarrow \sigma$.

La complexité en mémoire et en nombre d'opérations, pour tout l'algorithme, est pré-

Instructions	Mémoire	Nb. opérations
8	$\|L\|\|M\|$	$\frac{O}{\|I\|}\mathcal{O}(N^4)$
12-13	$\|I\|N\|M\|$	ON^4
19	V	OVN^3
22-23	$\|I\|OVN$	O^2VN^2
29	V^2	O^2V^2N
31	1	O^2V^2

TAB. 3.1 *Complexité (temps/mémoire) de chaque étape du calcul selon Pople*

sentée dans le tableau 3.1.

Les étapes dominantes sont donc, pour la mémoire, le stockage des intégrales triplement transformées $< ia \mid j\sigma >$ en $|I|OVN$; et le calcul de la première transformation en ON^4. En pratique cependant, il faut aussi prendre en compte la complexité du calcul des ERI (ligne 8). Même si elle n'est que de $\mathcal{O}(N^4)$, la constante multiplicative (coût du calcul d'une intégrale) est importante et peut rendre cette étape plus coûteuse que la première transformation.

3.3.2.2 Parallélisation de MP2 par Nielsen

Deux versions parallèles de MP2 sont proposées par Nielsen. Dans les deux cas, le calcul des intégrales $(\mu\nu|\lambda\sigma)$ est distribué, comme dans la version parallèle de SCF, selon les valeurs de μ et ν. Dans la version (V1) chaque processeur effectue la transformation $\mu \rightarrow i$ sur les intégrales qu'il possède, avant un premier échange global pour que chacun récupère les intégrales $(i\nu|\lambda\sigma)$ complètement calculées et encore distribuées selon ν. Cela permet d'effectuer ensuite les transformations $\lambda \rightarrow j$ et $\sigma \rightarrow a$, en obtenant ainsi des intégrales triplement transformées et distribuées selon a. La 4ème transformation est enfin faite et chaque processeur a les $(ia|jb)$, pour un i, ses valeurs de a, tous les j et tous les b. Il faut donc un deuxième échange global personnalisé afin de récupérer les $(ia|jb)$ pour un i, un j, tous les a et tous les b. Ceci obtenu, chaque processeur peut localement calculer la contribution à l'énergie MP2 du couple i, j avec une ultime communication globale.

Cette version a le défaut d'imposer un échange global volumineux après la 1ère transformation puisqu'il faut échanger au total ON^3 intégrales mono-transformées. Dans la version $(V2)$ la distribution des calculs est différente : les $(\mu\nu|\lambda\sigma)$ sont initialement distribués uniquement sur λ. L'idée est de travailler localement au maximum, donc de distribuer selon un cœfficient qui sera transformé plus tard que les autres, ici le troisième. Cela reporte l'échange global à un moment où les intégrales transformées seront moins nombreuses. L'inconvénient est que l'on se prive ainsi d'une symétrie existant entre les cœfficients λ et σ, ce qui impose de calculer deux fois les intégrales. De plus la distribution selon un unique cœfficient conduit à un problème de déséquilibre dans la charge de

calculs, qui se sent particulièrement quand le nombre de processeurs augmente.

La figure 3.4, tirée de [50], présente les performances obtenues avec ces deux implémentations de MP2. L'exécution a eu lieu avec une base de taille $N = 253, O = 29$, sur une Paragon d'Intel. On voit clairement la limitation dûe aux communications pour l'implémentation $(V1)$. $(V2)$ est nettement meilleur mais nécessite un algorithme de régulation de charge pour être réellement efficace.

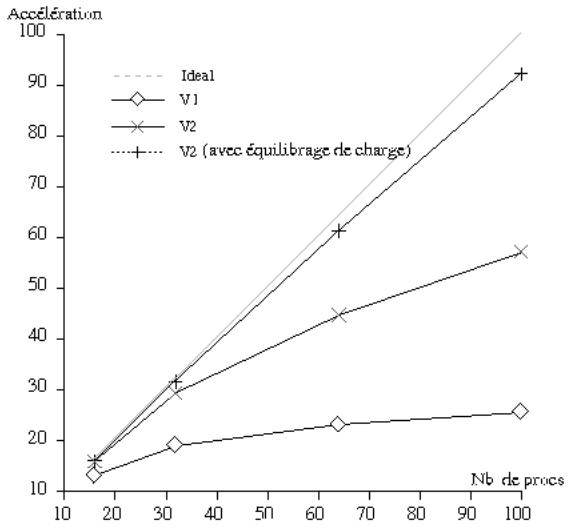


Figure 3.4 *Accélération de MP2 dans les implémentations de Nielsen*

3.4 Gaussian et Linda

En dehors de MPI, d'autres environnements de programmation parallèle existent qui sont utilisés pour ces calculs en chimie quantique. L'environnement Linda a été développé à l'université de Yale et a été utilisé dans le logiciel commercial Gaussian-94. SCF d'abord, puis MP2, ont ainsi bénéficié d'implantations parallèles.

3.4.1 L'environnement Linda

L'environnement Linda[6] propose un paradigme de programmation parallèle qui a l'avantage d'être conceptuellement très simple. Le programme définit de façon SPMD plusieurs processus qui accèdent à une mémoire (virtuellement) partagée, nommée *tuple-space*. Chaque donnée (ou *tuple*) dans le tuplespace est associée à un identificateur qui est

[6]http ://www.cs.yale.edu/Linda/linda.html

une chaîne de caractères. Un tuplespace peut donc être vu comme une table de hachage. Les accès sur les tuples peuvent être :

- en lecture : `rd` permet de lire la valeur : `rd("idf",? x)` renvoie la valeur associée à `"idf"` dans la variable `x`. `rdp` effectue la même opération en testant la disponibilité du tuple demandé dans le tuplespace ;
- en écriture : `out` permet d'écrire une valeur dans le tuplespace. `out("wxyz",200)` crée ainsi le tuple 200 associé à l'identificateur `wxyz` ;
- en lecture destructive : `in` lit et enlève du tuplespace la valeur demandée. La variante `inp` teste d'abord la disponibilité du tuple.

Enfin, la fonction `eval` permet d'exécuter une fonction en parallèle dans un processus Linda qui génère des tuples.

Il n'y a donc pas de notion de messages échangés : les "communications" sont effectuées au niveau des accès au tuplespace. Il n'y a pas non plus explicitement d'opérateur de synchronisation : en fait celles-ci sont effectuées par les accès en exclusion mutuelle des tuples par l'opérateur `in`.

3.4.2 Gaussian

Le logiciel Gaussian[7] est l'un des standards de la "computational chemistry". Il fournit une vaste gamme d'algorithmes de calculs, ainsi que des bases de données contenant, par exemple, des bases d'orbitales pour les calculs *ab-initio*.

Les calculs qu'il permet sont les suivants :

- Énergie : SCF, méthodes de perturbations d'ordre 2, 3, 4 et 5, directes ou non. Les méthodes par interaction de configurations (CI), Coupled-Cluster (CC) et Density Functional (DFT) sont aussi fournies.
- Optimisation de géométrie, calculs de propriétés moléculaires comme les moments multipôlaires par exemple.

Gaussian est développé depuis le début des années 1970. Sa dernière version date de 1998, et il fournit depuis 1994 des versions parallèles de SCF et DFT, qui utilisent Linda sur les architectures à mémoire partagée.

3.4.3 Performances

Gaussian présente un test de calcul MP2 sur une base de taille $N = 264$

Peu d'informations sont disponibles sur les performances parallèles du MP2 de Gaussian mais cette courbe fournie sur le site web officiel montre une perte de performance sur un nombre relativement faible de processeurs. Elle est encore plus effective lorsque l'on prend en compte les coordonnées $\log - \log$ des axes.

[7] http ://www.gaussian.com/

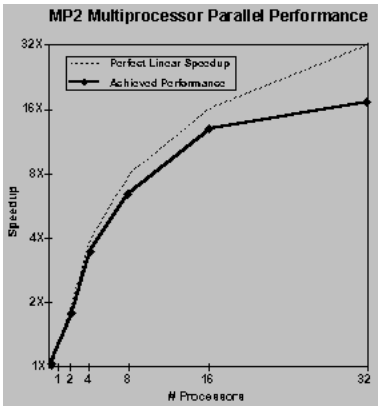


Figure 3.5 *Temps de calcul de MP2 avec le logiciel Gaussian*

3.4.4 Gaussian/Linda et l'ordonnancement

L'ordonnancement des calculs avec Linda est entièrement laissé à la gestion des accès au tuplespace. La simulation d'une mémoire partagée permet théoriquement d'envisager de bons algorithmes d'ordonnancement (*cf.* chapitre 5). Néanmoins aucun mécanisme n'est proposé au programmeur pour personnaliser l'ordonnancement de son application.

3.5 OpenMP

Dans le cadre de calculs de chimie ou mécanique quantique, on trouve également des codes parallélisés en utilisant la norme Open-MP. L'avantage est de pouvoir utiliser des directives ajoutées à un code séquentiel classique pour obtenir directement un programme parallèle sur SMP. La section suivante résume les grandes lignes de la norme Open-MP utilisée actuellement par Valiron et Noga pour paralléliser un algorithme Coupled-Cluster [66].

Présentation de Open-MP. Open-MP[8] permet l'utilisation de *threads* sur machine à mémoire partagée (SMP). La norme Open-MP [8] a été établie par un consortium de constructeurs et d'utilisateurs afin de fixer un ensemble de directives qui permet à un compilateur C ou Fortran classique de générer du code multithreadé. Elles étendent le modèle de programmation séquentiel en ajoutant des instructions pour distribuer les données, synchroniser les processus ou les ordonnancer.

Open-MP utilise un modèle d'exécution série-parallèle : le programme principal s'exécute en séquentiel dans un processus maître jusqu'à atteindre une directive parallèle. À

[8]http ://www.openmp.org

ce point le maître lance des processus fils qui s'exécutent concurrentiellement. Les processus peuvent accéder de façon concurrente à des données partagées. Une fois la partie parallèle (accolade fermante dans un code C, ou `end parallel` en Fortran) terminée les processus se synchronisent et le maître reprend son exécution.

Les directives principales sont les suivantes :
- `parallel` [`clause`] définit le début d'une région parallèle. La `clause` précise la distribution des variables définies dans la région (privées, partagées,...).
- `for`, `sections`, `single` définissent des structures de contrôle distribuées entre les processus. `for` permet d'effectuer des boucles itératives. Les itérations sont réparties sur les processus. `sections` permet de découper une suite d'instructions en sections, chacune d'elles étant exécutée par un unique processus. `single` définit un groupe d'instructions à effectuer en exclusion mutuelle par un seul processus.
- `barrier`, `atomic` ou `flush` permettent des synchronisations entre les processus, sur des instructions pour les deux premières, sur une donnée partagée pour `flush`.

Les directives peuvent être imbriquées, avec certains restrictions néanmoins.

La directive `scheduling` permet de préciser un ordonnancement des threads de calcul : il peut être par blocs, cyclique, cyclique par blocs ou de type "ferme de processus" et donc dynamique.

Enfin, Open-MP fournit une librarie d'utilitaires pour l'exécution qui permettent :
- l'accès à l'environnement d'exécution (nombre de processus à la disposition du programme par exemple) ;
- l'accès à des primitives de verrou.

Coupled-Cluster. Open-MP permet de paralléliser à faible coût de programmation des nids de boucles complexes, du genre de ceux utilisés dans Coupled-Cluster, où de fortes dépendances de données et des accès mémoire désordonnés rendraient délicate la gestion de la localité, par exemple par échange de messages. C'est cette approche que suivent par exemple Valiron et Noga dans [66]. Ils obtiennent une accélération allant jusqu'à 3 sur 4 processeurs, sur une machine à mémoire partagée.

3.6 Conclusion : contrôle de l'ordonnancement dans les codes de mécanique et chimie quantique

Dans ce chapitre nous avons présenté le travail réalisé en terme de calcul haute-performance sur le benchmark Linpack. Constatant le travail qu'un benchmark requiert, nous avons ensuite présenté plusieurs environnements plus ou moins standardisés qui per-

Env. parallèle	Paradigme	Problème traité	Ordonnancement
MPI	Messages	Linpack	codé dans le programme selon la machine
(P)arpack	Messages	BQ	laissé au soin du programmeur
Gaussian	Mémoire partagée (Linda)	SCF, MP2	caché au programmeur
Open-MP	Mémoire partagée	Coupled-Cluster	réalisé automatiquement

TAB. 3.2 *Résumé des environnements parallèles, des problèmes traités et des facilités d'ordonnancement proposées.*

mettent déjà de programmer efficacement, au moins sur certains types de machines, les algorithmes requis en mécanique quantique. Sur des machines comme des clusters de PC ou des machines à mémoire partagée, les performances peuvent être excellentes (*cf.* les calculs MP2 de Foster *et al.* ou la parallélisation Open-MP sur SMP). Le tableau 3.2 synthétise ce qui a été présenté dans ce chapitre.

Ces environnements ont tous deux limitations fortes : tout d'abord les programmes sont très peu portables. L'obtention de 81.6 GFlops/s sur le benchmark Linpack a nécessité le codage de l'algorithme de Broadcast de MPI. Un changement de machine, ou une simple modification de son réseau, fait perdre toute la performance (l'ajout de quelques nœuds sur un test avait fait chuter la performance de 75 à 16 GFlops/s par inadaptation de la topologie du réseau). Le cas est encore plus vrai pour Open-MP qui propose des solutions très intéressantes, mais uniquement sur mémoire partagée : il faut reprogrammer l'ensemble – y compris en changeant de modèle de programmation – pour passer sur machine à mémoire distribuée.

De plus ces environnements laissent au programmeur le soin de programmer lui-même les structures de contrôle permettant justement de garantir l'efficacité du programme. Par exemple, le programme (V2) de Nielsen nécessite l'ajout de modules de régulation de charge – travail qui revient au programmeur – pour gagner en efficacité. Ces structures de contrôle ne sont pas partie intégrante de l'algorithme initial à programmer mais sont des méta-algorithmes chargés de gérer le parallélisme.

Ainsi, non seulement ces environnements rajoutent une charge de travail non négligeable à l'implémentation de l'algorithme, mais en plus ils limitent la portabilité d'un programme en terme de performances. Pour garantir à la fois portabilité et performance, d'autre outils de programmation parallèle existent. Nous présentons dans le chapitre suivant l'environnement Athapascan qui rentre dans ce cadre.

4

L’environnement Athapascan

Ce chapitre introduit un autre modèle d’exécution parallèle : le flot de données, ainsi qu’un outil, l’environnement Athapascan, qui l’analyse pour ordonnancer au mieux le programme selon des critères à préciser, de façon automatique. Cela permet ainsi de pallier les inconvénients des environnements présentés dans le chapitre 3. La section 4.1 introduit formellement le flot de données et le graphe qui permet de l’analyser. La section suivante décrit l’environnement Athapascan qui implémente son calcul dynamique. La section 4.3.1 montre sur l’exemple de BQ un premier programme athapascan. En fin de chapitre, nous montrons comment, dans le cadre de la programmation du flot de données ATHAPASCAN , on peut contrôler de façon asynchrone la convergence d’un algorithme synchrone tel celui qui sert à résoudre BQ.

4.1 Modélisation de l’exécution : description du flot de données et graphes

A partir de ce chapitre nous caractérisons un programme par les calculs qu’il effectue (“tâches”) sur un ensemble de données. Ce double ensemble – tâches et données – forme un graphe où l’on retrouve les grandeurs principales du programme. Ce graphe permet également la formalisation du problème d’ordonnancement du programme.

4.1.1 Représentation de l'exécution par un graphe

Dans la suite nous caractérisons deux représentations d'un programme sous la forme d'un graphe orienté sans cycle. Nous considérons que ces représentations sont une spécialisation d'un graphe plus général : le graphe de tâches. Un graphe de tâches est composé des nœuds représentant les tâches et des arêtes non orientées représentant les dépendances entre les tâches.

Le graphe de précédence. Un graphe de précédence définit un ordre partiel entre les tâches selon la sémantique spécifiée par le langage. Cet ordre traduit les relations de dépendance entre les tâches comme des spécifications d'un ordre de précédence : les arcs entre les nœuds sont orientés.

Les dépendances étant introduites pour régler des conflits d'accès à des données, elles peuvent être interprétées comme des communications entre les tâches. Dans ce type de graphe une tâche est considérée prête dès que toutes les tâches qui la précédent dans le graphe sont terminées.

Notons que les graphes du type série-parallèle *fork*/*join* sont des graphes de précédence.

Le graphe de flot de données. Si les dépendances entre tâches sont considérées comme des communications d'échange de données, il est possible d'ajouter les informations sur les données communiquées dans le graphe : une donnée accédée par une tâche peut être caractérisée par le type d'opération que la tâche va effectuer sur elle :

- *lecture seule* : la tâche ne modifiera pas la donnée mais se contentera de lire sa valeur ;
- *écriture seule* : la tâche ne peut utiliser la donnée que comme "leftvalue" ;
- *écriture en accumulation* : plusieurs tâches vont effectuer une opération associative et commutative sur une donnée en écriture (typiquement, un calcul de somme itérée) ;
- *modification*, ou lecture-écriture : la tâche va modifier la valeur de la donnée.

Ce graphe est appelé alors graphe de flot de données (GFD).

Définition 1 *Le **graphe de flot de données** associé à un algorithme est le graphe $G = (V, E)$ tel que les tâches (V_t) et les données (V_d) forment l'ensemble $V = V_t \cup V_d$ des nœuds, et les accès des tâches sur les données forment l'ensemble E des arêtes. Ce graphe est biparti puisque $E \subset (V_t \times V_d) \cup (V_d \times V_t)$.*

La signification d'une arête est la suivante : soient $t \in V_t$ une tâche et $d \in V_d$ une donnée, l'arête $(t, v) \in E$ traduit un droit d'accès en écriture de la tâche t sur la donnée d, et l'arête (v, t) un droit d'accès en lecture de la tâche t sur la donnée d.

La différence entre les graphes de précédence et de flot de données se situe au niveau des synchronisations. Dans les graphes de flot de données une tâche est considérée prête dès que les données qu'elle a en entrée sont disponibles. Notons néanmoins qu'un graphe de flot de données contient les informations d'un graphe de précédence.

Pour la régulation de charge ce type de graphe fournit plus d'informations, puisque les données communiquées entre les tâches sont identifiées – les tailles des données peuvent également être fournies. Cette information peut être utilisée pour exploiter la localité des données lors du placement de tâches ou pour le routage des données accédées vers le site d'exécution d'une tâche.

4.1.2 Grandeurs caractéristiques du graphe

Les grandeurs caractéristiques d'un programme parallèle ont leur équivalent en terme de théorie des graphes et peuvent être mesurées sur le GFD :

- T_1 est le temps d'exécution séquentielle de l'algorithme décrit par le graphe. C'est la somme des temps d'exécution de chaque tâche, ou encore le nombre de tâches $|V|$ dans un modèle à tâches unitaires[1]. On parle aussi de travail de l'algorithme car à une constante multiplicative près, le temps est confondu avec le nombre d'opérations.
- T_∞ est le temps d'exécution parallèle de l'algorithme sur une infinité de processeurs. C'est donc aussi la profondeur du graphe, c'est-à-dire le plus long chemin du graphe (chemin critique), qui représente une suite de tâches obligatoirement exécutées séquentiellement.
- Le diamètre d du graphe représente le nombre minimal de processeurs requis pour exécuter l'algorithme en T_∞.

4.1.3 Ordonnancement du graphe

[11] définit formellement l'ordonnancement d'un GFD. Étant donné un GFD l'ordonnancement est une fonction qui spécifie, pour une architecture de machine donnée (processeurs $P_i, i = 1 \dots p$, mémoires $M_i, i = 1 \dots p$) :

- un site d'exécution pour chacun de ses noeuds tâches ;
- un emplacement dans l'une des mémoires pour chacun de ses noeuds données.

Les sites et espaces mémoires doivent être tels que l'ordre d'exécution des tâches et d'accès aux données soit compatible avec l'ordre partiel définit par le graphe.

A tout ordonnancement $\mathcal{S}$ on peut associer un temps $t(\mathcal{S})$, qui sera la date de terminaison de la dernière tâche ordonnancée.

[1] Bien sûr le coût "unitaire" est alors fonction de la granularité choisie par le programmeur : la complexité peut s'exprimer en terme de nombre de produits matriciels, de comparaison dans le cas d'un tri, de nombre d'arêtes en théorie des graphes…

Après avoir défini le modèle de programmation à partir du GFD, il reste à fournir un environnement permettant de l'implanter. C'est le cas de l'environnement ATHAPASCAN qui est présenté dans la section suivante.

4.2 L'environnement Athapascan : calcul et interprétation du Graphe de Flot de Données

La bibliothèque C++ ATHAPASCAN [2] [18, 24, 11] offre un environnement de programmation basé sur le modèle défini ci-dessus.

4.2.1 Athapascan

La bibliothèque ATHAPASCAN fournit à l'utilisateur deux mots-clé pour définir tâches et données du GFD.

- Définition des tâches : **Fork**. L'utilisateur définit explicitement la granularité de son programme en encapsulant les procédures qui seront appelées de façon asynchrone. Ces tâches sont créées par appel de la procédure générique `Fork<>()` de la bibliothèque instanciée avec le type de la tâche à créer. Le corps de la tâche sera implanté comme une classe C++ fournissant l'opérateur `()`, qui prend en entrée les paramètres de la tâche. La figure 4.1(a) illustre ces créations.
- Définition des données : **Shared**. La bibliothèque ATHAPASCAN définit une mémoire partagée afin de permettre aux tâches de coopérer. Cette mémoire peut contenir des objets de tout type, et ceux-ci sont déclarés comme des objets standards mais de type `Shared<T>` où `T` représente le type spécifié par l'utilisateur. Les données Shared seront ainsi les noeuds données du GFD.

Pour permettre à l'environnement ATHAPASCAN de gérer les arêtes du GFD (les accès des tâches sur les données), chaque tâche spécifie au moment de sa création les accès qui seront effectués sur les données de la mémoire partagée au cours de son exécution ou lors de l'exécution de toute sa descendance : les différentes données qui seront accédées sont spécifiées dans la liste des paramètres de la tâche et la description de ces droits d'accès (lecture `r`, écriture `w`, accumulation `cw`, modification `r_w`) est faite à l'aide d'un mécanisme de typage : on distinguera ainsi les données Shared accédées en lecture (r) de celles accédées en écriture (w).

Une tâche a un fonctionement entièrement asynchrone par rapport à la tâche qui l'a créée. Cela implique donc, entre autres, que la tâche mère ne peut accéder les résultats de la tâche fille : ces résultats seront exploités par une tâche nécessairement différente. Le système exécutif d'ATHAPASCAN garantit (afin de permettre à d'éventuels ordonnanceurs de faire des estimations fiables sur la durée d'exécution des tâches) que l'exécution d'une

[2] L'url de la page officielle du projet est la suivante : http ://www-id.imag.fr.

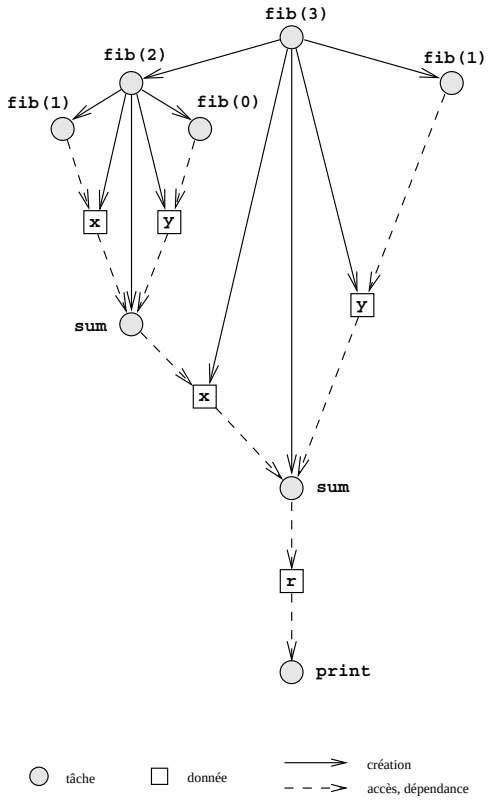
```
struct sum {
   void operator()( Shared_r<int> a,
                    Shared_r<int> b,
                    Shared_w<int> c ) {
      c.write( a.read() + b.read() );
   }
};

struct fib {
   void operator()( int n,
                    Shared_w<int> r ) {
      if( n<2 )
         r.write( n );
      else {
         Shared<int> x, y;
         Fork< fib >()( n-1, x );
         Fork< fib >()( n-2, y );
         Fork< sum >()( x, y, r );
      }
   }
};

struct print~: public à_task {
   void operator()( int n,
                    Shared_r<int> r )
   {
      printf("fib(%d)=%d\n", n, r.read());
   }
};

int main( int argc, char** argv )
{
   int n= atoi( argv[1] );
   Shared<int> r;
   Fork< fib >()( n, r );
   Fork< print >()( n, r );
   return 0;
}
```

(a)



(b)

Figure 4.1 *Calcul du n-ième nombre de Fibonacci en* ATHAPASCAN .
En (a), dans le code C++ utilisant la bibliothèque ATHAPASCAN *, l'expression du parallélisme se fait en créant explicitement des tâches (*Fork*). Les contraintes de précédence entre ces tâches sont déduites des accès effectués sur la mémoire partagée (objets* Shared*). Ces accès sont déclarés dans les prototypes des tâches par typage (*_r *pour une lecture,* _w *pour une écriture). Les accès effectifs aux données sont effectués par les fonctions membres* read() *pour une lecture et* write() *pour une écriture. En (b), le graphe de flot de données d'une exécution de* fib(3)*. Bien que cela soit le cas pour cet exemple, le graphe n'est pas forcément de type série-parallèle.*

tâche a lieu sans aucune synchronisation. Pour cela, une tâche ne pourra débuter son exécution que si toutes les données accédées en lecture sont prêtes, c'est-à-dire que toutes les tâches qui écrivaient sur cette donnée sont terminées.

La figure 4.1 illustre les outils offerts au programmeur par ATHAPASCAN . Le programmeur définit trois tâches : l'une effectuant la somme de deux doubles (sum) ; la seconde effectuant le calcul récursif même (fibo). La dernière qui affiche le résultat (print).

Chacune précise le type d'accès à ses données en mémoire partagée (r ou w). Selon ce type d'accès ATHAPASCAN fournit différents opérateurs de manipulation sur ces données : les valeurs accédées en lecture ont la fonction membre "read()" définie. Les données en écriture peuvent être modifiées grâce à l'opérateur write qui prend en argument la valeur à écrire. Enfin, les tâches sont appelées de façon asynchrone par l'opérateur Fork(). La partie droite de la figure montre le GFD de l'exécution de fib(3).

4.2.2 Construction du Graphe de Flot de Données

[24] décrit comment les modes d'accès sur les données en mémoire partagée permettent au noyau ATHAPASCAN de construire dynamiquement le GFD. Une donnée d est une succession de "versions" d_i, chacune d'entre-elles étant une variable à assignation unique qui contient la valeur de la donnée au top i. Chaque accès sur la donnée, spécifié dans le profil de la tâche, permet de mettre à jour le GFD pour gérer les éventuels conflits.

Lors de la déclaration d'une donnée x par une tâche le graphe est modifié par ajout de deux versions de x : la première qui contient la valeur initiale de x, la seconde qui contiendra la valeur après les futures écritures sur x.

Lors de la création d'une tâche t', en notant t sa mère, le graphe est modifié par ajout de t'. Selon les modes d'accès à ses paramètres, il y a également des modifications dans les versions de ces données en paramètres :

- accès en lecture : il suffit d'ajouter une arête entre la donnée x et la tâche t'.
- accès en écriture : pour une écriture concurrente, il suffit de rajouter une arête entre t' et x. Pour une écriture : soit t avait un accès en écriture simple et alors il faut ajouter une nouvelle version v' de x, qui contiendra le résultat de l'écriture de t' et qui sera antérieure à la version v de x sur laquelle t écrira.

Ainsi ATHAPASCAN peut construire dynamiquement le GFD à partir de la description des accès sur les données. Cela permet, comme illustré dans le chapitre 5, de calculer des ordonnancements du graphe.

4.3 BQ avec ATHAPASCAN

4.3.1 IRA en ATHAPASCAN

Comme expliqué en 3.2 l'algorithme d'Arnoldi avec redémarrage Implicite (IRA) est particulièrement adapté à certains calculs pour la chimie quantique. Cette section illustre sur IRA la programmation en ATHAPASCAN .

L'algorithme IRA [61] expliqué au chapitre 3 en 3.2 est présenté explicitement dans les figures (5), page 73 et (6) page 74.

Algorithme 5 Algorithme d'Arnoldi avec Redémarrage Implicite (IRA) — Itérations.

Fonction **Itérations()**
Entrées : 3 entiers $n > k + p$;
une matrice A de taille $n \times n$;
une matrice V de taille $n \times k$;
une matrice H de taille $k \times k$;
un vecteur r de taille n, qui vérifient :
$AV - VH = r \cdot e_k^t, V^tV = I_k$ et $V^tr = 0$.
Sorties : V, H complétées, de tailles $n \times (k + p)$ et $(k + p) \times (k + p)$;
un vecteur r tels que $AV - VH = r \cdot e_{k+p}^t, V^tV = I_{k+p}$ et $V^tr = 0$.
Pour $j = 1, 2, \ldots, p$ **faire** {p itérations d'Arnoldi}
  $\beta \leftarrow \|r\|$. Si $\beta < \epsilon$, STOP.
  La $k + j$-ième ligne de H devient βe_{k+j-1}^t. $v \leftarrow \frac{1}{\beta} r$. La $k + j$-ième colonne de V devient v.
  $w \leftarrow Av$
  $h \leftarrow V^tw$. La $k + j$-ième colonne de H devient h.
  $r \leftarrow w - Vh$.
  Tant que $\|s\| > \epsilon\|r\|$ **faire**
    $s \leftarrow V^tr$
    $r \leftarrow r - V^ts$
    $h \leftarrow h + s$
  Fin tant que
Fin pour

(La routine `Shift(H,p)` renvoie p valeurs qui serviront à l'algorithme QR avec translation. Plusieurs solutions sont possibles pour ces translations [61]. Celle que nous avons choisie d'implanter utilise les valeurs propres exactes de la matrice H (qui est "petite" et permet donc ce calcul à faible coût). `Shift(H,p)` calcule donc les $k + p$ valeurs propres de H et renvoie les p plus grandes (dans le cas symétrique les valeurs propres sont réelles). On forcera ainsi par translation la convergence vers les k valeurs propres les plus petites de A.)

Les type de données de base sont des matrices et des vecteurs. Les opérations de base à effectuer sont :

(i). calcul de norme euclidienne et de produit scalaire ;

(ii). produits Matrice $\times$ Vecteur, avec la matrice éventuellement transposée ; opérations de type AXpY ;

(iii). produits Matrice $\times$ Matrice

(iv). factorisation QR,

La section suivante présente la façon dont ATHAPASCAN permet de décrire le parallélisme dans l'algorithme IRAM.

La granularité du calcul peut être n'importe laquelle classiquement définie en algèbre

Algorithme 6 Algorithme d'Arnoldi avec Redémarrage Implicite (IRA) — redémarrage.

Fonction **Redémarrage()**
Entrées : 3 entiers $k, p, n > k + p$;
une matrice A de taille $n \times n$;
un vecteur v_1 de taille n, normalisé.
Sorties : V, H de tailles $n \times (k + p)$ et $(k + p) \times k + p$;
un vecteur r tels que $AV - VH = r \cdot e_{k+p}^t, V^tV = I_{k+p}$ et $V^tr = 0$.

Initialisation : $V \leftarrow v_1$; $H \leftarrow (v_1^t A v_1)$; $r \leftarrow Av_1 - v_1 H$.
$[V, H, r] \leftarrow$ **Iterations**$(A, V, H, r, 1, k - 1)$ {Initialisation : V et H vont être de taille $k \times k$.}
Tant que $\|r\| > \epsilon$ **faire**
  $[V, H, r] \leftarrow$ **Iterations**(A, V, H, r, k, p)
  $u \leftarrow$ **Shift**(H, p) {Cf. texte pour Shift()}
  $Q \leftarrow I_{k+p}$
  Pour $j = 1, 2, \ldots, p$ **faire** {Algorithme QR avec la translation u_j}
    $H \leftarrow Q_j^t H Q_j$
    $Q \leftarrow QQ_j$
  Fin pour
  $v \leftarrow (VQ)e_{k+1}$ {v devient la colonne $k + 1$ de VQ}
  $V \leftarrow (VQ) \begin{pmatrix} I_k \\ 0 \end{pmatrix}$ {On ne garde que les k premières lignes et colonnes de VQ.}
  $\beta_k \leftarrow e_{k+1}^t H e_k = H_{k+1,k}$
  $\sigma_k \leftarrow e_{k+p}^t Q e_k = Q_{k+p,k}$
  $r \leftarrow (\beta_k v + \sigma_k r)$
Fin tant que

linéaire : les matrices peuvent être découpées par colonnes, par lignes, par blocs ou cycliquement, par blocs cycliques, ou encore être découpées par blocs (sous-matrices). Nous présentons ici une implantation qui effectue une découpe par blocs. La matrice $A, n \times n$ va donc être vue comme une matrice de taille $N_b \times N_b$ de blocs de n/N_b lignes (on suppose la division exacte) et n/N_b colonnes. Chacun de ces blocs sera déclaré comme *Shared*.

De même les vecteurs (de taille n) manipulés seront des vecteurs de N_b vecteurs de taille N/N_b, chacun étant *Shared*.

La figure 4.2 page 75 montre un extrait du code ATHAPASCAN pour implémenter un type "matrix" qui peut être utilisé dans la mémoire partagée.

"matrix" est une classe C++ classique, qui dérive ici de la classe C++ vector (c'est un exemple d'implantation). ATHAPASCAN exige la définition de certains opérateurs (création par copie — celui-ci, dans l'exemple, est hérité de la classe C++ "vector" et n'est donc pas explicité dans le code de la figure —, destruction), d'une fonction d'empaquetage et d'une fonction de désempaquetage. La première sert à décrire quelles information

```
#include <vector>
#include "athapascan-1.h"

template<class T>
class matrix : public vector<T> {
public :
  matrix(int row_dim_ = 0, int col_dim_ = 0, const T& value = T(0))                matrix
    : vector<T>(col_dim_*row_dim_,value), _col_dim(col_dim_),
    _row_dim(row_dim_) { };
                                                                                   10
  int col_dim() const { return _col_dim; }                                         col_dim

  bool operator==(const matrix<T>& m) const { /* code ... */ }
  matrix<T>& operator+=(const matrix<T>& m);
  ~matrix() { }                                                                    ~matrix
};


/* Packing operator */
a1_ostream& operator << ( a1_ostream& out, const matrix<T>& m) {                   20
    out << m.row_dim() << m.col_dim();
    out.a1_pack ( m.begin(), m.row_dim()*m.col_dim() );
    return out; }

/* unpacking operator */
template<class T>
a1_istream& operator >> ( a1_istream& in, matrix<T>& m) {
  int p, q;
  in >> p >> q;
  m = matrix<T>(p,q);                                                              30
  in.a1_unpack ( m.begin(), p*q );
  return in; }
```

Figure 4.2 *Implémentation en* ATHAPASCAN *d'une classe "matrix"*

sur la classe doivent être transmises lors d'un envoi et la deuxième le format des données en réception.

Calcul de norme et produit scalaire Les opérations vectorielles s'écrivent alors très naturellement en ATHAPASCAN . Par exemple le produit scalaire de deux vecteurs u et v distribués comme ci-dessus va effectuer N_b tâches parallèles, la i-ième prenant en lecture les blocs i de u et v pour effectuer leur produit scalaire localement et écrire en accumulation (somme itérée) le résultat dans une variable partagée qui contiendra le

résultat final. La figure (4.3) présente le GFD associé.

Nous illustrons sur cet exemple les notations introduites en 4.1.2 : classiquement on prendra ici comme unité le produit de deux nombre flottants double précision. T_1 sera alors $\mathcal{O}(n)$ car le produit scalaire de 2 vecteurs à n entrées impliquera n multiplications. $T_\infty = \mathcal{O}(\log n)$ car avec un nombre infini de processeurs à sa disposition on peut effectuer un calcul arborescent tel que la profondeur du graphe de calcul sera logarithmique. Enfin, $d = \mathcal{O}(n)$ dans une version naïve de ce calcul car la première étape du calcul arborescent demande n processeurs (les techniques classiques de regroupement permettent d'être efficace et de n'utiliser que $d_{opt} = n/\log(n)$ processeurs, *cf.* [31] par exemple).

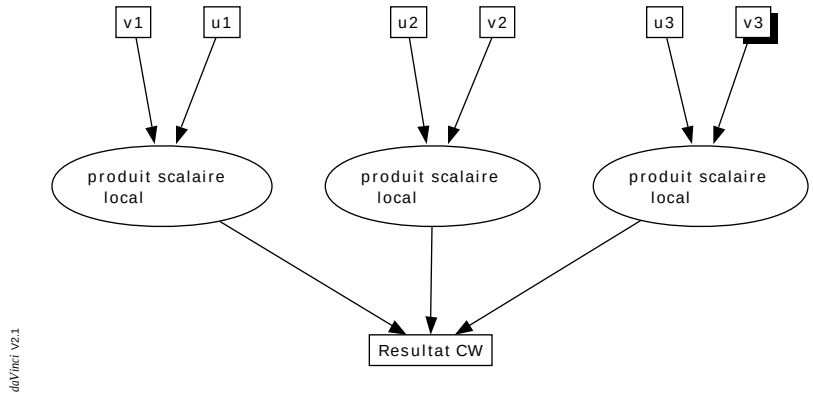


Figure 4.3 *Graphe de Flots de Données associé au calcul d'un produit scalaire*

Produits Matrice $\times$ Vecteur À partir de la formule $y_i = \sum_{j=1}^{N_b} A_{i,j}x_j$, il est immédiat que le calcul du bloc y_i du vecteur résultat est une écriture en accumulation des N_b calculs $A_{i,j}x_j$, eux-mêmes des produits Matrice$\times$ Vecteur effectués en séquentiel. La figure (4.4) présente le GFD de ce calcul **pdgemv** dans le cas de 3×3 blocs.

La figure 4.5, page 78 présente un exemple de code ATHAPASCAN sur le produit matrice-vecteur parallèle `pdgemv`.

On voit la définition de la tâche de calcul séquentiel `task_dgemv` qui prend en entrée 3 paramètres : une matrice (shared) A, un vecteur (shared) v, tous deux en lecture ; et un vecteur (shared) r déclaré ici en écriture concurrente. Cette tâche fait un appel à une routine des Blas (dgemv) pour calculer le produit Av, le résultat étant dans res. Enfin, elle accumule dans r la valeur de res, en utilisant la loi de cumulation définir par `vect_cumul`.

`vect_cumul` est elle-même implantée en ATHAPASCAN grâce à une fonction classe C++, selon la syntaxe usuelle de définition d'un opérateur "()". Cette fonction ne fait ni plus ni moins qu'une addition vectorielle (une fonction des Blas aurait pu être utilisée ici aussi).

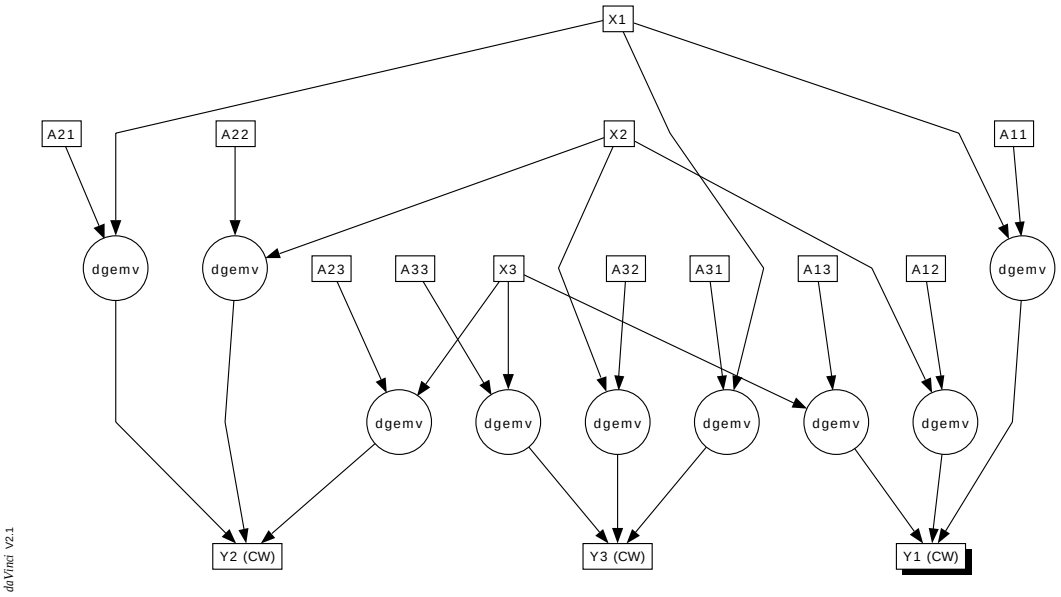


Figure 4.4 *Graphe de Flots de Données associé au calcul d'un produit Matrice× vecteur, pour* 3×3 *blocs.*

La fonction (qui n'est donc pas une tâche parallèle) `pdgemv` va "Forker" les tâches de calcul séquentielles `task_dgemv` avec les paramètres adaptés. Cette fonction prend en argument la matrice A qui est un tableau bidimensionnel de "matrix" : on retrouve la découpe de A par blocs. De même r et v sont des tableaux de "vector", c'est-à-dire des vecteurs distribués par blocs de lignes. `pdgemv` ne fait rien d'autre qu'appliquer la formule de définition du produit matrice-vecteur par blocs (boucles sur i et j).

Produits Matrice $\times$ Matrice Dans le calcul de IRAM cette opération n'intervient que pour un produit $V \times Q$ (V est $n \times (k+p)$ et Q $(k+p) \times (k+p)$), lignes $18-19$ de l'algorithme 6 et pour un produit HQ (produit de deux matrices de taille $(k+p) \times (k+p)$). La taille respective (typiquement, $k+p \ll n$, *cf.* la section 3.2.1.2) de ces matrices incite à dupliquer H et Q. Le produit HQ sera donc effectué en séquentiel. Le calcul VQ sera effectué en parallèle avec V distribué en blocs de lignes (*cf.* les vecteurs des paragraphes précédents) et avec Q accédé intégralement par chaque tâche de calcul.

Chaque tâche **pdgemm** va donc effectuer le produit matriciel d'un bloc de V (accédé en lecture) par la matrice Q (accédée en lecture) pour écrire le résultat dans un bloc de VQ.

AXpY Les opérations de type AXpY se programment sans difficulté comme le produit matriciel, avec un vecteur y en plus accédé en lecture et dont on ajoute les composantes au résultat du produit matrice× matrice.

```
#include "athapascan-1.h"
#include "matrix.h"

struct vect_cumul {
  void operator() (vector<double>& res, const vector<double>& val) {          operator
    int n = res.size();
    for (int i=0 ~; i<n ~; i++)
        res[i] += val[i];
  }
};                                                                              10

struct task_dgemv {
  void operator() (Shared_cw< vect_cumul, vector<double> > r,                   operator
                   Shared_r< matrix<double> > A,
                   Shared_r< vector<double> > v) {
    double un = 1.0;
    double zero = 0.0;
    int pas = 1;
    char trans = 'n';
    int m = A.read().row_dim();                                                 20
    int n = A.read().col_dim();
    int v_row = v.read().size();
    vector<double> *res = new vector<double>(v_row, 0 );
    dgemv(&trans, &m, &n, &un, A.read().begin(), &m,
          v.read().begin(), &pas, &zero, res->begin(), &pas);
    r.cumul( *res );
  }
};

void pdgemv(Shared< vector<double> >* r,                                        30 pdgemv
            Shared< matrix<double> >** A,
            Shared< vector<double> >* v,
            const int nb_blocks ) {
  for (int i=0; i<nb_blocks; i++)
    for (int j=0; j<nb_blocks; j++)
      Fork< task_dgemv>( COST )( r[i], A[i][j], v[j] );
}

// Exemple d'utilisation :
    pdgemv( r, A, v, nb_blocks );                                               40
```

Figure 4.5 *Implémentation en* ATHAPASCAN *d'un produit matriciel parallèle*

Itérations d'Arnoldi Avec ces opérations de base la fonction **Itérations()** (*cf.* algorithme 5) se programme facilement. La figure 4.6 page 79 montre le GFD associé tel que calculé par ATHAPASCAN à un moment donné d'une exécution. L'instruction 11 est un

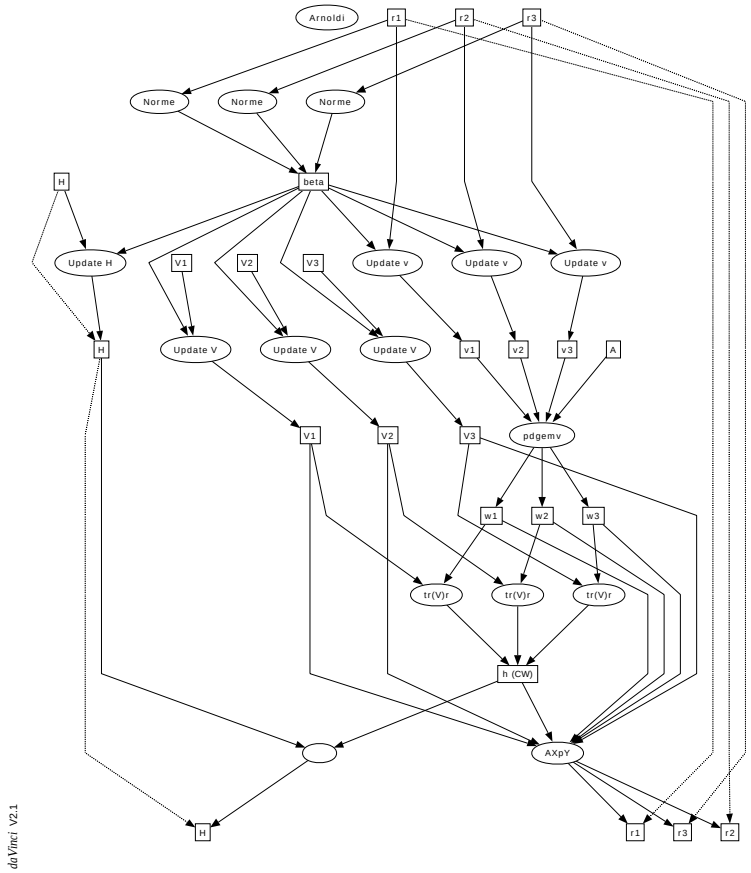


Figure 4.6 *Graphe de Flots de Données associé au calcul des itérations d'Arnoldi, dans le cas d'une découpe en $N_b = 3$ blocs.*

calcul de norme : $\beta \leftarrow \|r\|$, qui peut être effectué en parallèle lors de la mise à jour du vecteur r (ligne 15). L'instruction 12 met à jour la matrice H grâce à β. H est donc accédé en lecture-écriture et β en lecture. β peut aussi être accédée en lecture, avec r, pour a mise-à-jour de $v \leftarrow (1/\beta)r$. v est ensuite accédé en lecture pour la mise-à-jour de la matrice V.

L'instruction 13 effectue un produit matrice$\times$ vecteur $w \leftarrow Av$ déjà décrit. La ligne 14 effectue de même $h \leftarrow V^t w$. Une tâche supplémentaire est créée qui prend la matrice H en lecture/écriture et le vecteur h en lecture pour mettre à jour H dont la dernière colonne devient h.

Redémarrage La phase décrite dans l'algorithme 6 est décrit de la même façon en ATHAPASCAN . La figure 4.7 page 80 montre le GFD associé, tel que calculé par ATHA-

PASCAN lors d'une exécution.

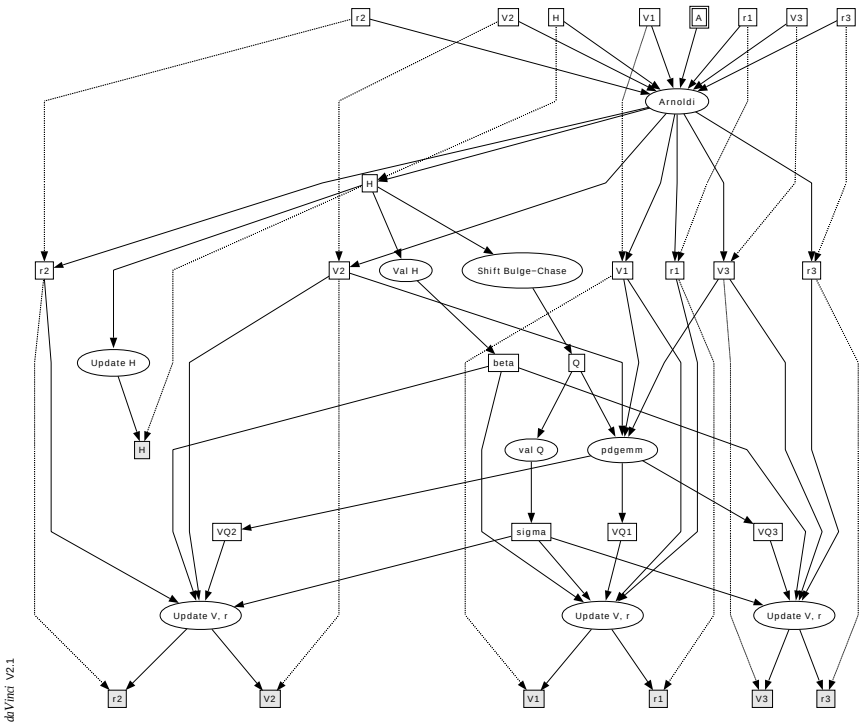


Figure 4.7 *Graphe de Flots de Données associé au calcul du redémarrage des itérations de Arnoldi dans le cas d'une découpe en $N_b = 3$ blocs.*

4.3.2 BQ : contrôle de l'erreur

Comme expliqué à la section précédente, une classe de méthodes numériques largement utilisées en mécanique quantique est celle des méthodes itératives : une opération élémentaire (produit matrice - vecteur par exemple ici) est répétée jusqu'à convergence, avec une structure de contrôle qui teste régulièrement si le critère de convergence a été atteint ou non.

Ce test nécessite souvent une synchronisation entre les tâches qui participent au calcul. C'est le cas dans la version proposée par Parpack, qui calcule une norme mettant en jeu toutes parties du vecteur mises à jour en parallèle par le produit matrice-vecteur. Ce genre de synchronisation est particulièrement visible sur les GFD ATHAPASCAN , comme par exemple dans la figure 4.7 où l'on voit le goulet d'étranglement que représente le calcul de norme qui s'accumule dans la variable `beta`. Dans une approche parallèle, un premier objectif est de diminuer le temps d'exécution. Pour cela, il faut enlever des synchronisations.

Nous présentons ici des techniques générales permettant de réduire le nombre de points de synchronisation dans un algorithme itératif. Une première solution est de changer complètement l'algorithme de résolution pour utiliser une méthode asynchrone (section 4.3.2.1. La section 4.3.2.2 présente une solution k-pas. Vient ensuite un algorithme amorti pour ce contrôle et enfin un compromis entre ces deux premiers algorithmes.

4.3.2.1 Méthodes asynchrones

Une première possibilité est de changer complètement l'algorithmique numérique utilisée pour employer des méthodes asynchrones. Nous suivons ici la présentation faite dans [12]. L'idée est qu'un algorithme itératif effectue à chaque itération une transformation H d'un espace vectoriel E dans lui-même. La convergence est obtenue quand on a trouvé un point fixe x^* de H. Pour introduire du parallélisme on décompose E en un produit cartésien de sous-espaces E_i : $E = E_1 \times E_2 \times \ldots \times E_n$. La transformation H s'écrit alors :

$$\begin{aligned} H \ : & E_1 \times E_2 \times \ldots \times E_n \longrightarrow E_1 \times E_2 \times \ldots \times E_n \\ & x = (x_1, x_2, \ldots, x_n) \longmapsto (H_1(x), H_2(x), \ldots, H_n(x)). \end{aligned}$$

Les méthodes asynchrones vont alors consister en l'itération parallèle de chaque opérateur H_i sur x, du type $x^{k+1} = H_i(x^k)$. Comme cela se fait sans synchronisation entre les tâches de calcul, à chaque itération celles-ci utiliseront des composantes $x_i^{k'}$ d'itérés k' potentiellement différents (par exemple à cause de délais de transmission dans un réseau, ou de temps d'accès en mémoire partagée non homogène).

[1] prouve la convergence numérique de certaines classes d'algorithmes désynchronisés. C'est par exemple le cas de la résolution itérative de systèmes linéaires $(I - A)x = b$, en utilisant une norme de type "max" ($\|x\| \stackrel{\text{def}}{=} \max_i |x_i|$) et si le rayon spectral de A est inférieur à 1.

Cependant, même si la convergence reste garantie dans certains cas, la vitesse de convergence est modifiée par rapport à un algorithme synchrone. [1] montre que dans certains cas la convergence peut être meilleure pour un schéma asynchrone (même si le nombre de communications est supérieur, ce qui risque de faire perdre en temps de communication un gain de temps numérique). [12] évoque également le cas des méthodes de décomposition de domaine où la modification du nombre de sous-domaines indépendants peut agir comme préconditionneur qui accélère la convergence.

4.3.2.2 Contrôle d'une itération et synchronisation : méthode k-pas

Sans évoquer davantage la modification algorithmique du calcul, on peut s'attacher à la modification algorithmique de la structure de contrôle de la convergence. Nous nous intéressons ici au cas où la terminaison du calcul est décidée lors de l'itération n_* par

un calcul nécessitant une synchronisation totale des calculs. Typiquement, cela sera le cas quand l'arrêt sera conditionné par la norme d'un vecteur qui devient inférieure à une tolérance ϵ donnée. On fera l'hypothèse suivante dans tout cette section : le test d'arrêt, à valeurs booléennes, est une fonction croissante du nombre d'itérations.

Une possibilité est de calculer le test d'arrêt à chaque itération. On est assuré ainsi d'en effectuer n_*, au prix de n_* synchronisations.

La première amélioration possible est d'utiliser une méthode k-pas : on ne fait le test d'arrêt que toutes les $k > 1$ itérations. Soit $n_1 = qk, q \in \mathbb{N}^*$ l'itération qui assure la terminaison. Afin de déterminer $n_* \in](q-1)k, qk]$ on repart de l'itération $(q-1)k$ en ré-effectuant chaque itération et le test à chaque fois. On aboutit ainsi à n_* au pire en $k+q-1$ synchronisations et après $kq+k-1 = \mathcal{O}(n_*+k)$ itérations.

4.3.2.3 Contrôle amorti

Une autre idée est de faire évoluer le nombre d'itérations que l'on passe sans effectuer de synchronisation.

Dans toute cette section la fonction $\log$ désigne le logarithme en base 2. Supposons que l'on effectue un test d'arrêt aux itérations $2^0, 2^1, \ldots, 2^{k_1}, k_1 > 0$, le test étant vérifié pour la première fois à $2^{k_1}, k_1 = \lceil \log n_1 \rceil$. On a alors $2^{k_1-1} < n_* \leq 2^{k_1} \stackrel{\text{def}}{=} n_1$. On peut réeffectuer le même procédé en partant de $2^{k_1-1} = n_1/2$: on effectue les tests d'arrêt aux itérations $\frac{n_1}{2} + 2^0, \frac{n_1}{2} + 2^1, \ldots, \frac{n_1}{2} + 2^{k_2}, k_2 \leq \lceil \log(n_1/2) \rceil$ tel que

$$\frac{n_1}{2} < \frac{n_1}{2} + 2^{k_2-1} \leq n_* \leq \frac{n_1}{2} + 2^{k_2} < n_1.$$

On pose alors $n_2 \stackrel{\text{def}}{=} 2^{k_2}$ et on itère le procédé à nouveau à partir de $\frac{n_1}{2} + \frac{n_2}{2}$. On calcule ainsi $k_l, l > 0$ par récurrence, tel que

$$\sum_{i=1}^{l-1} \frac{n_i}{2} + 2^{k_l-1} < n_* \leq \sum_{i=1}^{l-1} \frac{n_i}{2} + 2^{k_l}.$$

On pose $n_l \stackrel{\text{def}}{=} 2^{k_l}$.

Cela revient à effectuer la décomposition binaire de n_1.

Le nombre total de synchronisations effectuées $\mathcal{S}$ sera au pire égal à $k_1 + k_2 + \ldots k_l$. En notant $n_1 = 2^{k_1}$, il vient :

$$\mathcal{S} = \sum_{i=0}^{l} \log\left(\frac{n_1}{2^i}\right).$$

Cette somme vaut

$$\mathcal{S} = \frac{(\log n_1 - 1)\log n_1}{2} = \mathcal{O}(\log^2 n_1).$$

	Itérations	Synchronisations
k-pas	$< k(q+1)$	$< k+q$
amorti	$4n^*$	$\log^2(n^*)$
κ-amorti	$< q(k+4)$	$< q+\log^2(q)$

TAB. 4.1 *Coûts des 3 algorithmes de contrôle des itérations.*

Comme de plus $n_1/2 < n_* \leq n_1$, il vient que $n_* \leq n_1 < 2n_*$ et donc

$$\mathcal{S} = \mathcal{O}(\log^2 n_*).$$

Le nombre d'itérations calculées $\mathcal{N}$ pour aboutir à l'itération n_* est au pire égal à

$$\mathcal{N} = \sum_{i=0}^{l} \frac{n_1}{2^i} = 2n_1 - 1,$$

En utilisant la même majoration que ci-dessus ($n_1 < 2n_*$), on aboutit au résultat :

$$\mathcal{N} \leq 4n_* - 1.$$

4.3.2.4 Contrôle κ-amorti

Enfin, on peut marier les approches $k-$pas et amorties en effectuant des synchronisations de contrôle tous les k pas jusqu'à $n_1 = qk$ comme en 4.3.2.2 afin de pallier le défaut de l'approche amortie qui est de faire des "petits pas" au départ des itérations. Une fois n_1 atteint on utilise l'algorithme amorti à partir de $(q-1)k$ pour converger vers $n^* = qk + r, r < q$ en $4r$ itérations et $\log^2(r)$ synchronisations.

On aboutit à un algorithme κ-amorti qui demande $qk + 4r < q(k+4)$ itérations et $q + \log^2(n^* - qk) < q + \log^2(q)$ synchronisations.

En bilan de ces 3 algorithmes, on peut dresser le tableau 4.1.

Comme $n^* = kq + r, r < q$, on a un premier algorithme 4 fois meilleur (en ordre de grandeur) pour le nombre d'itérations que l'algorithme amorti. La différence de coût en terme de nombre de synchronisations est moins évident puisque $k+q$ peut être vu comme de l'ordre de grandeur de $2\sqrt{n^*}$, à comparer à $\log^2(n^*)$. Asymptotiquement l'algorithme k-pas est donc moins performant que l'algorithme amorti pour le nombre de synchronisations.

L'algorithme κ-amorti, compromis entre les deux, conserve un nombre d'itération de l'ordre de $n^* + \sqrt{n^*}$, proche de n^* avec un nombre de synchronisations plus faible que celui de l'algorithme amorti.

4.4 Conclusion

En conclusion de ce chapitre et de cette première partie, le graphe de flot de données apparaît comme un modèle de programmation parallèle qui permet de décrire des algorithmes du type de ceux requis par la mécanique ou la chimie quantique. L'environnement ATHAPASCAN permet une implantation de ce modèle de programmation alternatif au passage de messages ou aux directives de compilation type Open-MP. De plus le GFD, par une description fine de l'exécution (éventuellement) parallèle permet de pallier les limitations montrées pour les environnements classiques, que sont le manque de portabilité d'une architecture parallèle à une autre et la difficulté de garantir des performances. Nous avons également présenté dans ce chapitre la façon dont ATHAPASCAN nous permet de programmer l'algorithme IRA, qui sert à résoudre BQ. Enfin nous avons apporté une contribution sur des façons d'envisager de façon asynchrone le contrôle d'une application fortement synchrone, du genre de IRA.

La partie suivante propose une analyse théorique et des résultats expérimentaux sur les programmes ATHAPASCAN introduits dans cette première partie.

II

Contrôle de l’exécution du programme

Nous avons montré comment des calculs types de mécanique et chimie quantique — BQ et MP2 — peuvent être programmés en parallèle, et le type d'outils que les environnements classiques proposent, leurs atouts et leurs limitations. Le modèle de la programmation par description du flot de données présenté dans le chapitre précédent permet de traiter certaines de ces limitations. Le programmeur peut, grâce à un environnement comme ATHAPASCAN , décrire son algorithme sous forme de tâches parallèles qui accèdent des données selon un typage précis. Cette description à son tour permet de calculer le GFD. À partir du GFD, on peut :

- utiliser des modèles de machine différents, puisqu'il en est complètement indépendant en ne décrivant que l'algorithme ;
- effectuer des calculs d'ordonnancement selon un modèle de machine et selon les critères d'efficacité du programmeur.

Cette seconde partie traite de ces deux derniers problèmes.

Une fois le graphe de flot de données spécifié par le programmeur il est nécessaire de l'ordonnancer, c'est-à-dire d'attribuer à chaque tâche et donnée un site et une date d'exécution. Cela doit être fait en fonction d'un critère à optimiser. Le premier de ces critères est souvent le temps d'exécution que l'on cherche à minimiser. Le type d'algorithme que l'on peut employer dans ce but dépend à la fois du modèle de machine que l'on emploie (avec ou sans réseau de communication par exemple) et de la connaissance sur le graphe : on peut le connaitre dès la compilation (cas statique) ou au contraire, ce qui est plus général, le découvrir au fur et à mesure de l'exécution du programme. Les algorithmes de liste sont alors le plus souvent employés. Le chapitre 5 présente ces algorithmes et les résultats qu'ils permettent d'obtenir sur BQ.

Mais sur beaucoup d'applications "réalistes" le recours au parallélisme est requis autant par le gain de mémoire que par la diminution du temps de calcul (étant donnée la puissance des processeurs actuels c'est de plus en plus vrai). C'est en particulier le cas de MP2. Un autre objectif de l'ordonnancement peut donc être de garder un temps d'exécution performant tout en garantissant l'utilisation d'un espace mémoire "raisonnable", sans trop de surcoût dû à la gestion du parallélisme ou à la concurrence des allocations mémoire. Pour tenir compte de cet aspect mémoire, il faut annoter le GFD avec les quantités de mémoire allouées et libérées par chaque tâche. On obtient alors un outil sur lesquels des algorithmes d'ordonnancement permettent de garantir de bonnes performances. C'est le cadre du chapitre 6.

5

Contrôle du temps d’exécution et problème BQ

La notion de temps d’exécution est liée au modèle de programmation d’une part et au modèle de machine parallèle d’autre part. Ce chapitre se place dans le cadre du GFD (*cf.* chapitre 4) et décrit plusieurs bornes obtenues avec divers modèles de machine et différents ordonnancements. La première section rappelle rapidement quelques modèles de machines parallèles les plus utilisés en algorithmique. Pour ordonnancer des algorithmes sur ces modèles de machine on distingue le cas statique où le GFD est connu dès la compilation, du cas dynamique *a priori* plus difficile où rien n’est connu avant l’exécution du programme. La section 5.2 commence par ce dernier en présentant les algorithmes de type "liste" souvent les plus performants. La borne de Graham est rappelée sur un modèle de SMP, et une extension au cas d’un modèle distribué simple est ensuite introduite. La section 5.3 aborde le cas statique, où l’on peut comparer les performances à celles d’un programme de type MPI ou Open-MP sur les ordonnancements statiques "simples" (cycliques, par exemple) ; ou bien appliquer des algorithmes d’ordonnancement prenant en compte plus finement le modèle de machine, qui sont à nouveau de type "liste". Enfin le cas de l’ordonnancement de l’algorithme itératif IRAM utilisé pour résoudre BQ est présenté dans la dernière section.

5.1 Modèles de machine parallèle pour l'ordonnancement

Aborder l'analyse de l'exécution d'un programme parallèle sur une machine réelle est une tâche très difficile voire impossible de par la complexité du comportement des machines actuelles. Ceci est encore plus vrai pour les machines parallèles et distribuées.

La solution classique est alors de recourir à un modèle d'exécution [44]. Un tel modèle définit une machine abstraite ainsi que le comportement de cette machine lors de l'exécution d'un programme. Le modèle d'exécution doit être suffisamment simple pour rendre abordable l'analyse théorique d'une exécution. Il doit également être réaliste pour rendre les prédictions aussi proches que possible d'une exécution réelle. Il doit de plus représenter une large classe de machines pour rendre l'analyse théorique portable. Ces trois caractéristiques sont contradictoires et définir un modèle d'exécution passe par des compromis.

Plusieurs modèles d'exécution ont ainsi été définis pour représenter l'exécution d'un programme sur une machine parallèle. Ils peuvent être séparés en deux classes : les modèles à mémoire partagée et les modèles à mémoire distribuée.

5.1.1 Modèles d'exécution à mémoire partagée

Dans ces modèles, les processeurs communiquent entre eux par l'intermédiaire d'une mémoire partagée. Ces modèles peuvent représenter les machines à mémoire partagée (SMP) mais également les machines à mémoire distribuée dans lesquelles la mémoire est virtuellement partagée.

Le plus connu de ces modèles est le modèle PRAM (*Parallel Random Access Machine*) [21]. Une machine PRAM est composée de p processeurs et d'une mémoire partagée. Les processeurs sont synchronisés entre eux. À chaque top d'exécution, chaque processeur peut lire deux données, réaliser une opération et écrire une donnée. Ainsi, une opération arithmétique élémentaire peut être réalisée à chaque top d'exécution. Plusieurs variantes et extensions de ce modèle ont été proposées permettant de prendre en compte les conflits d'accès à la mémoire partagée. La latence et les délais d'accès à cette mémoire peuvent être considérés : par exemple le modèle Local-PRAM [38] associe à chaque processeur une mémoire locale.

5.1.2 Modèles d'exécution à mémoire distribuée

Dans ces modèles, les processeurs communiquent entre eux par l'intermédiaire d'un réseau de communication. Ces modèles d'exécution cherchent alors principalement à mo-

déliser les communications entre processeurs. On parle souvent de modèle de communication plutôt que de modèle d'exécution.

Le modèle délai est, jusqu'à récemment, le modèle de communication le plus utilisé pour le problème de l'ordonnancement d'applications représentées par un graphe de tâches [55, 52, 30, 68]. Dans ce modèle le réseau de communication est caractérisé par une latence ou délai, correspondant au temps τ nécessaire à la transition d'un mot entre deux processeurs. Nous utiliserons principalement dans ce chapitre le modèle de Hwang *et al* [30] dans lequel le temps de communication d'un message de taille n entre deux processeurs prend τn unités de temps.

Ce modèle délai est très imparfait puisqu'il considère une bande passante infinie en entrée et en sortie de chaque processeur : un processeur peut en effet envoyer ou recevoir un nombre illimité de messages à un instant donné ; de plus il néglige complètement la latence du réseau, pourtant importante dans la plupart des machines actuelles. Pour remédier à ces problèmes, des modèles plus proches des machines réelles ont été proposés. On peut citer parmi ceux-ci le modèle BSP proposé par Valiant [64] et le modèle LogP proposé par Culler *et al.* [15, 14]. Cependant, ces modèles sont beaucoup plus complexes à utiliser que les modèles délais, tant du point de vue de la construction d'algorithmes d'ordonnancement statiques que pour l'analyse théorique d'une exécution. C'est pourquoi très peu d'algorithmes d'ordonnancement ont été proposés sur le modèle LogP [47, 9, 37] ; de plus, la plupart de ces algorithmes sont restreints à des classes de graphes de tâches spécifiques comme les arbres, ce qui limite leur intérêt pour des applications réelles.

5.2 Ordonnancement dynamique du graphe

Dans le cas de la régulation dynamique de la charge, l'ordonnancement le plus rencontré en pratique consiste basiquement, lorsqu'un processeur devient inactif, à lui allouer une tâche prête s'il en existe. Un tel ordonnancement est appelé "algorithme de liste" : il requiert de gérer une structure évoluant dynamiquement et permettant d'accéder aux tâches prêtes.

Dans cette section, seul le ratio de performance (rapport entre le temps d'exécution fourni par l'algorithme ordonnancé et le temps d'exécution optimal de l'algorithme) d'un tel algorithme pour un GFD quelconque est étudié. Le surcoût induit par sa mise en œuvre sur une architecture distribuée (gestion du graphe et contrôle de l'ordonnancement d'une part, introduction des communications d'autre part) est étudié dans la section 5.2.2.

5.2.1 Cas des communications négligeables — modèle PRAM

Théorème 1 *[26] Si les communications ne sont pas prises en compte, tout algorithme d'ordonnancement de type liste a un ratio de performance borné par* $\left(2 - \frac{1}{m}\right)$ *sur une machine constituée de* m *processeurs identiques.*

Nous rappelons la preuve de ce résultat car son schéma intervient dans de nombreuses démonstrations.
Soit T_m la longueur de l'ordonnancement fourni par l'algorithme de liste sur m machines. Un algorithme de liste est tel que, à chaque instant, au moins un processeur exécute une tâche. Ainsi, si à un instant donné un processeur est inactif alors il existe au moins un processeur qui exécute une tâche. Soit t_{j_1} l'une des tâches qui s'est terminée à la date T_m et soit d_{j_1} la date de début d'exécution de t_{j_1}. Deux cas peuvent être distingués :

cas 1 : soit aucun processeur n'a été inactif avant d_{j_1}.

cas 2 : soit il existe au moins un processeur inactif à un certain instant avant d_{j_1}. Soit θ la plus grande date avant d_{j_1} à laquelle au moins un processeur est inactif. À θ, t_{j_1} n'est pas prête car sinon elle aurait été affectée à un processeur inactif. Donc il existe une tâche t_{j_2} telle que t_{j_2} est en cours d'exécution à θ et $t_{j_2} \prec t_{j_1}$. Soit d_{j_2} la date de début d'exécution de t_{j_2}.

En appliquant récursivement ce schéma jusqu'à ce que le cas 1 arrive, nous construisons une séquence de tâches $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$ telle que, à tout instant, soit tous les processeurs sont actifs soit un processeur exécute une tâche t_{j_i} ($1 \leq i \leq k$).

Soit T_∞ le temps minimal sur un nombre infini de processeurs et T_1 le nombre total d'opérations exécutées (temps sur un processeur séquentiel, appelé aussi travail). Le temps total d'inactivité $\#I$ est défini par $\#I = mT_m - T_1$. Pour $1 \leq i \leq k$, soit l_i la durée de la tâche t_{j_i}. Nous avons $\#I \leq (m-1)\sum_{i=1}^{k} l_i$, d'où :

$$mT_m \leq T_1 + (m-1)\sum_{i=1}^{k} l_i.$$

En outre, les tâches t_{j_i}, $1 \leq i \leq k$ étant sur un chemin critique, $\sum_{j=1}^{k} l_i \leq T_\infty$. On a finalement :

$$T_m \leq \frac{T_1}{m} + \left(1 - \frac{1}{m}\right) T_\infty \tag{5.1}$$

Par ailleurs, soit $T_m^\star$ la longueur de l'ordonnancement optimal. On a $T_\infty \leq T_m^\star$ et, comme T_1 opérations doivent être exécutées par tout ordonnancement, $mT_m^\star \geq T_1$. En remplaçant dans (5.1), on obtient : $T_m \leq \left(2 - \frac{1}{m}\right) T_m^\star$. □

On montre également qu'ajouter des informations sur les tâches, comme la durée de leur exécution, ne permet pas d'améliorer le ratio de performance sauf à restreindre le type de GFD traités (tâches indépendantes par exemple).

5.2.2 Prise en compte des communications : modèle délai

Dans le cadre d'une machine distribuée, on utilise un modèle prenant en compte les communications. Nous montrons ici comment le modèle délai permet des ordonnancements dynamiques avec un ratio borné. Nous commençons tout d'abord par justifier l'utilisation de ce modèle de machine en exposant comment on peut simuler une mémoire partagée avec accès à un mot de la mémoire en temps h, sur une machine à mémoire distribuée.

5.2.2.1 Simulation d'une machine à mémoire partagée

Pour prendre en compte le coût des communications (contention et latence), nous simulons une mémoire partagée. La simulation utilise des fonctions de hachage (choisies aléatoirement dans une classe de fonctions de hachage universelles) pour distribuer les objets accessibles par tous les processeurs (i.e. les transitions dans le graphe de dépendance) sur les différents modules mémoire [54, 38].

Le délai d'une simulation est le temps requis pour l'accès à une donnée de taille unité. Il est lié à l'évaluation de la fonction de hachage, à la contention mémoire (quand plusieurs accès sont effectués vers un même module) et à la latence (temps de routage). Dans [54], une simulation de délai $\Theta(\log p)$ sur un réseau "butterfly" est donnée pour des accès exclusifs (de type EREW). Dans [35], une simulation de délai $O(\log\log p \log^\star p)$ est donnée pour des accès aléatoires (de type EREW ou CRCW) sur une architecture disposant d'un réseau d'inteconnexion complet (la latence n'est pas prise en compte) ; pour des accès concurrents (CRCW), la simulation est à un facteur $\log^\star p$ de l'optimal.

Pour obtenir une simulation optimale, ces délais doivent être masqués par des calculs. La technique classique (virtualisation ou "parallel slackness" [38, 65, 35]) consiste à simuler q ($q > p$) processeurs virtuels sur les p processeurs de l'architecture physique, en utilisant généralement des processus légers. La simulation est alors dite optimale si le délai pour un accès est proportionnel à p/q.

Dans la suite de cette section, nous supposerons disposer d'une simulation optimale de délai h ; le nombre de processeurs m requis pour une telle simulation peut alors être supérieur au nombre de processeurs p de l'architecture. L'ordonnancement sera donc effectué sur m processeurs et non p, mais comparé à l'ordonnancement optimal sur p processeurs. Un processeur (parmi les m) peut lire ou écrire un objet dans n'importe quel module mémoire distant. Le coût de l'accès à une donnée de taille n est supposé borné par $h.n$ où h est une constante de la machine.

Synchronisation. Les accès en mémoire distante ne permettent pas directement de synchroniser les processeurs (à la différence des communications). Sur une architecture distribuée asynchrone, des outils spécifiques de synchronisation (verrous, sémaphores)

doivent alors être utilisés. Nous supposerons aussi que le coût d'accès à un verrou global, lorsque celui-ci est libre, est lui aussi borné par h (il requiert un aller-retour).

5.2.2.2 Ordonnancement dynamique avec communications

Dans le cadre du modèle délai, chaque mot mémoire étant supposé accessible en temps h, on peut adapter la preuve du ratio de l'algorithme glouton de Graham pour obtenir une borne sur l'efficacité d'un algorithme de type liste en environnement parallèle.

Néanmoins la gestion d'une liste de tâches prêtes n'est pas aussi simple en parallèle qu'en séquentiel car il faut gérer les accès concurrents dans une implémentation distribuée. Dans ce qui suit, nous considérons dans un but simplificateur une gestion séquentielle de la structure permettant de calculer les tâches prêtes (i.e. la liste) et de les affecter aux processeurs inactifs. À défaut d'être la plus performante sur un grand nombre de processeurs, elle permet d'estimer précisément le surcoût lié à la gestion de l'ordonnancement. Pour éviter de dédier un processeur à la gestion de cette structure, elle est stockée en mémoire (virtuellement) partagée et accédée par les processeurs en exclusion mutuelle ; l'exclusion est réalisée par un verrou global.

Théorème 2 *Soit un programme parallèle dont l'exécution génère n tâches et n_d dépendances (transitions). Soit T_1 le temps d'une exécution séquentielle (sur un processeur) et T_∞ le temps minimal sur un nombre infini de processeurs. Alors, dans le cadre d'un modèle délai de communication, G peut être ordonnancé sur m processeurs identiques en temps T_m borné par :*

$$T_m \leq \frac{T_1}{m} + T_\infty + O(n + n_d + m).$$

Comme pour le théorème 1, la preuve est basée sur la construction d'un ensemble de tâches situées sur un chemin critique dans G. Pour contrôler l'ordonnancement, nous utilisons une structure globale, visible par tous les processeurs et accédée en exclusion mutuelle grâce à un verrou global `verrou-graphe` ; outre le graphe G, cette structure contient :

- la liste L_t des tâches prêtes (initialisée avec les tâches de G de degré entrant nul) ;
- la liste L_i des processeurs inactifs ; à l'initialisation, seul un processeur (disons P_0) est supposé artficiellement actif – il simule la terminaison d'une tâche fictive –, les autres $m - 1$ processeurs étant mis dans L_i.

L'ordonnancement des tâches est géré de la façon suivante : lorsqu'un processeur p termine une tâche t (par exemple P_0 au top 0), il exécute les instructions de *post-traitement* liées à t suivantes :

(i). Prendre le verrou `verrou-graphe`.

(ii). Mettre à jour G en ajoutant les nouvelles tâches créées et les dépendances associées.

(iii). Mettre à jour la liste L_t des tâches prêtes en ajoutant les tâches successeurs immédiats de t dont tous les prédécesseurs sont terminés.

(iv). Tant que ($L_i \neq$ vide) et ($L_t \neq$ vide) faire
Enlever une tâche t' de L_t et un processeur p' de L_i ;
Lancer l'exécution de t' sur p'.

(v). Si ($L_t \neq$ vide) alors
Enlever une tâche t'
Relâcher le verrou `verrou-graphe` ;
Exécuter t' (sur le processeur p)

(vi). sinon
Ajouter p à la liste des processeurs inactifs
Relâcher le verrou `verrou-graphe` ;

De manière évidente, l'ordonnancement ainsi construit est admissible, i.e. ne viole aucune contrainte de précédence dans G. Soit T_m le temps d'exécution avec cet ordonnancement sur m machines. En reprenant le schéma de la preuve précédente (théorème 1), il s'agit de borner les tops d'inactivité *potentielle*. Les tops $\{1, ..., T_m\}$ sont partitionnés en trois sous-ensembles, A (activité), O (ordonnancement) et I (inactivité) de la façon suivante :

- à tout top dans A, tous les processeurs exécutent une tâche, i.e. une instruction de l'application.
- à tout top dans O, une instruction de post-traitement (i.e. gestion de l'ordonnancement) est en cours.
- à tout top dans I, la liste L_i des processeurs inactifs est non vide mais aucune instruction de post-traitement n'est en cours d'exécution.

En remarquant que, lorsqu'un processeur est en attente du verrou, une instruction de post-traitement est nécessairement en cours sur un autre processeur, il n'y a pas dans $A \bigcup O$ de tops où un processeur est en attente du verrou.

$\#A$ peut être trivialement borné par $\frac{T_1}{m}$. L'exécution en exclusion mutuelle des instructions de post-traitement permet de borner facilement $\#O$. En effet, une fois le verrou pris, le post-traitement ne consiste qu'en la mise à jour des listes (les opérations de base sont en coût constant) et au lancement de nouvelles exécutions éventuelles. Lorsque le verrou est libre, le temps nécessaire à sa prise est borné par h. Lorsqu'il n'est pas libre, il existe nécessairement un autre processeur en train d'exécuter une instruction de post-traitement (éventuellement en train de prendre ou de relâcher le verrou). Par suite, le nombre de tops dans $\#O$ est borné par $O(h(n + n_d + m))$ (i.e. la taille du graphe).

$\#I$ est borné de manière analogue à la preuve de Graham, en remarquant que lorsqu'un processeur est inactif et aucun en cours de post-traitement, il existe nécessairement une tâche sur un chemin critique qui est en cours d'exécution. Par suite, $\#I < T_\infty$.

Pour conclure, $mT_m = m\#O + m\#A + (m-1)\#I$. D'où[1] $T_m \leq \frac{T_1}{m} + T_\infty + O(n + n_d + m)$. □

[1]Le ratio $(2 - 1/m)$ de Graham se retrouve directement dans le cas où $\#O = 0$.

Implémentation d'un algorithme de liste. En pratique, la gestion centralisée des listes et l'accès en exclusion mutuelle est évité. D'une part, le graphe est stocké de manière distribuée ; d'autre part, lorsqu'un processeur est inactif, il scrute les autres processeurs pour trouver une nouvelle tâche à exécuter (on parle de "vol" de tâche). Une telle stratégie est analysée en moyenne d'un point de vue théorique dans [5, 33] et permet d'obtenir un résultat similaire au théorème 2.

5.3 Connaissance statique du graphe

Dans le cas où l'on connaît le GFD à la compilation on peut, dans le cadre de l'utilisation du GFD comme modèle de programmation, retrouver le type de programmation "MPI" où le programmeur code lui-même l'ordonnancement à travers son utilisation de la localité des données et de l'ordre des calculs. On peut aussi calculer des ordonnancements prouvés efficaces, et qui éventuellement utilisent des modèles de machine non-triviaux.

5.3.1 Ordonnancement "trivial" et programmation "MPI"

On peut dans le cas où on connaît statiquement le GFD obtenir un code dont la performance est équivalente à un programme MPI. La programmation par échanges de messages implique que le programmeur explicite dans son code :

– le site d'exécution de chaque instruction ;
– la date d'exécution au plus tôt de chaque instruction.

Typiquement on programme une distribution cyclique des calculs ou encore un schéma de type "ferme de tâches". (C'est le cas par exemple de nombre d'applications d'algèbre linéaire comme le benchmark Linpack ; les programmes de mécanique ou chimie quantique basés sur MPI, présentés dans le chapitre 3, recourent à ce type de programmation.) L'efficacité de l'ordonnancement est justifié empiriquement ou grâce aux caractéristiques connues de l'application et de son GFD. L'avantage de cette programmation est également de ne pas imposer au programme le surcoût du calcul de l'ordonnancement, puisque celui-ci est "codé en dur" dans le programme même. Ne subsiste que le coût éventuel de la mise en oeuvre de l'algorithme d'ordonnancement. Notons que les environnements de programmation comme Open-MP ne proposent également comme ordonnancement que des stratégies cycliques ou de "ferme de processeurs".

Un environnement comme ATHAPASCAN autorise également l'utilisation d'informations statiques sur le GFD de l'algorithme pour l'ordonnancer. On retrouve alors des performances tout à fait comparables à celles d'un programme Open-MP ou multi-threadé (pour SMP) ou d'un programme MPI (pour machine distribuée). Le surcoût, en général faible, de l'ordonnancement est le même dans les 3 environnements parallèles. La figure (5.1) page 97, tirée de [18], présente les courbes de performance comparées entre une factorisation LL^t programmée en ATHAPASCAN et le même algorithme de la librairie

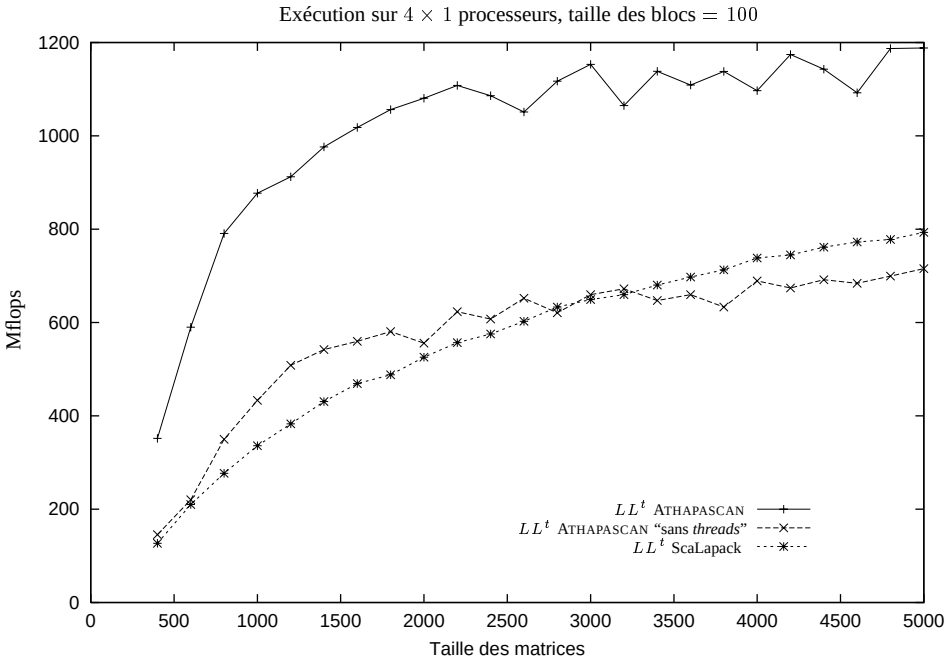


Figure 5.1 *Comparaison* ATHAPASCAN */ScaLapack sur une machine SMP (factorisation LL^t).*

ScaLapack. Les deux algorithmes ATHAPASCAN et ScaLapack sont identiques : La factorisation est réalisée sans pivotage, la matrice est partitionnée en blocs de même taille 100 et le même placement cyclique bidimensionel des blocs sur les nœuds est effectué. Les exécutions ont été effectuées sur un nœud SMP de la même machine, une station SUN sous Solaris 7, (localisé à l'Université du Delaware), chacune contenant quatre processeurs Ultra Sparc II cadencé à 250 MHz et 512 méga octets de mémoire. La puissance de référence d'un processeur de cette machine est de 390 Mflops. La courbe "LL^t ATHAPASCAN " a été obtenue en utilisant ATHAPASCAN avec un processus UNIX et quatre processus légers, la courbe "LL^t ATHAPASCAN sans *threads*" a été obtenue en utilisant ATHAPASCAN avec quatre processus UNIX et la courbe "LL^t ScaLapack" a été obtenue en utilisant Scalapack et quatre processus UNIX. On remarque que la version d'ATHAPASCAN qui n'emploie pas les threads et est donc la plus comparable au programme Scalapack a des performances tout à fait équivalentes ; de plus, l'utilisation de processus légers plutôt que des processus UNIX pour exploiter un nœud SMP améliore jusqu'à deux fois les performances obtenues.

L'utilisation du GFD n'impose donc pas de surcoût par rapport à une programmation "statique" usuelle de l'ordonnancement. Par contre le modèle ATHAPASCAN permet de découpler l'expression de l'algorithme de son ordonnancement (Open-MP permet aussi cela). L'utilisateur gagne donc en facilité de programmation ; et il peut également employer des outils d'ordonnancement plus élaborés qui tireront tout leur profit des informations du GFD.

5.3.2 ETF/ERT : ordonnancement statique glouton avec prise en compte des communications

5.3.2.1 Performance des ordonnancements statiques sur modèle délai

Les algorithmes théoriques d'ordonnancement les plus classiquement utilisés pour prendre en compte des communications sont ETF *Earliest Task First* ou ERT *Earliest Ready First* [29]. Ces deux ordonnancements sont de type liste. Ils consistent à ordonnancer la tâche qui pourra commencer son exécution au plus tôt (en prenant en compte les délais éventuels de communication) sur l'un des processeurs inactifs. Soit $P = \{P_1, \ldots, P_p\}$ les p processeurs. L'algorithme évalue donc, pour chaque couple $(r_i, p_j) \in R \times P$, la date de démarrage de la tâche r_i sur le processeur p_j, en considérant les délais de communication ainsi que la date de disponibilité de p_j. Soit (r_i, p_j) un couple associé à la plus petite date de démarrage ; la tâche r_i est alors ordonnancée sur le processeur p_j.

Le théorème suivant évalue la performance d'un tel ordonnancement sur le modèle délai proposé dans [29] : le temps de communication de n mots entre deux processeurs est supposé prendre τn unités de temps ; on note C_{max} le maximum, sur tous les chemins du graphe, du volume de communication cumulé sur un chemin.

Théorème 3 *[43] Si l'on néglige le coût de calcul et de réalisation de l'ordonnancement, tout ordonnancement de type liste assignant une tâche prête sur le processeur qui démarrera son exécution au plus tôt conduit à un temps d'exécution T_p majoré par :*

$$T_p \leq \frac{T_1}{p} + (1 - \frac{1}{p})T_\infty + \tau C_{max}.$$

Il est à noter que la majoration utilise le fait que le graphe de flots de données est entièrement connu, ainsi que les durées des tâches et les volumes de communication. Cependant, ces algorithmes peuvent aussi être utilisés à la volée ; dans ce cas, Doreille établit une majoration similaire : $T_p \leq \frac{T_1}{p} + (1 - \frac{1}{p})T_\infty + \tau C_{max^\star}$ [18] ($C_{max^\star}$ est le volume maximum de communications sur un chemin du graphe légèrement modifié : pour chaque tâche le volume de données entrant par un arc est aligné sur le plus grand volume de données entrant dans cette tâche). Par ailleurs, le coût du calcul de l'ordonnancement est $O(pn^2)$ [29].

L'environnement ATHAPASCAN permet d'utiliser ce type d'ordonnancement *à l'exécution* pour la classe des programmes où seule la tâche racine crée d'autres tâches (les autres créations de tâches sont alors dégénérées en exécution séquentielle). Un ordonnancement statique peut alors être calculé à partir du graphe généré après l'exécution de la tâche racine. Cet ordonnancement fournit un placement et un ordre d'exécution de toutes les tâches du graphe. Ainsi le placement des tâches peut être connu par tous les processeurs en début d'exécution.

Ce modèle d'exécution est surtout adapté aux applications pour lesquelles le graphe de flot de données (et donc le parallélisme de l'application) peut être décrit de manière

fine à partir de l'exécution d'une petite partie du programme sur les données en entrée. Notons cependant qu'il peut être facilement étendu aux applications dans lesquelles le parallélisme est décrit par phases (succession de phases de description du parallélisme séparées par des phases d'exécution).

L'exécution du programme est alors divisée en quatre étapes distinctes :

(i). Exécution de la tâche racine t_1, qui va générer le graphe de flot de données comme décrit dans la section.

(ii). Calcul d'un ordonnancement statique de ce graphe G, fournissant un placement des tâches sur les m processeurs et un temps d'exécution du graphe $\omega_m(G)$.

(iii). Diffusion de ce graphe et du placement des tâches sur les processeurs. Une copie du graphe est alors présente sur chaque processeur.

(iv). Interprétation du graphe et exécution des tâches de ce graphe.

La dernière étape utilise activement le fait que le graphe est dupliqué sur chaque processeur. La figure 5.2 montre un exemple simple de l'exécution sur deux processeurs d'un graphe contenant 5 tâches et un objet partagé. Sur le processeur 1, la tâche t_1 est prête mais comme elle a été placée sur le processeur 2, elle est retirée du graphe sans être exécutée. Le flot de données arrive alors sur le nœud a_2 qui attend la version de l'objet modifiée par t_1 avant de devenir prêt. Sur le processeur 2, la tâche t_1 est exécutée puis retirée du graphe. Le flot de données arrive alors sur le nœud a_2 qui devient prêt. Il est alors possible en regardant les sites d'exécution des tâches branchées sur a_2 de voir que le processeur 1 a besoin de la version de l'objet partagé pour exécuter t_3. Le processeur 2 peut alors immédiatement après la terminaison de t_1 envoyer les valeurs de la version de l'objet sur le processeur 1. Le même mécanisme est appliqué lors de la terminaison de t_4. Le processeur 1 peut alors immédiatement envoyer au processeur 2 la valeur de la version de l'objet.

5.3.2.2 Mesures obtenues sur la factorisation de Choleski

Mathias Doreille [18] a mesuré les performances obtenues sur la factorisation LL^t en environnement distribué (sur IBM SP1). La figure (5.3) page 100 présente les courbes de performance comparées entre une factorisation LL^t programmée en ATHAPASCAN et le même algorithme de la librairie ScaLapack.

L'algorithme d'ordonnancement ici utilisé est un placement cyclique bidimensionnel. On retrouve le bon comportement du programme ATHAPASCAN par rapport à ScaLapack déjà obtenu sur SMP (*cf.* figure (5.1)). Les ordonnancements ETF et ERT ont également été testés et ETF se révèle légèrement meilleur que le placement cyclique.

L'utilisation de ces algorithmes est ainsi validée expérimentalement dans l'environnement ATHAPASCAN . La section suivante présente l'accélération obtenue par le programme ATHAPASCAN implémentant l'algorithme IRA, pour des exécutions sur architecture à mémoire partagée.

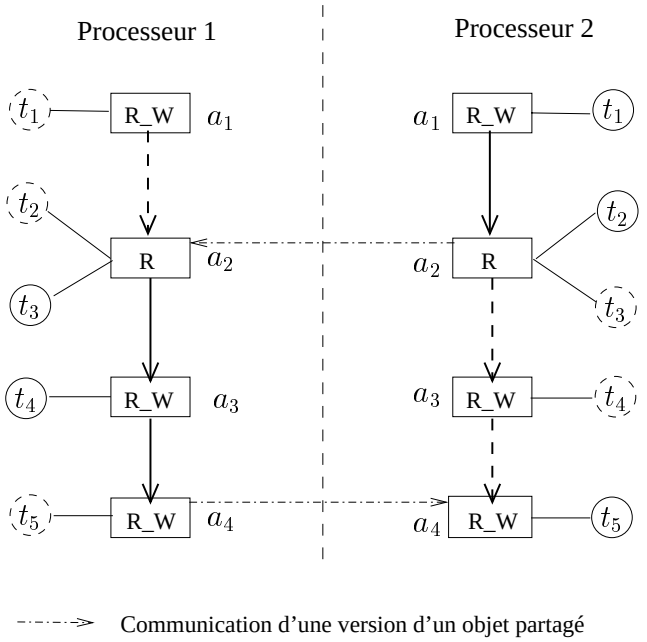


Figure 5.2 *Exécution d'un graphe sur deux processeurs. Les cercles en pointillés représentent les tâches qui ont été placées sur un autre processeur.*

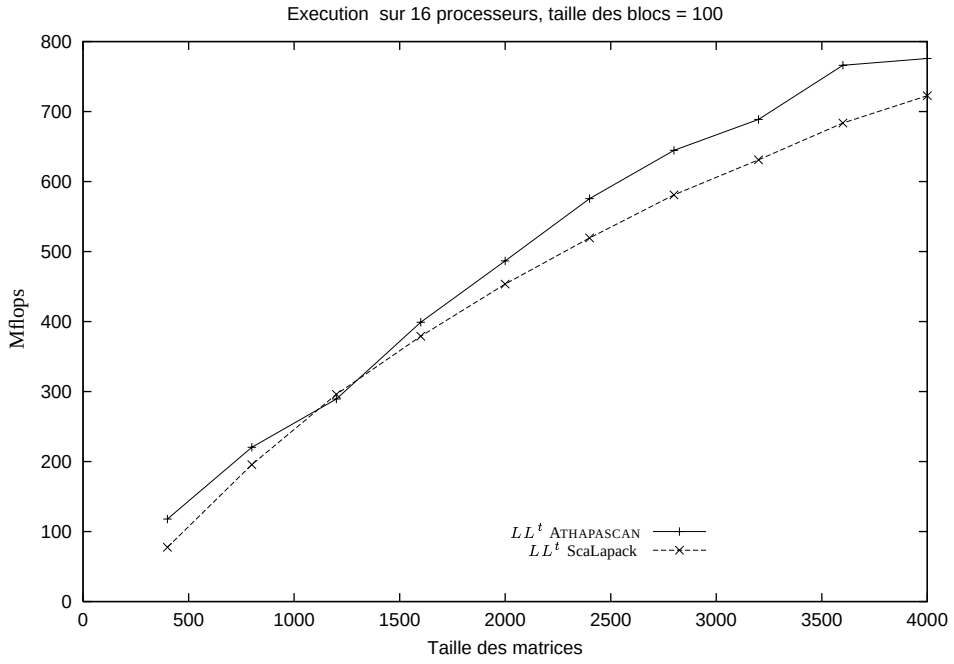


Figure 5.3 *Comparaison* ATHAPASCAN */ScaLapack sur une machine distribuée (factorisation LL^t).*

5.4 Application : BQ en ATHAPASCAN sur SMP

Le programme ATHAPASCAN présenté dans la section 4.3.1 du chapitre 4 peut bien-sûr être ordonnancé selon les techniques présentées dans ce chapitre.

La figure (5.4) page 101 présente l'accélération obtenue sur le programme ATHAPASCAN de calcul de valeurs propres par l'algorithme IRA (*cf.* section 4.3.1), sur une machine SMP à 4 processeurs. Chaque nœud est un Pentium-Pro, cadencé à 200 MHz et disposant de 256 Mo de mémoire. Cette architecture, certes déjà âgée pour le calcul intensif, sert néanmoins à illustrer les performances que le programme ATHAPASCAN permet d'envisager.

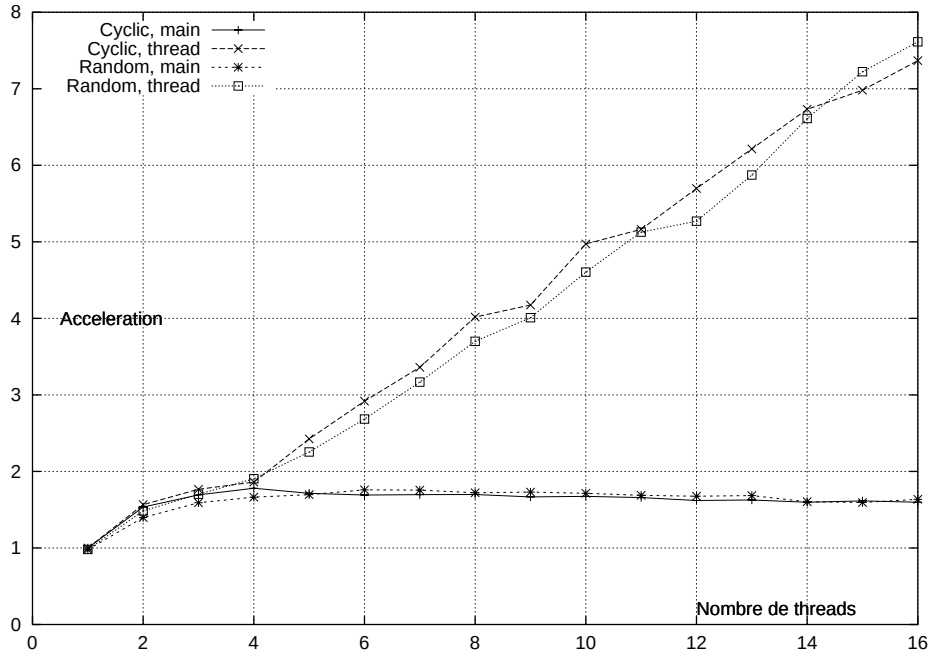


Figure 5.4 *Accélération du programme IRA, sur un quadriprocesseur, en fonction du nombre de threads utilisés pour ordonnancer les tâches de calcul. Sont présentées les accélérations pour 2 ordonnancements (cyclique et aléatoire), et dans chaque cas on donne les performances du programme total (main) et de chaque thread.*

Deux algorithmes d'ordonnancement sont ici présentés : le premier affecte les tâches prêtes de façon cyclique sur les 4 processeurs. Le second les affecte de façon aléatoire. Pour chaque ordonnancement, deux courbes sont données :

– l'accélération du programme total (main). Il va de soi que pour les deux ordonnancements elle stagne à partir de 4 threads puisque seuls quatre processeurs sont disponibles ;
– l'accélération mesurée dans chaque thread qui participe au calcul, c'est-à-dire T_1/T_p^m, où T_p^m est le temps d'exécution maximum parmi les p threads qui s'exécutent. Pour les deux ordonnancements proposés on constate que l'accélération est linéaire en ce qui concerne les threads : on peut donc attendre un bon comportement du programme ATHAPASCAN sur un SMP disposant de plus de noeuds.

On remarque que l'accélération, en valeur absolue, culmine pour le programme total autour de 1.8, soit une efficacité de moins de 0.5 en tenant compte des 4 noeuds de calcul. Un benchmark parallèle NAS, basé sur Open-MP, obtient cet ordre de grandeur sur cette

machine, sur un algorithme de gradient relativement proche du type d'opérations effectuées par IRA. L'utilisation de la librairie PAPI ([10]) dans le cadre du travail de M. Pillon a permis d'associer cette faible performance à une saturation des accès au bus mémoire sur ce type d'application, sur cette machine déjà vieille.

Sur cet exemple on n'observe pas de différence significative entre les deux algorithmes d'ordonnancement proposés. Au niveau de l'accélération mesurée sur un thread, la performance est néanmoins légérement meilleure avec l'ordonnancement cyclique.

En conclusion de ce chapitre nous avons présenté plusieurs stratégies d'ordonnancement, statiques et dynamiques, pour machines à memoire partagée et machines à mémoire distribuée, pour lesquelles l'analyse du flot de données permet l'obtention de performances théoriquement bonnes. Des expériences menées par M. Doreille illustrent la possibilité qu'apporte l'environnement ATHAPASCAN d'utiliser ces algorithmes pour obtenir de bonnes performances. Notre expérimentation sur SMP avec le programme IRA semble prometteuse pour les performances des versions distribuées.

6

Contrôle de l'espace pour MP2

Des ordonnancements efficaces en temps ont été décrits dans le chapitre précédent. On s'intéresse dans ce chapitre à des ordonnancements qui, tout en garantissant un temps d'exécution optimal, limitent la consommation mémoire du programme.

Le Graphe de Flot de Données (GFD) définit un ordre partiel sur les tâches de l'algorithme qu'il décrit. Une exécution d'un programme qui implémente ce GFD superpose à cet ordre partiel un ordre global, compatible avec lui. Cet ordre global peut être défini par le programmeur. C'est par exemple le cadre proposé par l'environnement Cilk [6], qui repose sur un ordre séquentiel d'exécution en profondeur d'abord. Un ordonnancement glouton au plus près de cet ordre séquentiel permet alors d'envisager des bonnes performances en temps et en mémoire.

L'autre approche que nous proposons consiste à calculer automatiquement un ordre séquentiel total respectant l'ordre local, qui optimise l'allocation mémoire. Minimiser la mémoire sur des exécutions parallèles n'est pas intéressant puisque d'un ordonnancement parallèle optimal en mémoire on déduit simplement un ordonnancement séquentiel optimal en mémoire : l'optimum en mémoire calculé en parallèle est donc forcément supérieur ou égal à l'optimum en mémoire calculé en séquentiel. Une fois un ordonnancement séquentiel calculé qui donne un espace mémoire minimal on peut ordonnancer en parallèle les nœuds du GFD en suivant au plus près cet ordre séquentiel optimal.

Pour calculer l'ordre séquentiel optimal on doit annoter le GFD, selon un modèle d'exécution tenant compte de la mémoire. On peut ensuite envisager des algorithmes ou des heuristiques permettant le calcul du meilleur ordonnancement.

Ce chapitre présente tout d'abord un état des lieux des langages basés sur une exécution séquentielle : Cilk et Nesl pour les graphes série-parallèles (sections 6.1.1 et 6.1.2)

et la généralisation proposée par Blelloch/Narlikar. Ces deux approches imposent au programmeur le choix d'un ordre séquentiel de référence. La section 6.2 propose un modèle de programmation parallèle avec annotation du GFD pour tenir compte de la mémoire, qui permet de calculer un ordonnancement séquentiel optimal en mémoire. La section 6.3 explique ce calcul et sa complexité pour l'ordonnancement glouton basé sur cet ordonnancement. Enfin nous montrons sur MP2 comment l'analyse du flot de données permet des ordonnancements efficaces en mémoire, en séquentiel et donc en parallèle.

6.1 Techniques de contrôle de l'explosion mémoire des programmes parallèles

Cette section présente les solutions qui existent pour contrôler le coût en mémoire d'un programme parallèle. On peut commencer par restreindre la classe de graphes que l'on ordonnance. Dans le cadre des programmes série-parallèle un algorithme d'ordonnancement basé sur du vol de travail permet de borner le surcoût en mémoire dû au parallélisme (6.1.1). Plus généralement, on peut définir une classe d'ordonnancements des programmes représentés sous forme d'un graphe orienté acyclique (DAG), qui garantit un surcoût en $S_1 + \mathcal{O}(pT_\infty)$. (6.1.3). Cette classe d'ordonnanceurs repose sur un ordonnancement séquentiel du DAG et son espace mémoire S_1.

6.1.1 Restriction de la classe des programmes aux graphes série-parallèle

Une première approche pour contrôler le comportement mémoire des exécutions parallèles d'un programme est de restreindre le type de parallélisme utilisable par le programmeur. Un exemple de langage ayant cet objectif est Cilk[1] [33, 6], langage destiné à la programmation des machines parallèles à mémoire partagée dont le développement à débuté en 1993 au MIT. C'est une extension du langage C qui offre des primitives pour l'expression du parallélisme de contrôle par création explicite de tâches. Un modèle de coût, permettant de garantir les efficacités en temps et en consommation mémoire, est associé au modèle de programmation.

La description du parallélisme se fait à l'aide du mot clé `spawn` placé devant un appel de fonction, comme illustré figure (6.1)(a) page 105. Conceptuellement, lors de l'exécution du programme, une tâche sera créée pour évaluer cette fonction. La sémantique de cet appel diffère de celle de l'appel classique d'une fonction au sens où la procédure appelante peut continuer son exécution en parallèle de l'évaluation de la fonction appelée au lieu d'attendre son retour pour continuer. Cette exécution étant asynchrone, la procédure créatrice ne peut pas utiliser le résultat de la fonction appelée sans synchronisation. Cette

[1] La page officielle du projet est maintenue à l'url suivante : http ://supertech.lcs.mit.edu/cilk/.

synchronisation est explicite par utilisation de l'instruction `sync`. Cette instruction a pour effet d'attendre la terminaison de toutes les fonctions appelées en parallèle par la fonction mère avant ce `sync` : le parallélisme exprimé est donc de type série-parallèle (en tenant compte du fait que la tâche mère s'exécute en concurrence avec ses filles, jusqu'à rencontrer l'instruction `sync`), comme illustré figure (6.1)(b). Les tâches sœurs créées sont supposées indépendantes : il y a donc risque de concurrence sur les accès à la mémoire partagée, concurrence qui doit être gérée par l'utilisateur.

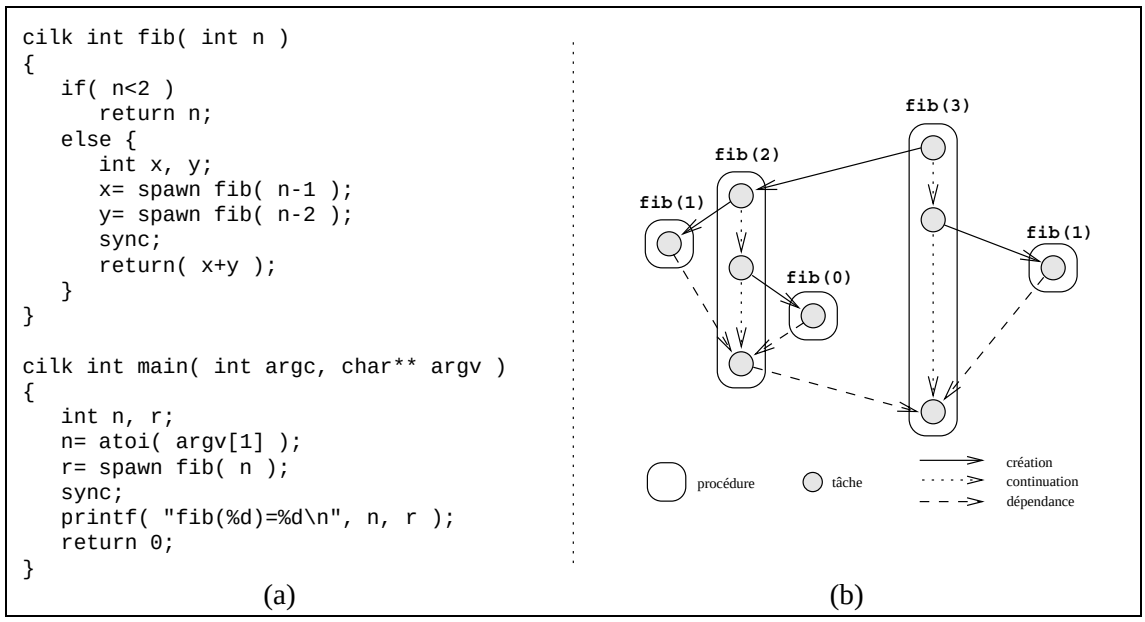


Figure 6.1 *Calcul du n-ième nombre de Fibonacci en Cilk.*
*En (a), dans le code Cilk, l'expression du parallélisme se fait en créant une tâche (*spawn*) à la place d'un classique appel de fonction. L'instruction* sync *permet d'attendre la terminaison de toutes les tâches créés par la procédure, ici* fib(n-1) *et* fib(n-2)*. En (b), le graphe d'exécution d'un appel à* fib(3) *est représenté. Ce graphe est forcément de type série-parallèle du fait de la synchronisation globale sur la fratrie effectuée par l'instruction* sync*.*

Au modèle de programmation Cilk est associé un modèle de coût [5]. En plus des grandeurs déjà définies en 4.1.2 page 69, chaque programme est caractérisé par S_1, la consommation mémoire de l'exécution sur un seul processeur lors d'une exécution en profondeur du graphe de tâches. Les tâches définies par l'utilisateur sont ordonnancées dynamiquement sur les processeurs par un algorithme de type liste, appelé *work-stealing*, qui fonctionne par vol de travail : lorsqu'un processeur devient inactif, il tire au sort un autre processeur, la victime, à qui il va voler une tâche prête à être exécutée. L'implantation de cet ordonnanceur garantit que la durée d'exécution T_p d'un programme (qui n'utilise pas de variables d'exclusion mutuelle) sur une machine à p processeurs et que la consommation mémoire S_p seront telles que :

$$\begin{aligned} T_p &= \frac{T_1}{p} + \mathcal{O}(T_\infty), \\ S_p &\leq pS_1. \end{aligned}$$

Le coût de l'ordonnancement est de l'ordre du nombre de requêtes de vol [7], donc de l'ordre de $\mathcal{O}(T_\infty)$ qui est considéré faible devant $\frac{T_1}{p}$. Une technique de compilation fine [23] permet de placer tout le surcoût d'ordonnancement lors des vols.

6.1.2 Ordonnancement au plus près d'un parcours en profondeur d'abord

Dans [2], une autre approche est proposée : En se basant sur un ordre séquentiel de référence, limiter les tâches du GFD ordonnancées selon un ordre différent afin d'éviter de faire exploser l'espace mémoire utilisé. L'ordonnancement séquentiel de référence proposé est le parcours en profondeur d'abord du graphe.

Selon cette technique, T_p reste le même mais

$$S_p = S_1 + \mathcal{O}(pT_\infty).$$

Cette nouvelle borne est meilleure dans les cas où $T_\infty = o(S_1)$, ce qui est le cas par exemple des problèmes de $\mathcal{NC}$. Néanmoins l'ordonnanceur en ligne utilisé souffre de plusieurs défauts : il impose des surcoûts d'ordonnancement trop importants. De plus, il est synchrone et ignore la localité des calculs dans l'ordonnancement qu'il calcule. Nous présentons dans cette section le langage Nesl, qui permet au programmeur l'expression d'un parallélisme série-parallèle avec cet ordonnancement.

NESL[2] est un langage de type fonctionnel qui permet d'exploiter un parallélisme de données de type série-parallèle. Il a été développé au tout début des années 1990 à l'université de Carnegie Mellon (le langage est toujours maintenu mais la dernière version date de novembre 1995). Le projet Scandal en est l'héritier.

La génération du parallélisme se fait en appliquant une même fonction sur une séquence de valeurs : la fonction sera appliquée en parallèle sur chacun des éléments de la séquence, comme illustré figure (6.2)(a). C'est un parallélisme de données de type SPMD. Un autre moyen de générer du parallélisme est d'utiliser des fonctions prédéfinies par le langage, fonctions parallèles qui opèrent sur une séquence dans son ensemble : par exemple le calcul de la somme, du tri ou la recherche du minimum. Tous ces appels sont synchrones, le programme ne passant à l'instruction suivante que lorsque le résultat a été entièrement calculé, c'est-à-dire lorsque toutes les fonctions parallèles générées ont été terminées ; le parallélisme généré est donc de type série-parallèle, comme illustré figure (6.2)(b).

Les tâches créées lors de l'exécution sont insérées dans un graphe de flot de données comme illustré figure (6.2)(b). Un modèle de coût est associé au modèle de programmation, en définissant cette fois S_1, l'espace mémoire requis par une exécution séquentielle,

[2]La page officielle du projet dans lequel est développé le langage NESL est maintenue à l'url suivante : urlhttp ://www.cs.cmu.edu/šcandal/nesl.html.

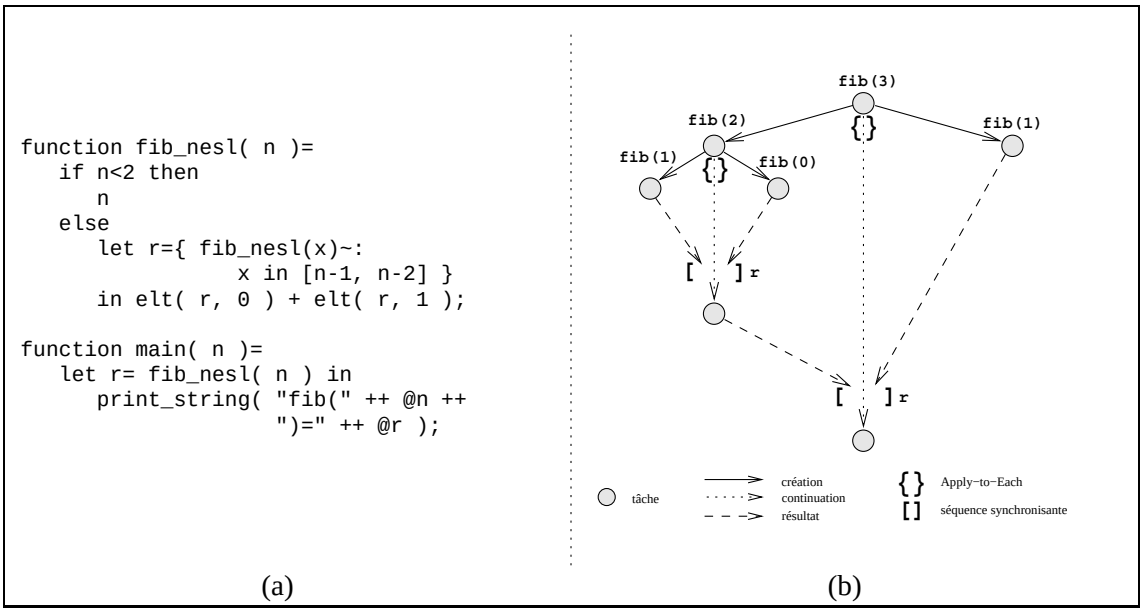


Figure 6.2 *Calcul du n-ième nombre de Fibonacci en NESL.*
En (a), dans le code NESL, l'expression du parallélisme se fait en appliquant une fonction sur une séquence à l'aide de la construction Apply-to-Each *$\{ f(x) : x$ in $seq \}$. Le résultat de cet opérateur est une nouvelle séquence obtenue en appliquant la fonction fib à chacun des éléments de la séquence initiale. La fonction elt() retourne un élément d'une séquence et l'opérateur tranforme un nombre en une chaîne de caractères. En (b), le graphe d'exécution d'un appel à fib(3) est représenté. Ce graphe est forcément de type série-parallèle, les tâches mères étant synchronisées sur les séquences produites par les tâches filles.*

cette exécution correspondant à un parcours du graphe en profondeur d'abord (en parcourant les tâches filles selon leur ordre de création, c'est-à-dire de gauche à droite dans les représentations classiques du graphe). Pour l'exemple de Fibonacci, $S_1 \approx n$.

Les processeurs sont associés aux nœuds de ce graphe dynamiquement et par étapes : régulièrement un nombre fixé ($p \log p$, afin de permettre le recouvrement du coût d'ordonnancement) de tâches prêtes sont affectées aux p processeurs de la machine. Le nombre d'étapes de cet algorithme d'ordonnancement peut être majoré, ce qui permet de garantir les résultats suivants pour la durée d'exécution T_p et la consommation mémoire S_p de toute exécution sur une machine à p processeurs :

$$\begin{aligned} T_p &\leq O\left(\frac{T_1}{p} + T_\infty \log p\right) \\ S_p &\leq O(S_1 + T_\infty p \log p) \end{aligned}$$

L'implantation de cet ordonnancement est basé sur un algorithme probabiliste, les performances en temps sont donc obtenues avec une forte probabilité. Pour des programmes générant suffisamment de parallélisme, c'est-à-dire pour lesquels $\frac{T_1}{p} \gg T_\infty$ et $S_1 \gg T_\infty$, les performances obtenues sont à un facteur $1 + o(1)$ des optimales. Le nombre de tâches créées est limité par une technique de création paresseuse [49, 4] ce qui permet d'expri-

mer des tâches de durée très petite au niveau du code source de l'application sans que cela soit pénalisant pour les performances à l'exécution.

6.1.3 Langages avec variables de synchronisation

Ces deux résultats sont obtenus dans le cadre des graphes série-parallèles. Blelloch, Narlikar *et al.* proposent dans [3] une généralisation de ce dernier résultat au cas des langages avec variables de synchronisation. En conservant les notations de la section précédente et en définissant σ le nombre de synchronisations (c'est-à-dire le nombre d'accès à des variables de synchronisation à assignation unique), l'algorithme Async-Q propose un ordonnancement en ligne du GFD, qui respecte les bornes suivantes sur une machine PRAM à p processeurs :

$$\begin{aligned} T_p &= \mathcal{O}(\frac{T_1}{p} + \frac{\sigma \log(pT_\infty)}{p} + T_\infty \log(pT_\infty)), \\ S_p &\leq S_1 + \mathcal{O}(pT_\infty \log(pT_\infty)). \end{aligned}$$

Async-Q est de plus asynchrone et peut être implémenté efficacement.

Il reste basé sur un ordre séquentiel de référence qui est le parcours en profondeur du GFD, en retardant l'exécution des tâches qui effectuent des grosses allocation en mémoire. De la sorte on évite de les exécuter tôt (et donc potentiellement en parallèle).

L'inconvénient de ces solutions est d'imposer, par un mécanisme d'ordonnancement qui est fondé sur lui, un ordre séquentiel global sur le GFD. La fin de ce chapitre propose de garder un ordonnancement parallèle basé sur un ordre global séquentiel de référence, mais en laissant libre le programmeur (ou l'environnement de programmation) de le calculer.

6.2 Intérêt de la représentation par le Graphe de Flot de Données pour prendre en compte la mémoire

Cette section décrit le modèle d'exécution basé sur une annotation du GFD pour tenir compte de la mémoire dans l'ordonnancement. Nous commençons par décrire deux modèles déjà existants avant de présenter notre modèle basé sur le GFD.

6.2.1 Exemples de modèles

6.2.1.1 Le modèle de Simpson et Burton

Un modèle d'exécution basé sur un graphe de dépendance ne prend en compte que les tâches et définit sur elles un ordre partiel. Pour y ajouter la mémoire, dans le cadre de graphes série-parallèles, Simpson et Burton [59] proposent de définir sur le graphe deux fonctions net et high. Ils les construisent récursivement : sur un nœud n,

(i). $\mathrm{net}(n)$ renvoie la quantité de mémoire allouée par la tâche, moins celle libérée ;

(ii). $\mathrm{high}(n)$ renvoie la mémoire maximum utilisée par la tâche au cours de son exécution.

Il faut également les règles de calcul pour passer des nœuds au graphe. Etant donnée la définition récursive d'un graphe série-parallèle, les auteurs donnent les règles de calcul de net et high lors de la composition d'un graphe G par composition série de deux sous-graphes G_1 et G_2 ($G = G_1 + G_2$) et par composition parallèle ($G = G_1 \| G_2$) :

$$\begin{aligned}
\mathrm{net}(G_1 + G_2) &= \mathrm{net}(G_1) + \mathrm{net}(G_2)\ ; \\
\mathrm{high}(G_1 + G_2) &= \max\left(\mathrm{high}(G_1), \mathrm{net}(G_1) + \mathrm{high}(G_2)\right)\ ; \\
\mathrm{net}(G_1 \| G_2) &= \mathrm{net}(G_1) + \mathrm{net}(G_2)\ ; \\
\mathrm{high}(G_1 \| G_2) &= \max\left(\mathrm{high}(G_1 + G_2), \mathrm{high}(G_2 + G_1)\right).
\end{aligned}$$

Avec cette modélisation de la mémoire les auteurs peuvent calculer récursivement l'espace mémoire requis par une exécution séquentielle en profondeur d'abord du graphe série-parallèle et l'utiliser pour l'ordonnancer en parallèle.

6.2.1.2 Le modèle de Blelloch/Narlikar

Dans le cadre de l'ordonnancement d'un programme parallèle avec variables de synchronisation, et donc d'un DAG qui n'est plus forcément série-parallèle, Blelloch et al. proposent un modèle d'exécution prenant en compte la mémoire [3].

À chaque nœud v du DAG est associé un couple d'entiers :
- $n(v)$, le "bilan", en terme d'espace alloué par la tâche. C'est la différence entre le total de l'espace alloué et le total de l'espace libéré par la tâche. $n(v)$ peut donc être négatif ;
- $h(v)$, l'espace maximum requis au cours de l'exécution de la tâche.

En notant V_i l'ensemble des tâches en cours d'exécution au top i et C_i l'ensembles des tâches qui terminent à un top $j \leq i$, la mémoire requise pour l'exécution d'un programme à n_e entrées selon l'ordonnancement défini par les $V_i, i = 1 \ldots N$, est :

$$\mathcal{S}_V = n_e + \max_{i=1\ldots N} \left(\sum_{v \in C_i} n(v) + \sum_{v \in V_i \setminus C_i} h(v) \right).$$

La première sommation représente la mémoire qui est en cours d'allocation par la suite de tâches qui ont été ordonnancées à l'instant i. La deuxième est un majorant sur l'espace requis par les tâches en cours d'exécution à cet instant.

Les modèles de Burton et de Blelloch/Narlikar permettent de tenir compte, de façon abstraite, de la mémoire pour ordonnancer efficacement un graphe. ATHAPASCAN propose un environnement concret qui permet, pour toute application, de calculer à la volée le GFD sur lequel on peut appliquer le type de travaux exposés ci-dessus. La suite de ce chapitre propose donc une solution pour adapter le GFD afin de prendre en compte la mémoire et d'obtenir ainsi *via* ATHAPASCAN des programmes efficaces en temps et en mémoire.

6.2.2 Graphe de Flot de Données et espace mémoire

Par définition les données sont codées dans le GFD et on peut donc naturellement annoter le graphe avec l'espace mémoire qu'elles occupent. Pour tout nœud donnée $v \in V_d$ on note $\mathcal{M}(v)$ le volume mémoire que la donnée occupe. C'est le net au sens de la section précédente. On n'affecte pas de volume mémoire aux tâches de calcul elles-mêmes : on considère que la pile locale à chaque tâche est de taille bornée. Les nœuds "tâches" du GFD ajoutent donc un volume constant d'occupation mémoire.

L'intérêt du GFD est de pouvoir prendre en compte les opérations de diffusion et de réduction des données de manière plus fine qu'un graphe de précédence. De même que [18] et [24] l'utilisent pour borner le volume de messages nécessaire à un ordonnancement optimal en temps, nous proposons donc de tirer parti des dépendances entre les nœuds données et les nœuds tâches du GFD pour affiner le modèle de mémoire en précisant les dates d'allocation et de libération de la mémoire requise par les données.

Annotation des arêtes. Le GFD permet un contrôle fin des allocations et libérations mémoires pour chaque type d'accès à une donnée : lecture ; écriture ; lecture concurrente ; écriture concurrente ; lecture-écriture. On va donc associer à chaque arête a entre une donnée $v \in V_d$ et une tâche $t \in V_t$ un poids $w(a)$ qui quantifiera l'action sur la mémoire.

Lecture. $v \in V_d$ une fois lue par une tâche t, la place qu'elle occupe en mémoire partagée peut être libérée. On associe donc un poids $w(a) = -\mathcal{M}(v)$ à l'arête $v \longrightarrow t$. Le signe $-$ exprime la libération de la mémoire $\mathcal{M}(v)$ qui était occupée par v.

Écriture. C'est le cas symétrique du précédent : une tâche t écrit la donnée v. Elle doit donc allouer de la mémoire partagée : on associe un poids $w(a) = \mathcal{M}(v)$ à l'arête $t \longrightarrow v$.

Lecture concurrente. Plusieurs tâches $t_i \in V_t, i = 1 \dots R$ accèdent en lecture à la donnée v. Différencier les arêtes $a_i = v \longrightarrow t_i$ permet de définir à quel moment la mémoire $\mathcal{M}(v)$ peut être libérée : quand la dernière tâche (selon l'ordonnancement choisi) a effectué l'accès. Les accès des autres tâches ne modifient pas l'état de la mémoire. On note donc i_f l'indice de la dernière tâche à effectuer l'accès, et on pose :

$$\begin{aligned} w(v \longrightarrow t_{i_f}) &= -\mathcal{M}(v)\ ; \\ w(v \longrightarrow t_i) &= 0, \forall i \in 1 \dots R, i \neq i_f. \end{aligned}$$

Écriture concurrente. De même que pour les écritures concurrentes on peut fixer exactement quelle tâche $t_i \in V_t$ parmi les $i \in 1 \dots W$ qui accèdent une donnée v en écriture concurrente va effectuer l'allocation de la mémoire nécessaire : la première à effectuer l'écriture, selon l'ordonnancement choisi. En notant i_0 l'indice de cette tâche, on pose donc :

$$\begin{aligned} w(t_{i_0} \longrightarrow v) &= \mathcal{M}(v)\ ; \\ w(t_i \longrightarrow v) &= 0, \forall i \in 1 \dots R, i \neq i_0. \end{aligned}$$

Lecture-écriture. L'accès en lecture-écriture ne permet que la modification de la mémoire pointée par la donnée. Une arête $a : v \longleftrightarrow t$ porte donc un poids $w(a) = 0$.

Annotations des tâches. On peut alors définir l'espace mémoire requis par un noeud tâche t du graphe $G = (V, T)$: soit l'ensemble R_t des arêtes exprimant une lecture d'un nœud donnée d par t. R_t est l'ensemble des prédécesseurs de t dans G. Et soit W_t l'ensemble des arêtes exprimant une écriture par t (ensemble des successeurs de t). Alors on pose :

$$\mathcal{M}(t) = \sum_{a \in R_t \cup W_t} \mathcal{M}(a).$$

6.3 Calcul de l'ordonnancement séquentiel optimal d'un GFD

Muni de ce modèle d'exécution on peut calculer l'ordonnancement séquentiel du GFD optimal en mémoire.

6.3.1 Formalisation du problème d'optimisation

Soit un ordonnancement séquentiel des tâches t_{σ_i} du graphe (dans le cadre séquentiel il suffit de fixer un ordre d'exécution des tâches de G pour ordonnancer tout le graphe),

où σ est une permutation de $\{1, \ldots, n\}$. On note V_i l'ensemble des tâches prêtes au top i, c'est-à-dire celles dont les prédécesseurs ont été ordonnancées à un top $\leq i-1$ (au top V_1 contient les les noeuds de degré entrant nul). On peut alors définir l'espace mémoire requis par cet ordonnancement :

$$\mathcal{E}(\sigma) = \max_{i=1,\ldots\ldots} \sum_{t \in V_i} \mathcal{M}(t).$$

On cherche donc l'ordonnancement σ qui minimise $\mathcal{E}(\sigma)$. Ce problème sera noté MinS_1 dans la suite.

6.3.2 Complexité du problème

Cette section montre que le problème d'optimisation ci-dessus est NP-dur. Pour cela on commence par se ramener au problème de décidabilité qui lui est polynomialement NP-réductible (*cf.* [25] chap. 5) : étant donné le GFD $G = (V, E)$ annoté comme il est décrit ci-dessus et un entier M, existe-t-il un ordonnancement σ et des ensembles $R_i, W_i, i = 1 \ldots n$, tels que $\mathcal{E}(\sigma) \leq M$?

On définit le problème MRS (Minimum Register Sufficiency) ainsi : étant donné un graphe orienté sans circuit $G = (V, E)$ et un entier naturel K, existe-t-il un calcul de G utilisant au plus K registres, c'est-à-dire une numérotation $v_1, v_2, \ldots, v_n$ des nœuds de V ($n = |V|$) et une suite $S_0, S_1, \ldots, S_n$ de sous-ensembles de V, chacun tel que $|S_i| \leq K$, tels que :

- $S_0 = \emptyset$,
- S_n contient tous les nœuds de degré entrant nul ;
- $\forall i = 1 \ldots n$:
  - $v_i \in S_i$;
  - $S_i \backslash \{v_i\} \subseteq S_{i-1}$;
  - S_{i-1} contient toutes les nœuds u tels que $(v_i, u) \in E$.

Ce problème est NP-complet ([25]).

Pour montrer la NP-difficulté de MinS_1, il suffit de montrer que MRS s'y réduit polynomialement : MRS $\prec_{\text{NP}}$ MinS_1, c'est-à-dire que si l'on se donne un oracle résolvant MinS_1, on sait résoudre MRS en temps polynomial.

La preuve de la réduction est directe : étant donné un graphe orienté sams circuit G et un entier naturel K, on construit le graphe G' formé des noeuds et des arêtes de G, avec l'ajout pour chaque nœud t d'un nœud t' et d'une arête $t \longrightarrow t'$ de poids 1. Le nœud t' est un nœud "donnée" accédé en écriture par le nœud "tâche" t. Dans G' chaque arête a a ainsi le poids $w(a) = 1$ ou $w(a) = 0$ (pour les arêtes de $G \backslash G'$). La construction de G' se fait en temps linéaire par rapport à la taille de G. L'oracle permet la résolution de MinS_1 en temps polynomial sur G'.

Si la réponse est positive alors l'ordonnancement σ des tâches avec $\mathcal{E}(\sigma) \leq K$ définit N ensembles $V_i, i = 0 \ldots N-1$ de tâches prêtes au top i, t toutes (par définition de G') de poids $\mathcal{M}(t) = 1$, donc en nombre inférieur ou égal à K. En notant $S_i \stackrel{\text{def}}{=} V_{N-i}, i = 0 \ldots N-1$, et $S_0 \stackrel{\text{def}}{=} \emptyset$, on a bien que $S_N = V_0$ contient les tâches de degré entrant nul ; que S_N est vide. Numérotons également les tâches "à l'envers" : soit $v_i \stackrel{\text{def}}{=} t_{\sigma_{N-1}}$. Par définition, les successeurs d'une tâche t ordonnancée au top i seront dans V_{i+1} et donc S_i vérifie la dernière propriété également.

Comme V_{N-1} contient forcément la dernière tâche ordonnancée $t_{\sigma_{N-1}}$, par définition $v_1 \in S_1$. Et en supposant que $v_i \in S_i$, alors soit v_{i+1} est un prédécesseur de v_i et alors par définition, v_{i+1} était dans S_{i+1} ; soit v_{i+1} était déjà prête avant que v_i ne soit ordonnancée, mais alors elle était déjà forcément dans S_{i+1}. Dans les deux alternatives on a la même conclusion : $v_{i+1} \in S_{i+1}$. Par récurrence on conclut donc : $\forall i, v_I \in S_i$.

La dernière propriété, $S_i \backslash \{v_i\} \subseteq S_{i-1}$, est évidente car une tâche prête au pas i le reste au pas $i+1$.

La numérotation v_i et les ensembles V_i sont donc solution du problème MRS.

Si la réponse est négative la preuve ci-dessus montre que la numérotation demandée pour MRS n'est que l'ordonnancement σ de MinS_1, sur G', à l'envers. Si la réponse est négative pour ce dernier, il ne peut donc y avoir de numérotation qui satisfasse MRS.

On a donc en temps polynomial une réponse à MRS qui utilise un oracle MinS_1, ce qui établit la réductibilité et donc que MinS_1 est NP-dur. □

Enfin, étant donnée une solution au problème MinS_1, il est trivial de parcourir les N noeuds de G et de calculer $\mathcal{E}(\sigma)$ en temps linéaire, afin de vérifier la propriété $\mathcal{E}(\sigma) \leq M$. Ceci prouve que Min$S_1$ est dans NP.

MinS_1 est donc NP-complet. □

6.3.3 Heuristiques de résolution et calcul de l'espace minimum

Heuristique. Dans [36], Klein *et al.* proposent une heuristique pour déterminer le nombre $S(n)$ de registres nécessaires à l'ordonnancement d'un graphe G à n tâches en temps polynomial, avec $S(n) = \mathcal{O}(S^*(n) \log^2 n)$, où $S^*(n)$ est le nombre minimal de registres nécessaires.

Étant donnée l'interdépendance montrée ci-dessus entre MinS_1 et MRS, l'heuristique de Klein permet de calculer, en temps polynomial, un ordonnancement séquentiel en mémoire à un facteur $\log^2 n$ de l'optimal.

Calcul de l'espace optimal. Si l'on veut calculer exactement l'espace optimal, on peut utiliser la borne supérieure obtenue par l'heuristique de Klein et la combiner avec un algorithme de recherche exhaustive, type Branch and Bound, qui utilisera la borne heuristique pour élaguer fortement l'arbre de recherche. On peut également contraindre le temps de parcours arborescent en décidant de ne pas chercher l'optimal mais simplement de parcourir les solutions jusqu'à amélioration de la borne de l'heuristique.

6.4 Application : flot de données de MP2 et ordonnancement

Cette section illustre sur MP2 le calcul du GFD et les divers ordonnancements que l'on peut envisager afin de minimiser la mémoire requise. Le calcul selon Pople (*cf.* le chapitre 3, section 3.3.2.1 page 58) propose en fait un ordonnancement des différentes étapes du calcul de MP2 : à l'aide d'un paramètre (la taille des seaux I) le programmeur peut régler le taux de recalcul en fonction de l'espace mémoire dont il dispose.

L'approche par analyse du GFD permet de généraliser ce système d'ordonnancement, en prenant en compte la mémoire comme expliqué dans les sections ci-dessus. Nous commençons dans la section suivante par étudier la granularité du problème MP2 afin de définir les tâches du GFD. Pour cela nous utilisons une réécriture matricielle de l'algorithme. Nous montrons ensuite comment ordonnancer le GFD obtenu afin d'être efficace en mémoire.

6.4.1 Analyse du flot de données de MP2

Le premier choix à faire est celui de la granularité du programme parallèle. On décide de ne pas se placer au grain le plus fin, mais à celui d'un bloc d'ERI. Cela permet de travailler sur des matrices, pour lesquelles des outils numériques standards existent qui garantissent des implémentations aux performances quasiment optimales en séquentiel. Les transformations se réécrivent naturellement comme des produits matriciels.

6.4.1.1 Reformulation de MP2

Dans toute la suite on utilisera des matrices de matrices (ou supermatrices) : M est une matrice réelle à $N_l.N_{bl}$ lignes et $N_c.N_{bc}$ colonnes, décrite comme une matrice de $N_l \times N_c$ blocs de taille $N_{bl} \times N_{bc}$. On notera que M est de taille $[N_l N_c](N_{bl} N_{bc})$, et on notera $M_{[\alpha,\beta],(i,j)}$ le cœfficient d'indice $(\alpha-1)N+i, (\beta-1)N+j$, *i.e.* l'élément $(i,j)(1 \leq i \leq N_{bl}, 1 \leq j \leq N_{bc})$ du bloc $(\alpha,\beta), 1 \leq \alpha \leq N_l, 1 \leq \beta \leq N_c$ de M.

Le calcul MP2 utilise donc comme entrées une matrice M de taille [NN](NN) ‘avec $M_{[\lambda,\sigma],(\mu,\nu)} =< \mu\nu \mid \lambda\sigma >$.

Une instruction comme celle de la ligne 12 se réécrit

$$\begin{aligned} \forall\ i \in I,\ \forall\ \lambda,\sigma \in L,\quad M'_{[\lambda,\sigma],(i,\nu)} &= \sum_{\mu=1}^{N} C^t_{Oi,\mu} M_{[\lambda,\sigma],(\mu,\nu)}, \\ &= \left(C^t_O M_{[\lambda,\sigma]}\right)_{i,\nu}, \end{aligned}$$

c’est-à-dire que : $M'_{[\lambda,\sigma]} = C^t_O M_{[\lambda,\sigma]}$, et M' est $[NN](ON)$. La transformation $\nu \to a$ se réécrit de même comme le produit $M''_{[\lambda,\sigma],(i,a)} = \left(M'_{[\lambda,\sigma]} C_V\right)_{i,a}, \forall\ i \in I,\ \forall\ \lambda,\sigma \in L$, où M'' est une matrice $[NN](OV)$. On a donc directement :

$$M''_{[\lambda,\sigma]} = C^t_O M_{[\lambda,\sigma]} C_V\ \forall \lambda,\sigma = 1 \ldots N.$$

Ceci se réécrit encore en terme de supermatrice. On définit la matrice $[NN](ON)$ $\bar{C^t_O}$ dont seuls les blocs diagonaux sont non-nuls et valent tous la matrice C^t_O de taille $O \times N$. On se donne aussi la matrice $\bar{C_V}$ de taille $[NN](NV)$ dont seuls les blocs diagonaux sont non-nuls et valent tous la matrice C_V de taille $N \times V$. On obtient alors

$$M'' = \bar{C^t_O} \times \bar{C_V}.$$

De la même façon, les transformations $\lambda \to j$ et $\sigma \to b$ s’écrivent comme des produits matriciels. On note N la supermatrice $[OV](NN)$: $M_{[i,a],(\lambda,\sigma)} =< ia \mid \lambda\sigma >\ ,\forall\ i \in 1 \ldots O,\ \forall\ b \in 1 \ldots V,\ \forall\ \lambda,\sigma \in 1 \ldots N$. Il vient donc :

$$N''_{[i,a]} = C^t_O N_{[i,a]} C_V,\ \forall i = 1 \ldots O, \forall a = 1 \ldots V,$$

ou encore

$$N'' = \bar{C^t_O} \times N \times \bar{C_V}.$$

Pour calculer l’énergie E, on n’a plus qu’à effectuer la sommation sur les cœfficients de N''.

La transformation permettant de passer de M'' à N s’écrit simplement. Elle consiste à inverser les cœfficients des blocs et ceux des indices dans les blocs. C’est-à-dire que l’on effectue :

$$(\lambda-1)N+i, (\sigma-1)N+a \longrightarrow (i-1)O+\sigma, (a-1)V+\sigma,$$

afin de transformer une matrice $[NN](OV)$ en une matrice $[OV](NN)$.

Par exemple, une matrice $[22](23)$ sera ainsi transformée :

$$\left(\begin{array}{ccc|ccc} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \longrightarrow \left(\begin{array}{cc|cc|cc} 1 & 0 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 4 & 0 & 5 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

Soit $1 \leq p, q \leq N$ deux entiers naturels. Soit $\Delta_{p,q}$ la matrice $N \times N$ nulle partout sauf l'élément (p, q) qui vaut 1 ; soit $P_{p,q}$ la matrice $N \times N$ définie par :

$$P_{p,q} = I_n - \Delta_{p,p} - \Delta_{q,q} + \Delta_{p,q} + \Delta_{q,p} = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ \vdots & 0 & \dots & 1 & \dots \\ \vdots & 0 & 1 & \dots & \vdots \\ \vdots & 0 & 0 & 0 & \vdots \\ \vdots & 1 & 0 & 0 & \vdots \\ 0 & \dots & \dots & \dots & 1 \end{pmatrix},$$

où les 1 hors-diagonale sont en (p, q) et (q, p).

On vérifie facilement que la matrice $P_{p,q}A$ est la matrice A dont on a permuté les lignes p et q.

Dans notre système de notation par double indice, la ligne i du bloc λ a l'indice "global" $(\lambda - 1)N + i$, et pour permuter la ligne i du bloc λ avec la ligne λ du bloc i, il faut donc multiplier à gauche par $P_{(\lambda-1)N+i,(i-1)O+\lambda}$.

Si l'on veut maintenant permuter le cœfficient (i, a) du bloc (λ, σ) et le cœfficient (λ, σ) du bloc (i, a), on doit permuter les lignes mais aussi permuter les colonnes, donc multiplier à droite par $P_{(\sigma-1)N+a,(a-1)V+\sigma}$.

Comme on veut permuter tous les cœfficients de tous les blocs la transformation se ramène à multiplier M à gauche par :

$$P_l = \prod_{\lambda=1\dots N, i=1\dots O} P_{(\lambda-1)N+i,(i-1)O+\lambda}$$

et à droite par :

$$P_c = \prod_{\sigma=1\dots N, a=1\dots V} P_{(\sigma-1)N+a.(a-1)V+\sigma}$$

On a donc $N = P_l \times M'' \times P_c$ et finalement :

$$N'' = \bar{C_O^t} \cdot P_l \cdot \bar{C_O^t} \cdot M \cdot \bar{C_V} \cdot P_c \cdot \bar{C_V}. \tag{6.1}$$

Le problème du calcul de l'énergie MP2 se réduit donc à celui du calcul de la norme de Frœbenius de la matrice N'', les cœfficients de la matrice M étant fournis en entrée. La section suivante présente différents parenthésages de ce calcul matriciel et les GFD associés.

6.4.1.2 Flots de données de la reformulation de MP2

On propose ici deux parenthésages de ce calcul matriciel de MP2 et les GFD associés. L'un possède une synchronisation de moins que l'autre. On peut aussi les comparer aux deux versions MPI de Nielsen (*cf.* 3.3.2).

Parenthésage avec synchronisation globale. La première idée est d'effectuer d'abord localement les transformations de tous les cœfficients possibles, puis de "permuter" les indices de blocs et de cœfficients dans la matrice résultat afin de procéder aux deux autres transformations. Cela conduit au parenthésage suivant :

$$N'' = \bar{C_O^t} \cdot \left(P_l \cdot (\bar{C_O^t} \cdot M \cdot \bar{C_V}) \cdot P_c\right) \cdot \bar{C_V}. \tag{6.2}$$

Avec ce parenthésage, chaque bloc $[i,a]$ de N doit être multiplié à gauche par C_O^t et à droite par C_V. Mais $N_{[i,a]}$ est formé de cœfficients de M'' issus de chacun des N^2 blocs $M''_{[\lambda,\sigma]}$. On a donc nécessairement une synchronisation globale entre les N^2 calculs produisant les $M''_{[\lambda,\sigma]}$ et les OV calculs à partir des $N_{[i,a]}$.

Autre parenthésage. Afin d'éviter une synchronisation globale entre les données calculées, on propose le parenthésage suivant :

$$N'' = (\bar{C_O^t} \cdot P_l) \cdot (\bar{C_O^t} \cdot M \cdot \bar{C_V}) \cdot (P_c \cdot \bar{C_V}). \tag{6.3}$$

Si l'on regarde la structure des matrices $\bar{C_O^t} \cdot P_l$ et $P_c \cdot \bar{C_V}$, on obtient :

– $\bar{C_O^t} \cdot P_l$ est une matrice $[ON](OO)$, dont le bloc $[i,\mu]$ est constitué de la colonne μ de C_O^t, placée en colonne i du bloc (exemple avec $N = 5, O = 2$) :

$$\left(\begin{array}{c|c|c|c|c} C^t_{O1,1} & C^t_{O1,2} & C^t_{O1,3} & C^t_{O1,4} & C^t_{O1,N} \\ C^t_{O2,1} & C^t_{O2,2} & C^t_{O2,3} & C^t_{O2,4} & C^t_{O2,N} \\ \hline C^t_{O1,1} & C^t_{O1,2} & C^t_{O1,3} & C^t_{O1,4} & C^t_{O1,N} \\ C^t_{O2,1} & C^t_{O2,2} & C^t_{O2,3} & C^t_{O2,4} & C^t_{O2,N} \end{array}\right)$$

– a l'inverse, $P_c \cdot \bar{C_V}$ est structurée par lignes : c'est une matrice $[NV](VV)$ donc chaque bloc $[\nu,a]$ est constitué de la ligne ν de C_V, placée à la ligne a du bloc (exemple avec $N = 5, V = N - O = 3$) :

$$\left(\begin{array}{ccc|ccc|ccc} C^t_{V1,1} & C_{V1,2} & C_{V1,3} & C^t_{V1,1} & C_{V1,2} & C_{V1,3} & C^t_{V1,1} & C_{V1,2} & C_{V1,3} \\ \hline C^t_{V2,1} & C_{V2,2} & C_{V2,3} & C^t_{V2,1} & C_{V2,2} & C_{V2,3} & C^t_{V2,1} & C_{V2,2} & C_{V2,3} \\ \hline C^t_{V3,1} & C_{V3,2} & C_{V3,3} & C^t_{V3,1} & C_{V3,2} & C_{V3,3} & C^t_{V3,1} & C_{V3,2} & C_{V3,3} \\ \hline C^t_{V4,1} & C_{V4,2} & C_{V4,3} & C^t_{V4,1} & C_{V4,2} & C_{V4,3} & C^t_{V4,1} & C_{V4,2} & C_{V4,3} \\ \hline C^t_{V5,1} & C_{V5,2} & C_{V5,3} & C^t_{V5,1} & C_{V5,2} & C_{V5,3} & C^t_{V5,1} & C_{V5,2} & C_{V5,3} \end{array}\right)$$

La structure de ces matrices permet de simplifier les produits à effectuer. La figure (6.3) visualise la succession d'opérations à effectuer et les cœfficients concernés.

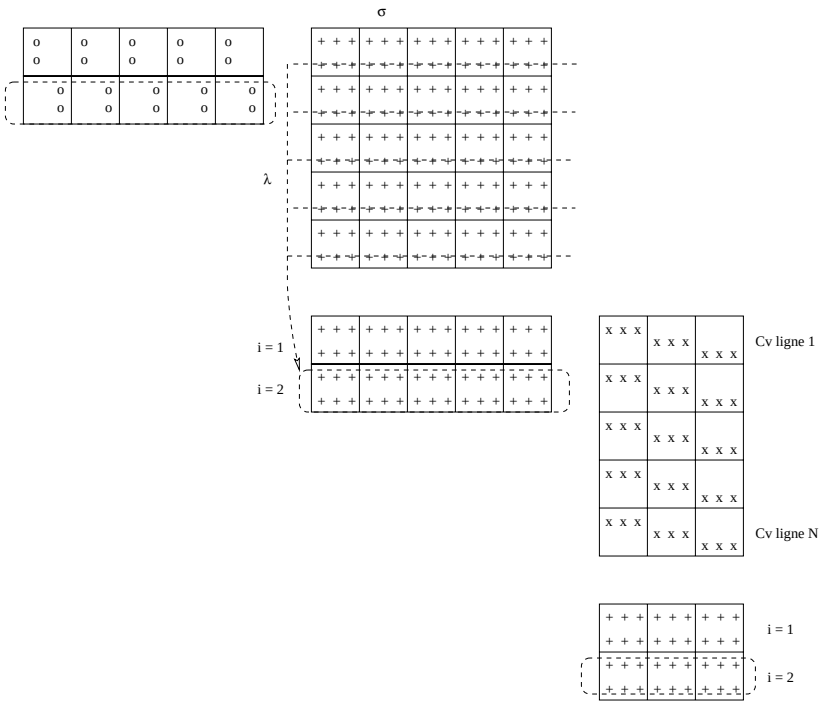


Figure 6.3 *Calculs par bloc pour obtenir l'énergie MP2.*

En haut au centre on trouve la matrice $\bar{C_O^t} \cdot M \cdot \bar{C_V}$, de taille $[NN](OV)$ qui est multipliée à gauche par $\bar{C_O^t} \cdot P_l$. Le résultat est donc $[ON](OV)$. Etant donnée la structure creuse de $\bar{C_O^t} \cdot M \cdot \bar{C_V}$, la superligne $i \in 1 \ldots O$ de cette dernière ne dépend que de la ligne i de chaque bloc $[\lambda, \sigma]$ de M'' (la figure montre comment on obtient les cœfficients de la ligne $i = 2$).

Le produit à droite par $P_c \cdot \bar{C_V}$ peut ensuite être effectué, pour obtenir une matrice $[OV](OV)$. Le bloc $[i, a], i \in 1 \ldots O, a \in 1 \ldots V$, ne dépend que des colonnes a de chaque bloc $[i, \sigma], \sigma = 1 \ldots N$ de la matrice précédente. Néanmoins, la contribution à E met en jeu pour un i fixé et un j fixé tous les a et b. Notre indiçage et nos opérations par bloc fournissant les V valeurs de a à partir d'un i donné, nous devons utiliser les V valeurs de b pour calculer la contribution de i, j et donc utiliser toute la superligne i de la matrice N obtenue à la fin.

Graphe de Flot de Données du dernier parenthésage. On peut tracer le GFD de ce dernier parenthésage.

La figure (6.4) indique les dépendances de données exactes de ce parenthésage dans le cas $N = 5, O = 2, V = 3$. En partant de la fin, on retrouve la dépendance entre E et les

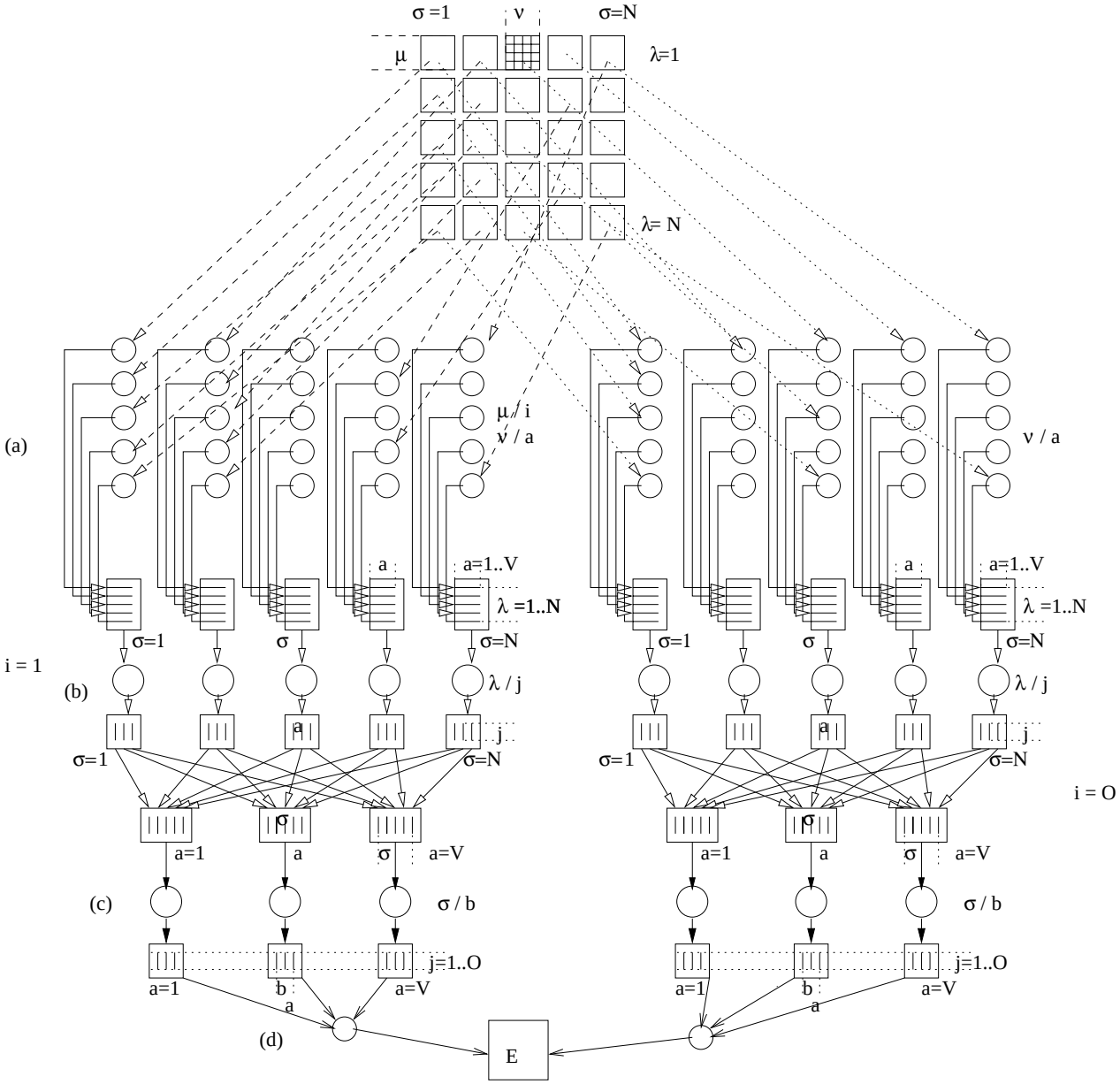


Figure 6.4 *Graphe de flot de données de l'algorithme de calcul MP2 dans la version sans synchronisation globale, pour le cas $N = 5, O = 2, V = 3$. On indique à côté de chaque groupe de tâches (a), (b) et (c) la ou les transformations qu'elles effectuent, et pour chaque donnée les indices qu'elles concernent.*

données i, $a, b = 1 \ldots V, j = 1 \ldots O$. Au dessus par contre on a indépendance totale des calculs pour différentes valeurs de i (*cf.* la branche $i = 1$ et la branche $i = 2 = O$). Les tâches (c) $\sigma \to b$ (produit à droite par $P_c \cdot \bar{C_V}$) se font en parallèle sur V blocs différents (blocs $[i, a]$ du paragraphe précédent). Néanmoins, comme précisé ci-dessus, chacun de ces blocs est formé à partir des colonnes a de chaque bloc $[i, \sigma], \sigma = 1 \ldots N$ de la matrice

précédente, ce qui se traduit par une dépendance "all-to-all" sur le graphe.

Toujours en remontant, on a auparavant la transformation $\lambda \rightarrow j$ (produit à gauche par $\bar{C_O^t} \cdot P_l$) qui s'effectue par les tâches (b) sur N blocs indépendants de taille NV. Ces blocs étaient le résultat des produits $\bar{C_O^t} \cdot M \cdot \bar{C_V}$ dont on n'avait gardé dans chaque moitié du graphe que la ligne i de chaque bloc : à partir des N^2 blocs de taille $N \times N$ de la matrice M de départ, on a donc N^2 tâches (a) qui calculent les lignes i des blocs du produit $\bar{C_O^t} \cdot M \cdot \bar{C_V}$ (en réalité il s'agit donc de deux produits vecteurs $\times$ matrices successifs) et effectuent donc les deux premières transformations $\mu \rightarrow i$ et $\nu \rightarrow a$.

Néanmoins on remarque que la dernière dépendance pour calculer E met en jeu à i fixé l'ensemble des données $a, b = 1 \ldots V$ et $j = 1 \ldots O$. Or la formule de calcul de E (3.5) de i, j indiquait que l'on pouvait mettre à jour E pour un couple $(i, j) \in 1 \ldots O$ donné.

Pour mettre à profit cette dépendance moins contraignante il faut effectuer les transformations dans un ordre différent afin de renuméroter les cœfficients des matrices employées.

Renumérotation des transformations. L'algorithme précédent transformait la matrice $M_{[\lambda,\sigma]}(\mu, \nu)$ en la matrice $N''_{[i,a]}(j, b)$. Afin de garder des produits par blocs tout en aboutissant après les quatre transformations à un ensemble de valeurs (a, b) pour un couple (i, j) fixé, on redéfinit l'ordre des cœfficients de façon à transformer $M_{[\nu,\sigma]}(\mu, \lambda)$ en $N''_{[i,j]}(a, b)$.

On pose donc $M_{[\nu,\sigma]}(\mu, \lambda) =< \mu\nu \mid \lambda\sigma >$. Les équations de transformations, de même que précédemment, se réécrivent alors :

$$N'' = (\bar{C_V^t} \cdot P_l) \cdot (\bar{C_O^t} \cdot M \cdot \bar{C_O}) \cdot (P_c \cdot \bar{C_V}), \tag{6.4}$$

où les matrices sont :
- $[NN](NN)$ pour M ;
- $[OO](NN)$ pour $\bar{C_O^t} \cdot M \cdot \bar{C_O}$;
- P_l est la même matrice que dans 6.1. Cette fois, par contre, on a $P_c = P_l$.

Le graphe de flot de données associé à cet algorithme est fourni par la figure (6.5).

6.4.1.3 Caractéristiques des graphes

Dans cette section on suppose que la complexité en calcul d'un produit matriciel d'une matrice $p \times q$ par une matrice $q \times r$ est pqr.

Dans les deux graphes présentés pour modéliser le calcul MP2 dans sa formulation matricielle, on peut annoter les tâches par leur complexité en calculs et en mémoire (selon les définitions présentées aux sections précédentes de ce chapitre). Par exemple dans le

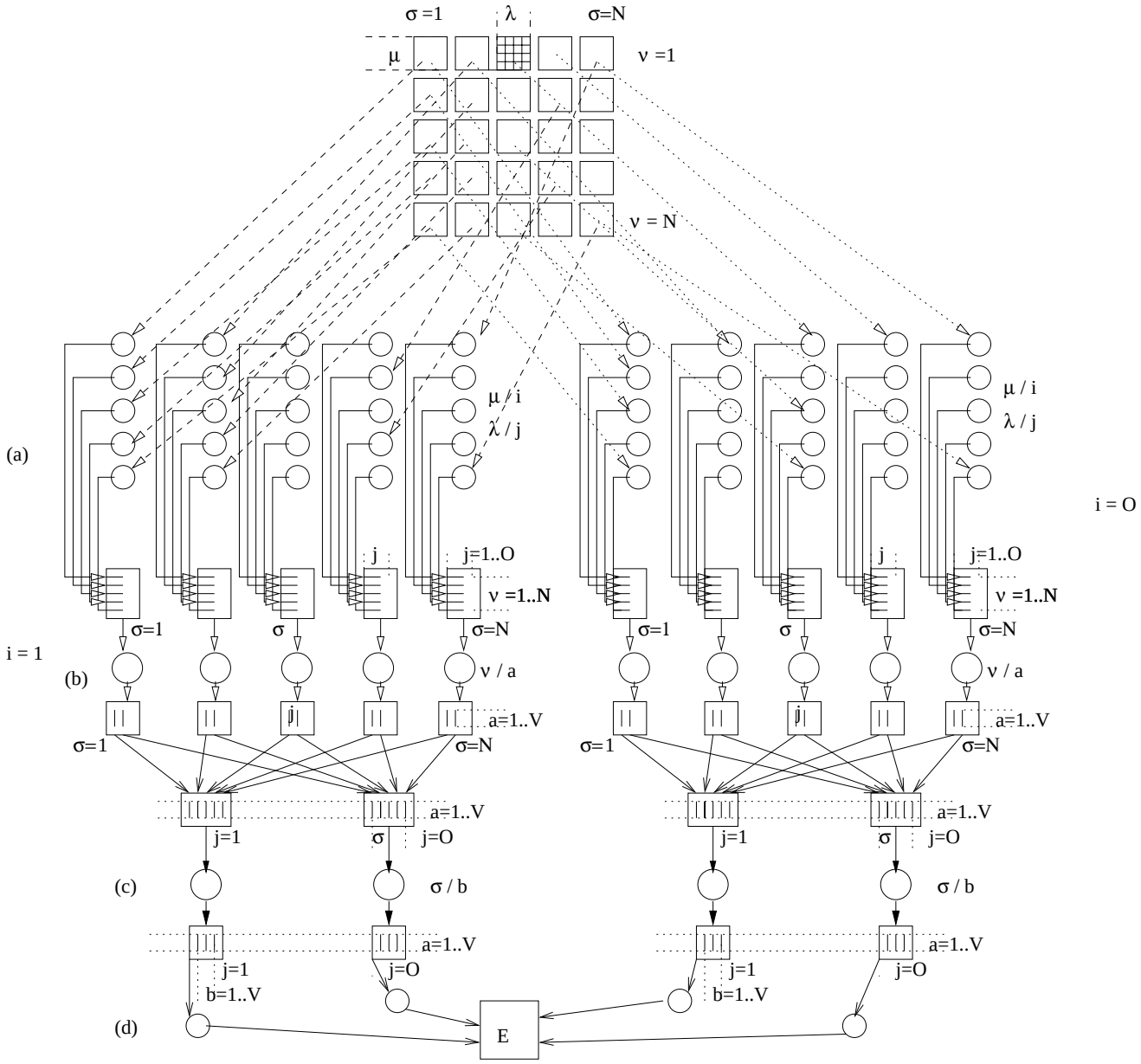


Figure 6.5 *Graphe de flot de données de l'algorithme de calcul MP2 dans la version sans synchronisation globale et avec renumérotation des transformations, pour le cas $N = 5, O = 2, V = 3$. On indique à côté de chaque groupe de tâches (a), (b) ou (c) la ou les transformations qu'elles effectuent et pour chaque donnée les indices qu'elle concerne.*

cas du premier graphe, les ON^2 tâches (a) qui calculent $\mu \to i$ et $\nu \to a$ ont chacune une complexité :

– $N^2 + NV$ en calcul (chaque tâche effectue le calcul de la ligne i du bloc λ, σ de la matrice M'' comme expliqué figure (6.3). Ceci implique un produit matriciel $1 \times N$

par $N \times N$ suivi d'un autre $1 \times N$ par $N \times V$) ;
– V en mémoire.

Les ON tâches (b) qui effectuent la transformation $\lambda \rightarrow j$ ont chacune la complexité suivante :
– ONV en calcul (produit matriciel ON par NV) ;
– OV en mémoire.

Les V^2 tâches de (c) sont en :
– ONV en calcul (produit matriciel ON par NV) ;
– OV en mémoire.

Enfin les tâches de (d) sont en :
– OV^2 en calcul (sommation sur les termes des données en entrée) ;
– 1 en mémoire.

Les complexités en temps et en mémoire de chaque tâche du GFD avec renumérotation des indices (figure (6.5)) sont légèrement différentes : les tâches du groupe (a) (1ère et 2ème transformations $\mu \rightarrow i$ et $\lambda \rightarrow j$) effectuent chacune $N^2 + NO$ opérations avec un espace mémoire O. Celles du groupe (b) ($\nu \rightarrow a$) s'effectuent en ONV calculs, avec une mémoire OV. Celles du groupe (c) ($\sigma \rightarrow b$) prennent un temps NV^2 et un espace mémoire V^2. Enfin celles de (d) se font en temps V^2 et en mémoire 1.

On peut alors calculer T_∞ pour chacun des deux graphes : il est constitué dans les deux cas de N tâches du groupe (a) suivies d'une tâche du groupe (b), d'une tâche de (c) et enfin d'une tâche (d). D'où, pour le graphe sans renumérotation (6.4) : $T_\infty = N^3 + N^2V + 2ONV + OV^2$. Dans le cas de l'autre graphe (6.5), $T_\infty = N^3 + N^2O + ONV + NV^2 + V^2 = 2N^3 + V^2$ en prenant en compte que $O + V = N$.

Les GFD présentés ne sont pas série-parallèles. Ils comportent tous deux une forte étape desynchronisation entre les tâches (b) et (c). Dans le cas du premier graphe le nombre σ de variables à assignation unique vaut NV pour chaque valeur de i, soit ONV au total. Dans le cas de l'autre GFD, σ vaut O^2N.

En conclusion sur ces deux graphes proposés, dans le cas $V \gg O$, le T_∞ du GFD numéro 2 est plus petit que celui du GFD numéro 1, et son nombre de synchronisations est également plus petit. La renumérotation des indices semble donc intéressante dans ce cas pour avoir plus de parallélisme. Physiquement ce cas correspond à un calcul sur un ensemble de particules en faible nombre, modélisé avec une grande base. Dans l'autre cas par contre le GFD (6.4) a un T_∞ qui paraît meilleur (terme dominant en N^3, contre $2N^3$ pour l'autre) et moins de synchronisations.

6.4.2 Ordonnancement théorique de MP2 avec contrainte sur la mémoire

Cette section compare les espaces mémoires requis par différents ordonnancements de MP2. Le GFD étant connu statiquement on peut calculer des ordonnancements efficaces

en mémoire directement.

6.4.2.1 Ordonnancement de Pople

L'ordonnancement proposé par Pople (décrit dans l'algorithme 4 page 59) utilise le graphe 6.4. L'ordonnancement est présenté de façon schématique dans la figure (6.6), page 124.

Le parcours du GFD proposé par Pople commence par une exécution en profondeur d'abord. En effet la boucle externe sur les valeurs de i ordonnance de façon séquentielle les branches du GFD sur indicées par i. À l'intérieur de ces branches, le calcul est fait dans l'ordre sur tous les indices $\nu\lambda\sigma$ pour la 1ère transformation $\mu \rightarrow i$; puis, pour une valeur de σ fixée (ligne 17), c'est-à-dire une branche de la partie i du GFD que l'on suit en descendant, on effectue les transformations $\nu \rightarrow a$ puis $\lambda \rightarrow j$ (lignes 21—23 ; tâches du groupe (b) sur la figure). Ensuite ("Fin Pour" de la ligne 26) on termine la boucle sur λ et σ et ce n'est que quand tous ces indices ont été transformés que l'on effectue la 4ème transformation, en suivant l'ordre sur a (*cf.* la figure, le groupe de tâches nommé (c)). Enfin on effectue le calcul de la contribution à l'énergie (tâche (d)), puis on exécute les tâches pour la valeur suivante de i (deuxième moitié du GFD de la figure) selon le même ordre.

On retrouve bien sûr avec le GFD parcouru dans cet ordre les complexités en temps et en mémoire de Pople. Pour chaque valeur de i Les N premières tâches (a) ordonnancées s'exécutent en temps (au total) $N^3 + VN^2$ et en espace mémoire VN. Elles sont suivies d'une tâche de (b) qui prend ONV calculs et un espace mémoire OV. Cette séquence de $N + 1$ tâches se répète N fois et seul l'espace mémoire de la dernière tâche a besoin d'être conservé, d'où un temps en $N^4 + VN^3 + ON^2V$ et un espace mémoire de NOV. Ensuite sont ordonnancées les V tâches de (c) qui prennent un temps (total) de ONV^2 et un espace mémoire (total) de OV^2. L'ultime tâche (d) s'exécute en temps OV^2 en espace mémoire 1.

Au total l'ensemble de cette étape pour une valeur de i prend donc un temps

$$\mathcal{O}(N^4 + VN^3 + ON^2V + ONV^2 + OV^2),$$

et un espace mémoire en $\max(NOV, OV^2, 1) = OVN$. Si enfin on ordonnance ces branches du GFD selon des valeurs de i en seaux I, on retrouve donc bien les complexités présentées par Pople dans la table 3.1 page 60.

6.4.2.2 Ordonnancement en profondeur et duplication de tâches

On peut également envisager de dupliquer les tâches afin de limiter l'espace mémoire requis tout en gardant un ordonnancement en profondeur d'abord. Cependant ceci n'est

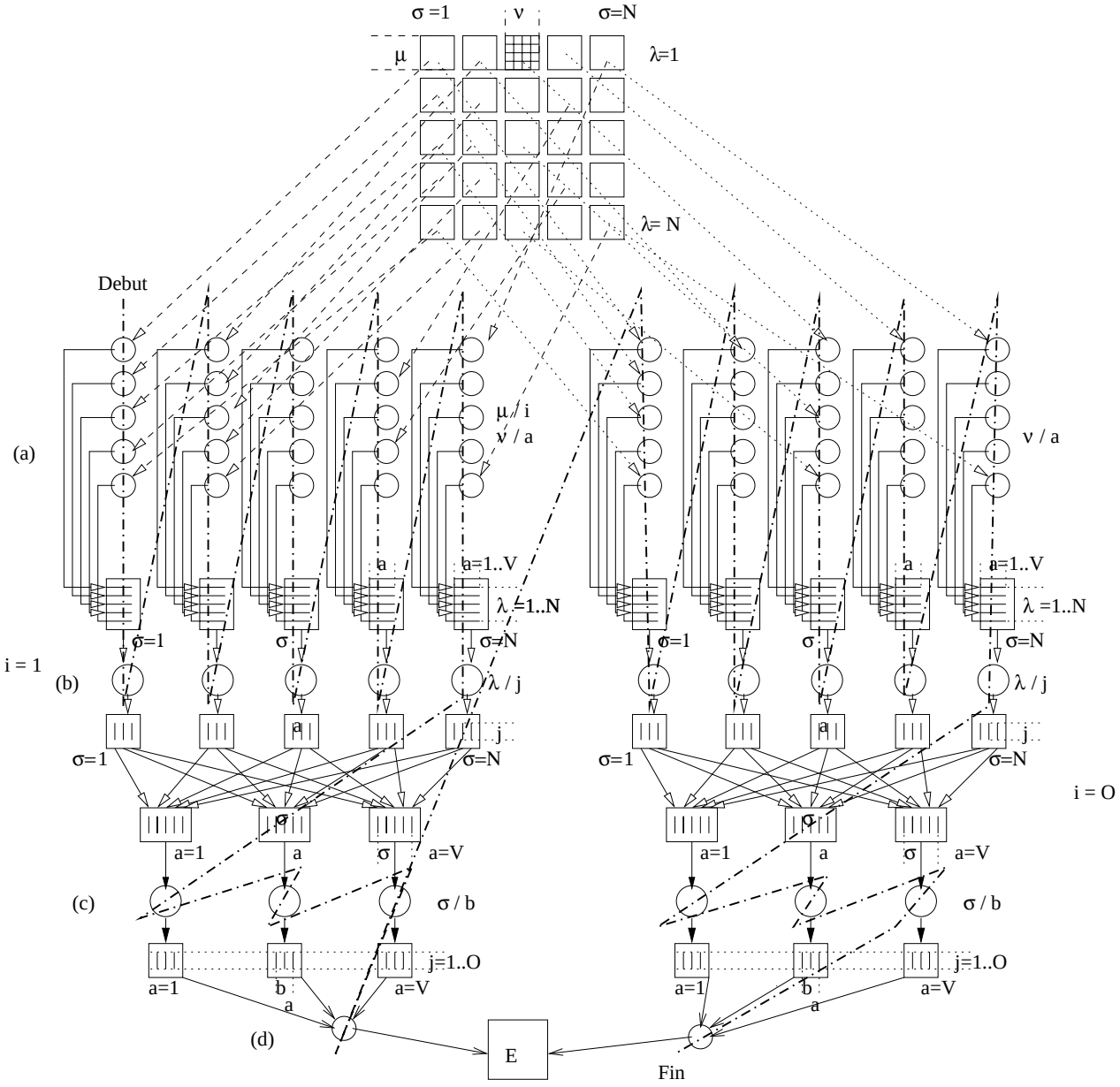


Figure 6.6 *Ordonnancement selon Pople du GFD de MP2. Les tâches sont exécutées en suivant le trait gras en pointillés, en commençant à "Debut" et en terminant à "fin".*

possible que sur le second graphe présenté dans la figure (6.5) page 121, moins synchronisé que celui équivalent au calcul de Pople. La figure (6.7) présente l'ordre d'exécution des tâches (a) et (b) pour le calcul d'une tâche de (c), le tout pour une valeur de i fixée, ainsi que l'espace mémoire requis par chaque tâche de ces groupes avec l'ordonnancement proposé.

Pour une valeur de i donnée on peut ordonnancer tout d'abord N tâches du groupe (a)

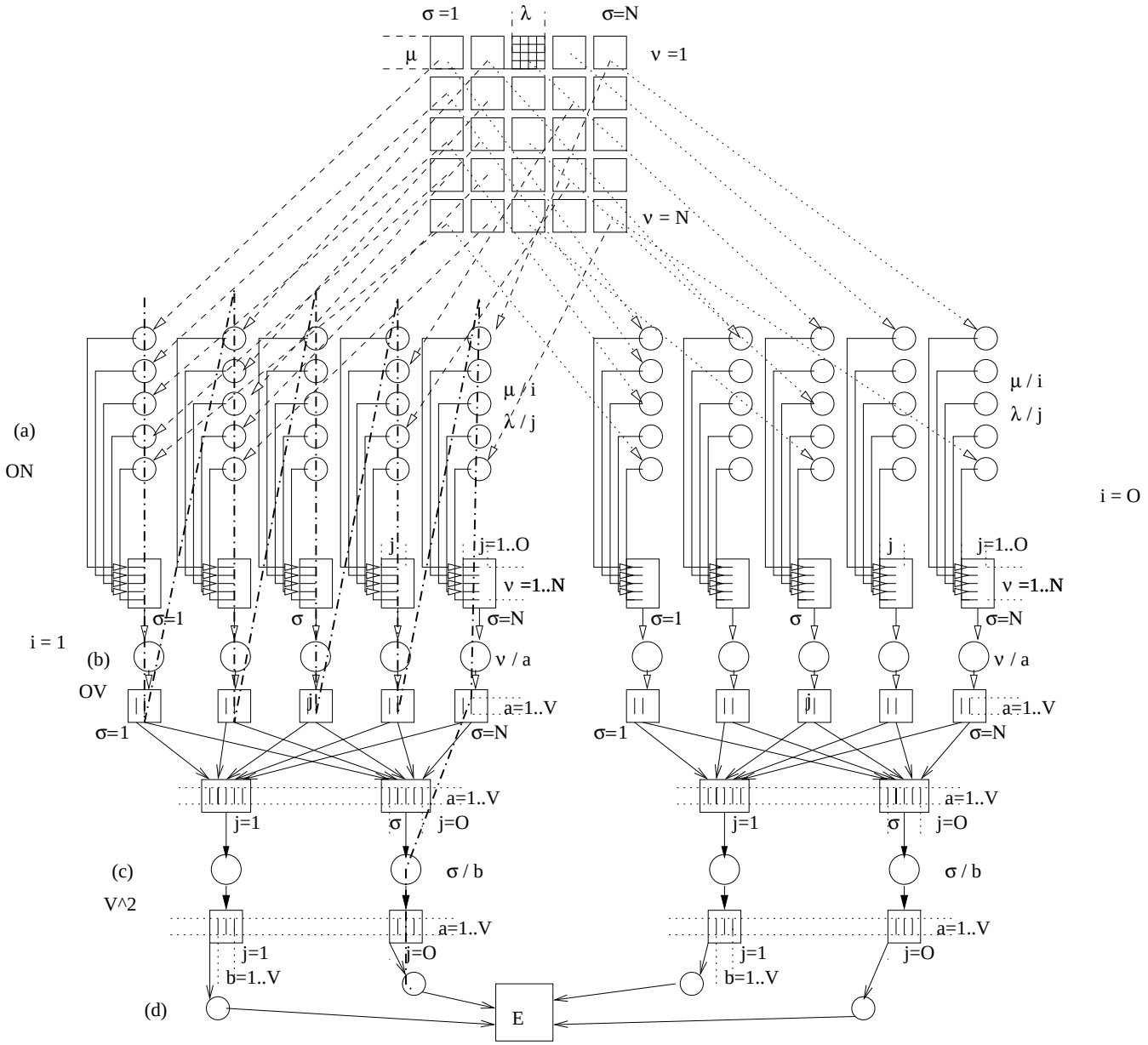


Figure 6.7 *Ordonnancement du GFD de MP2 désynchronisé par renumérotation des transformations. Les tâches sont exécutées en suivant le trait gras en pointillés pour calculer une contribution à E. L'espace mémoire requis par chaque tâche des groupes (a), (b) et (c) apparaît dans la marge gauche.*

puis une tâche du groupe (b) ; mémoriser les O données calculées qui seront nécessaires à la tâche de (c), puis ordonnancer N autres tâches de (a) suivie de la tâche de (b), et ainsi de suite jusqu'à obtention des ON données en entrée de la tâche $\sigma \rightarrow b$ de (c) qui peut alors être exécutée avec un espace mémoire V^2. Enfin, la tâche (d) accumule le résultat dans E en temps V^2 et en espace 1.

On exécute ainsi les N^2 tâches de (a) et les N tâches de (b) pour obtenir une contribution à E. On doit donc dupliquer O fois les N^2 tâches de (a) et les N tâches de (b) (à i fixé). Le temps d'exécution de cette version de MP2 est donc finalement :

$$O\left(O(N^4 + N^3O + ON^2V) + ONV^2 + OV^2\right).$$

(on retrouve dans cette formule le O en facteur qui compte chaque valeur de $i = 1 \ldots O$ et à l'intérieur des parenthèses les O duplications des étapes (a) et (b) suivies des O tâches de (c) et des O tâches de (d).)

L'espace mémoire requis est lui égal à $\max(NO, V^2) = \max\left((O+V)O, V^2\right)$. Ceci est à rapprocher probablement de l'article de Taylor, [63] cité par Pople, qui évoque un calcul de MP2 en $\mathcal{O}(N^2)$ en mémoire.

En ne gardant que le terme dominant en O^2N^4 dans le nombre d'opérations (car $N = O + V$), on retrouve le surcoût des O duplications par rapport au coût ON^4 du calcul de Pople classique. Concrètement, certains calculs physiques qui voudraient simuler des ensembles à peu de particules (ce que O mesure) mais avec une grosse précision ($V >> O$) peuvent être favorables à cet ordonnancement qui est seulement quadratique en mémoire là où le calcul de Pople est cubique.

6.4.3 Conclusion : MP2 — ordonnancement parallèle efficace en mémoire

Nous avons proposé dans cette section deux graphes de flot de données modélisant des ordonnancements des calculs de MP2. Le premier, (6.4) correspond à l'ordre proposé par Pople. Le second, (6.5), est basé sur une autre numérotation des indices à transformer. Ces deux GFD ont des caractéristiques T_∞ (temps d'exécution parallèle sur un nombre non-borné de processeurs) et σ (nombre de points de synchronisations) qui les différencient selon la taille respective des paramètres physiques O et V. Nous avons rappelé le temps de calcul séquentiel ($T_1 = ON^4$ en première approximation) et l'espace mémoire ($S_1 = OVN$) requis par l'ordonnancement séquentiel de Pople sur le premier GFD. Nous avons proposé également un ordonnancement séquentiel du deuxième GFD qui aboutit à un temps d'exécution $T_1 = O^2N^4$ avec un espace mémoire $S_1 = \max(V^2, NO)$ qui est donc seulement quadratique. La table (6.1) résume ces grandeurs pour chaque GFD. En séquentiel uniquement, dans le cas (physiquement significatif) où $V \gg O$, on peut envisager le deuxième ordonnancement qui réduit l'espace mémoire d'un ordre de grandeur au prix de O recalculs. De plus, toujours dans ce cas, ce deuxième GFD présente des caractéristiques parallèles plus intéressantes (*cf.* σ et T_∞).

Ceci est d'autant plus intéressant que les algorithmes d'ordonnancement parallèle online du type de Async-Q présentés dans ce chapitre sont basés pour leurs performances

	T_∞	σ	T_1	S_1
GFD (6.4)	$N^3 + N^2V + 2ONV + OV^2$	OVN	ON^4	OVN
GFD (6.5)	$2N^3 + V^2$	OON	O^2N^4	$\max(V^2, ON)$

TAB. 6.1 *Grandeurs caractéristiques des deux GFD pour MP2 (temps d'exécution parallèle sur infinité de processeurs ; nombre de synchronisations ; temps séquentiel ; espace mémoire séquentiel) avec les ordonnancements proposés.*

sur l'ordonnancement séquentiel, puisque :

$$\begin{aligned} T_p &= \mathcal{O}(\frac{T_1}{p} + \frac{\sigma \log(pT_\infty)}{p} + T_\infty \log(pT_\infty)), \\ S_p &\leq S_1 + \mathcal{O}(pT_\infty \log(pT_\infty)). \end{aligned}$$

Avec ce type d'algorithme dynamique basé sur l'ordonnancement du GFD (6.5) on garantit donc à la fois un temps parallèle meilleur qu'avec l'ordonnancement de Pople grâce à un σ et un T_∞ meilleurs, mais également un espace mémoire parallèle S_p qui reste à un facteur additif proportionnel à pT_∞ de S_1. En ordre de grandeur donc, le programme parallèle garde un volume mémoire total (*i.e.* la somme des espaces mémoires requis sur chaque nœud par le programme) égal à celui du programme séquentiel. Dans le cas d'une machine homogène par exemple on aura donc un espace mémoire de l'ordre de S_1/p requis par nœud.

En conclusion de cette étude théorique nous illustrons donc sur l'exemple de MP2 comment l'étude du flot de données permet de calculer des ordonnancements séquentiels efficaces en temps et en mémoire, qui à leur tour débouchent sur des ordonnancements parallèles efficaces. Rien n'empêche donc un environnement tel ATHAPASCAN d'effectuer ce genre d'analyse du GFD à l'exécution grâce aux heuristiques citées, de déduire un ordonnancement séquentiel efficace en temps et en mémoire et finalement d'appliquer un ordonnancement parallèle tel Async-Q basé sur cet ordonnancement séquentiel.

7

Bilan et perspectives : de l'application au programme parallèle performant

L'objectif premier du travail présenté était, partant des besoins en calcul haute performance exprimés dans la communauté de la physique/chimie quantique, d'étudier quels outils de parallélisme permettaient d'apporter des réponses, et dans quelle mesure ces réponses satisfaisaient les demandes. Dans le même temps les problématiques propres au calcul parallèle ont été abordées. Ce chapitre résume les principales thématiques adressées dans ce document et les contributions apportées, avant de présenter quelques perspectives possibles.

Bilan des travaux effectués

Le point de départ du travail applicatif est la compréhension des domaines concernés, ici la mécanique et la chimie quantique. Deux calculs types, le problème d'une particule piégée dans une boîte quantique (BQ) et MP2, la méthode des perturbations de Møller-Plesset, ont été plus particulièrement traités. L'approche suivie a été dans les deux cas de proposer une reformulation algorithmique des méthodes de résolution, qui semblait particulièrement adaptée aux problèmes : emploi de méthodes itératives de calcul de valeurs propres (IRAM) dans le cas de (BQ) ; reformulation matricielle (par blocs) dans le cas des boucles de transformations de MP2. Ces solutions algorithmiques ont été guidées à chaque fois par des critères d'efficacité en temps (calcul par bloc permettant l'emploi des

BLAS, par exemple), en mémoire (méthode itérative adaptée au calcul sur des matrices creuses pour (BQ)), ainsi que pour leur degré de parallélisme.

Plusieurs solutions existantes en terme d'environnements de programmation parallèle et de leur adéquation aux applications traitées ont été présentées et comparées. La performance de ces environnements a été montrée tout particulièrement dans le cas de MPI sur l'exemple du benchmark Linpack, pour une architecture de grappe de PC qui semble être l'un des types de machine parallèle émergeant dans la communauté.

Les limitations de ces environnements ont cependant été également mises à jour. Elles sont de plusieurs ordres :

- **portabilité du code produit.** Les hautes performances obtenues par un codage très spécifique à un environnement et/ou à une machine ne sont souvent pas portables.
- **expressivité du parallélisme de l'algorithme cible.** Que ce soit dans le cadre algorithmique pur ou dans le cadre de la programmation d'une application, le surcoût de gestion du parallélisme, par exemple dans un paradigme de type "passage de message", empêche le programmeur de coder son algorithme aussi naturellement qu'il le conçoit. On retrouve là la problématique des langages de programmation séquentielle qui doivent donner au programmeur les bons outils pour exprimer son algorithmique.

Pour le premier point, cette thèse focalise sur l'ordonnancement comme outil permettant d'adapter le code, de façon transparente pour l'algorithme (et donc pour le programmeur), à la machine cible. L'ordonnancement calcule un site et une date d'exécution pour chaque tâche de l'algorithme en fonction des caractéristiques de la machine et des objectifs d'efficacité du programmeur qui peuvent être par exemple le temps d'exécution ou la mémoire requis par l'exécution.

Cependant, pour permettre le calcul d'un tel ordonnancement, il est nécessaire de spécifier l'algorithme à coder. Une proposition pour décrire l'algorithme de façon à l'ordonnancer efficacement (ce qui répond en même temps au second point) est de le caractériser par son flot de données, lequel peut être modélisé par un graphe bipartite qui représente les tâches de calcul, les données et les accès des unes sur les autres : le graphe de flot de données (GFD). Un environnement de programmation parallèle, ATHAPASCAN , est également introduit, qui permet la description du flot de données *via* le graphe, sa construction au cours de l'exécution du programme et son ordonnancement selon les stratégies choisies par le programmeur.

Pour ordonnancer le graphe selon l'objectif de performance choisi, différents algorithmes peuvent être utilisés. Nous rappelons ici quels algorithmes d'ordonnancement permettent l'obtention de programmes hautement performants en terme de temps de calcul, et les illustrons sur des programmes ATHAPASCAN : calcul de factorisation de Choleski et méthode d'Arnoldi avec redémarrage, laquelle peut être employée pour (BQ).

Dans le cas important et fréquent en chimie quantique où la mémoire est une motivation clé pour le passage au parallélisme il faut que l'ordonnancement calculé réduise

au maximum l'espace utilisé. Une contribution de ce travail est la proposition d'annotation du Graphe de Flot de Données utilisé dans ATHAPASCAN par la mémoire utilisée. La construction concrète du GFD par ATHAPASCAN permet alors de calculer des ordonnancements minimaux en mémoire : nous montrons que ce calcul est théoriquement NP-complet mais des heuristiques existent qui permettent d'être à un facteur $\log^2(n)$ de l'optimal (n étant le nombre de noeuds du GFD). Cela peut suffire, ou fournir une borne qui permet d'envisager une recherche exhaustive de la solution optimale avec élagage de l'arbre grâce à la solution de l'heuristique. Dans le cas particulier du calcul MP2 nous montrons enfin comment obtenir un ordonnancement des calculs efficace en temps et en mémoire.

Perspectives

Le travail exposé a donc regardé un spectre de problèmes assez large et pluridisciplinaire. Les perspectives ouvertes sont à la mesure de ce spectre et il reste bon nombre de pistes à explorer au moment de conclure ce document.

- Au niveau d'ATHAPASCAN , l'étape suivante serait l'implémentation dans l'environnement de l'annotation du GFD pour tenir compte de la mémoire, et l'écriture des ordonnanceurs adaptés. La possibilité offerte par l'environnement d'écrire ceux-ci de façon modulaire devrait simplifier la dernière partie de ce travail de programmation. Toute une série de tests seraient alors envisageables.
- Pour l'ordonnancement il reste à imaginer les heuristiques qui permettraient de prendre en compte des modèles plus complets des machines parallèles, par exemple pour prendre en compte des machines à hiérarchie de parallélisme telles I-Cluster avec ses 2 niveaux de communication (intra- et extra-switches), des machines couplées par Internet, ou encore des grappes de SMP. Cela en soi est déjà le sujet de recherches entières.
- En ce qui concerne le traitement numérique des algorithmes de mécanique et chimie quantique regardés, plusieurs axes de travail auraient pu être suivis également :
  - sur (BQ), de nombreuses technique numériques sont l'objet de recherches sur le type de méthodes ici proposées : préconditionnement pour accélérer la convergence ; comparaison entre les diverses stratégies de redémarrage possibles ; convergence de l'algorithme, selon le nombre de valeurs propres recherchées et les propriétés du Hamiltonien à diagonaliser (ou, en terme physiques : existence ou non d'un premier état excité dans certains puits de potentiel) ;
  - sur (MP2), le lien avec le calcul tensoriel et la façon dont il permet d'exprimer les boucles reste à faire.
- Un autre objectif naturel serait de "boucler la boucle" en revenant des problèmes-types étudiés à une application complète, dans toute sa complexité. Le travail à faire est important : les codes complets utilisés en chimie quantique, par exemple, posent des problèmes d'accès aux données irréguliers, d'utilisation de symétries fortes, qui peuvent fortement compliquer un algorithme "épuré" lors d'une étude théorique —

qui reste néanmoins un préalable indispensable et en elle-même complexe, pour une programmation efficace.

L'ensemble de ce travail a finalement confronté les attentes, pour certains problèmes de mécanique et chimie quantique, aux solutions existantes et aux pistes de recherche en parallélisme sur le calcul haute-performance, afin de tenter de combler les manques de ce qui est déjà en cours d'utilisation. L'approche transversale a certainement accumulé les problèmes des différentes disciplines mais a permis également de montrer quelles solutions peuvent se dégager pour la programmation performante d'applications numériques pour la mécanique quantique.

Annexe A

Évaluation de performances de I-Cluster pour l'entrée au Top 500 : optimisation du benchmark Linpack

Nous présentons dans cette annexe un résumé plus technique du travail fourni pour optimiser le benchmark Linpack, qui sert à établir le classement des 500 super-calculateurs les plus puissants au monde (sur ce type de calcul, bien sûr), sur la grappe I-Cluster du laboratoire ID-IMAG, développée en collaboration avec H.-P et l'INRIA Rhône-Alpes. Le rapport technique [56] reprend de façon plus détaillée cette section.

La première section introduit rapidement l'algorithme de pivot partiel implémenté dans le test Linpack. La section suivante détaille chaque type de paramètre qui a été étudié dans le travail : l'usage des Blas (testés sur plusieurs calculs types) ; les paramètres algorithmiques de Linpack (topologie de grille de processeurs, taille de matrice et de blocs, diffusion, etc. . .) ; l'influence de la topologie du réseau de I-Cluster. Enfin, la dernière section résume à la fois le calendrier des résultats mesurés et l'importance relative de chacun des paramètres, pour autant que ceux-ci puissent être considérés de façon autonome.

A.1 Benchmark Linpack

Le test Linpack effectue la résolution d'un système linéaire grâce à une factorisation LU par la méthode du pivot partiel (routine dgefa de Lapack). Le calcul est fait par blocs, dans sa version "right-looking" : la matrice M, de taille $n \times n$, à factoriser est découpée en blocs de taille $n_b \times n_b$. Une factorisation LU partielle de M peut s'écrire sous la forme :

$$PM = \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & 0 & I \end{pmatrix} \times \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & M'_{22} & M'_{23} \\ & M'_{32} & M'_{33} \end{pmatrix},$$

où L_{11} et U_{11} sont des matrices $n_b \times n_b$ et P est une matrice de permutation représentant le pivotage. Les "M'" indiquent des valeurs de M modifiées par l'exécution de l'algorithme.

Lorsque l'on avance la factorisation d'une étape, on calcule la prochaine colonne de blocs de L et la prochaine ligne de blocs de U tels que :

$$\begin{pmatrix} I & \\ & P_2 \end{pmatrix} PM = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{pmatrix} \times \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & M'_{33} \end{pmatrix}.$$

(P_2 est une matrice de permutation d'ordre $n - n_b$.) En comparant ces deux équations et en identifiant les blocs, on voit que l'on commence par effectuer la factorisation de la première colonne de blocs de la sous-matrice courante :

$$P_2 \begin{pmatrix} M'_{22} \\ M'_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}. \tag{A.1}$$

On effectue alors le pivotage de la partie de la sous-matrice à droite de cette colonne :

$$\begin{pmatrix} M'_{23} \\ M'_{33} \end{pmatrix} \leftarrow P_2 \times \begin{pmatrix} M'_{23} \\ M'_{33} \end{pmatrix}, \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \leftarrow P_2 \times \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}. \tag{A.2}$$

Ensuite on résout le système triangulaire (calcul du pivot) :

$$U_{23} = L_{22}^{-1} M'_{23}. \tag{A.3}$$

Enfin on met à jour la partie M'_{33} :

$$M'_{33} \leftarrow M'_{33} - L_{32} U_{23}.$$

On est ensuite ramené à la même forme de factorisation et on peut appliquer ces étapes récursivement.

Pour paralléliser ce calcul, le test Linpack distribue les blocs de façon cyclique sur une grille de $p \times q$ processeurs. Ceci implique qu'à chaque itération alternent les phases suivantes :

(i). une colonne de processus (qui possèdent les blocs de la colonne du milieu de U) effectuent la factorisation (A.1). Ceci constitue une section critique de l'algorithme ;

(ii). cette colonne de processus doit ensuite diffuser à la fois le pivot calculé par (A.3) et les pivotages à effectuer sur les lignes. Cette diffusion, pour chaque bloc, n'est nécessaire que vers les processus qui sont sur la ligne (dans la grille de processeurs) du bloc puisque les opérations de pivotage et de mise à jour ne se font que sur les lignes ;

(iii). chaque processeur ayant reçu le pivot peut mettre à jour ses blocs en effectuant (A.1).

A.2 Paramètres à optimiser

De très nombreux paramètres sont à optimiser sur ce benchmark. Certains portent sur les opérations (séquentielles) de base : c'est le cas de tout ce qui concerne les calculs à effectuer sur des blocs, qui repose essentiellement sur les routines des Blas. D'autres sont liés à l'algorithmique de Linpack. Il faut bien sûr prendre en compte le réseau de la machine cible. Enfin des réglages au niveau du système doivent être effectués sur ce benchmark.

A.2.1 Blas

Chaque noeud de I-Cluster est un processeur Pentium III cadencé à 733 Mhz, disposant de 256 Mo de mémoire. À raison d'une opération flottante par cycle, on s'attend donc à une puissance de crête théorique de 733 MFlops/s par noeud.

Nous avons effectué plusieurs tests des Blas sur un nœud de I-Cluster : produit matriciel (dgemm) pour petite matrice (tenant dans le cache) et "grosse" matrice (de taille de l'ordre de plusieurs milliers) ; résolution de système linéaire (linpack-1000), afin de se placer dans les conditions identiques au travail exigé par le benchmark (*cf.* calcul du pivot).

4 versions des Blas ont été testées : deux basées sur Atlas [67], l'une sur la librairie MKL d'Intel, et une dernière sur des Blas spécialisées pour Pentium III par Greg Henry, également d'Intel.

A.2.1.1 Produit de matrices

Les figures A.1, A.2 et A.3 présentent les performances mesurées sur des produits de matrices de tailles diverses, avec Atlas (version 3.2.0).

On atteint les 630 MFlops/s sur un produit de matrices 102×102, soit 95 $\%$ de la puissance de crête théorique. Ce résultat est comparable aux benchmarks des universités du Maine et de Lunenfled qui disposent de clusters de PIII sous Linux. Une fois la taille du cache (256 Ko) dépassée, on reste aux alentours de 500 MFlops/s, ce qui est encore satisfaisant.

On observe un phénomène de "plateaux" (*cf.* A.3) dans les temps pour des groupes de tailles voisines : $dgemm$ découpe la matrice en blocs de taille optimale en fonction du processeur et le temps de calcul d'un produit matriciel est donc fonction de la taille de la matrice modulo la taille des blocs. Quand on effectue le quotient du nombre d'opérations par le temps, ce phénomène est atténué au fur et à mesure que le nombre d'opérations augmente, d'où ces oscillations amorties que l'on observe.

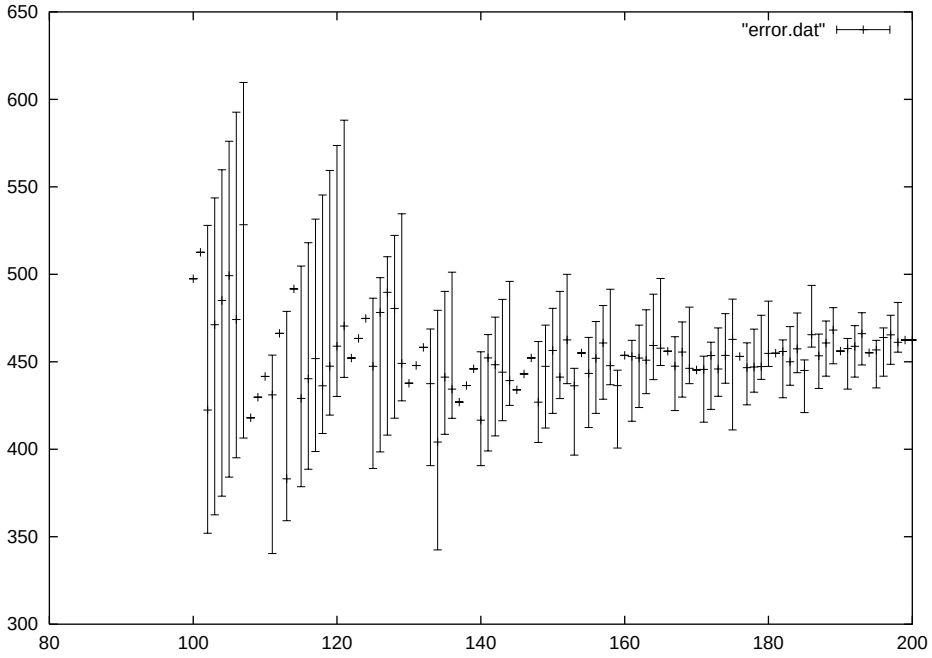


Figure A.1 *Puissance obtenue sur un produit matriciel, vs. taille de la matrice (100...200). Compilation avec Atlas.*

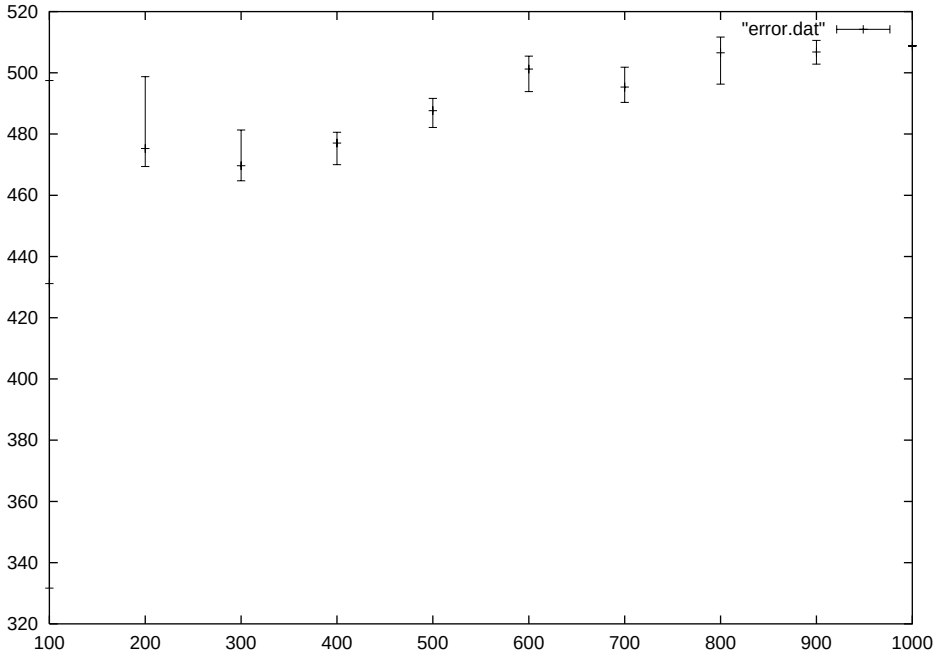


Figure A.2 *Puissance obtenue sur un produit matriciel, vs. taille de la matrice (100...1000). Compilation avec Atlas.*

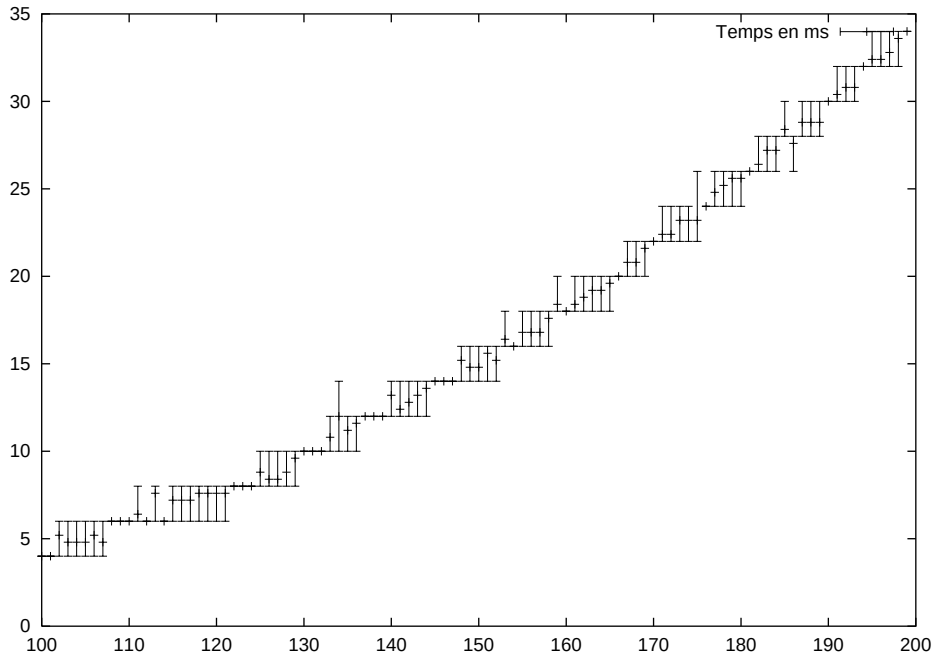


Figure A.3 *Temps de calcul (ms) d'un produit matriciel Blas. Compilation avec Atlas.*

A.2.1.2 Linpack1000 et utilisation de Lapack (version Atlas)

Le deuxième test séquentiel effectué est le benchmark Linpack1000, qui résout de manière itérative une série de systèmes aléatoires de taille 1000×1000.

En utilisant la librairie lapack, basée sur Atlas, on aboutit, sur 100 tests, à une puissance moyenne de 417 MFlops/s (min = 407.7 MFlops/s, max = 420.5 MFlops/s). Cette puissance, vu la taille du problème traité, est comparable aux 500 MFlops/s obtenus sur le test du produit matriciel.

De plus, on trouve sur le site de linpack (www.netlib.org/benchmark) deux résultats pour des processeurs comparables : le Pentium II Xeon 450 MHz affiche un résultat de 295 MFlops/s sur linpack1000 ; le Sun Ultra Sparc II à 336 MHz donne lui 461 MFlops/s sur ce test.

Nous avons conclu de ces tests séquentiels :
– que l'on peut attendre une utilisation assez performante de chaque noeud de la grappe I-Cluster, aux alentours de 500 MFlops/s sur de gros problèmes, voire encore plus près de la performance de crête sur des problèmes plus petits.
– que les librairies Blas et Lapack compilées via Atlas sur la grappe sont fonctionnelles pour atteindre ces performances simplement.

	Moyenne	écart-type
Henry	499.38 MFlop/s	6.52 MFlop/s
Atlas 3.2.0	493.96 MFlop/s	1.8 MFlop/s
Atlas 3.2.1	478.3 MFlop/s	1.7 MFlop/s
MKL	439.9 MFlop/s	2.1 MFlop/s

TAB. A.1 *Puissance CPU sur un produit matriciel de taille 2000, pour 4 librairies optimisées pour Pentium III.*

A.2.1.3 Comparaison avec les librairies Intel

Après discussion avec Greg Henry (http ://www.cs.utk.edu/ ghenry/hidden/h4GMHy2/blas.html, http ://www.cs.utk.edu/ ghenry/) et Bruce Greer qui font partie de l'équipe d'Intel développant la librairie MKL (http ://developer.intel.com/software/products/mkl/index.htm), nous avons choisi de focaliser nos tests sur le produit de grosses matrices, de taille proche de celles manipulées par le test Linpack. (transa=transb='N', M=13000, N=2000, K=96, alpha=-1.0, LDA=LDC=13000, LDB=96, beta=1.0 pour dgemm.)

C'est sur cette taille de matrices et sur dgemm que nous avons comparé la librairie MKL d'Intel, pour Pentium III, avec Atlas. Nous avons également testé une version d'Atlas linkée avec la routine spécifique de produit matriciel codée par Greg Henry.

Les exécutions ont été faites 20 fois, sous Linux noyau 2.4.2. Les résultats sont inclus dans le tableau A.1.

Nous avons retenu de ce test que la version optimale semblait être la combinaison Atlas 3.2.0 + Greg Henry. C'est à lier à l'expérience menée à l'université de Chemnitz (Matthias Pester m.pester@mathematik.tu-chemnitz.de), où ils ont gagné 17 GFlops/s avec le dgemm de Henry, par rapport à Atlas 3β.

De plus des problèmes numériques avec MKL, sous le noyau 2.4.2, sont apparus qui nous ont conduits à écarter cette librairie.

En ce qui concerne le test Linpack-1000, le tableau A.2 présente nos conclusions.

Là aussi Atlas l'a emporté face aux librairies MKL. Finalement nous avons aussi testé nos librairies directement sur une exécution séquentielle du benchmark Linpack parallèle (en prenant tous les paramètres spécifiques à ce code égaux par ailleurs). Les résultats sont allés de 943 à 952 secondes, sur 9 exécutions, avec Atlas/Henry, vs. 1027 à 1044 secondes pour Atlas 3.2.0 (12 exécutions). Cela a également contribué à notre choix de garder cette combinaison Atlas + Henry. Remarquons enfin que ces temps d'exécutions signifient environ 500 MFlop/s pour l'exécution séquentielle du benchmark Linpack sur un nœud de I-cluster. Cela est à comparer avec les quelques 360 MFlops/s par nœud que nos meilleures exécutions parallèles ont permis de dégager.

Puissance CPU - Atlas 3.2.0			
	Min	Max	Moyenne
1st run	417.9 MFlop/s	425.9 MFlop/s	425.6 MFlop/s
2nd run	415.3 MFlop/s	428.6 MFlop/s	425.6 MFlop/s
Puissance CPU - Intel MKL			
	Min	Max	Moyenne
1st run	326.2 MFlop/s	336 MFlop/s	333.6 MFlop/s
2nd run	327.8 MFlop/s	334.3 MFlop/s	333.8 MFlop/s

TAB. A.2 *Puissance CPU sur Linpack-1000, pour 2 librairies optimisées pour Pentium III.*

A.2.2 Paramètres algorithmiques

L'algorithme parallèle même implémenté dans le benchmark comporte de nombreux paramètres à régler selon la machine cible. De plus ils sont parfois interdépendants. Parmi ceux-ci figurent :

La topologie de la grille $p \times q$ de processeurs. Le "tuning guide" de Linpack conseille d'utiliser une grille carrée. L'idée est de faire un compromis entre le nombre de lignes (le calcul de l'indice du pivot impose de faire ce calcul de façon séquentielle selon une colonne et un nombre trop grand de lignes va donc séquentialiser le programme) et le nombre de colonnes (les diffusions du pivot calculé vont se faire selon une ligne et plus on a de colonnes, plus la diffusion va être coûteuse). Greg Henry rapporte au contraire de bons résultats obtenus avec une grille 4 fois plus large que profonde. Nous avons dû tester chaque configuration possible selon le nombre de noeuds utilisés. Bien sûr, selon cette topologie, l'algorithme de diffusion également a dû être adapté.

- sur 32 noeuds : on peut faire des grilles 8×4, 4×8, 16×2, 2×16. Les 4 furent testées. L'avantage de ce petit nombre de noeuds est de pouvoir utiliser 32 noeuds répartis sur un seul commutateur d'I-Cluster, en limitant ainsi l'hétérogénéité du réseau. Plus de 180 expériences ont été faites sur les grilles 8×4. La meilleure configuration a donné 11 GFlops/s. Sur une grille 16×2 le meilleur résultat fut de 8.5 GFlops/s. Sur une grille 2×16 la meilleure performance a été 9.7 GFlops/s.
- sur 96 et 98 noeuds : $96 = 8 \times 12$. Au vu des moins bonnes performances sur les grilles 16×2 et 2×16 nous avons privilégié des grilles carrées. Nous n'avons donc pas testé, par exemple 4×24. Nous avons obtenu 34 GFlops/s sur une grille 8×12, et 35.1 GFlops/s sur $98 = 7 \times 14$ noeuds (grille pourtant nettement moins "carrée"). Cela représente de l'ordre de 360 MFlops/s par nœud. Le passage de 32 à 100 noeuds a ainsi donné du crédit aux espoirs de bon passage à l'échelle de Linpack sur I-Cluster.
- sur 215 noeuds : on peut uniquement faire une grille 43×5. Cette configuration, à l'origine celle qui devait être la version définitive de I-Cluster, était donc très défavorable au test Linpack. De fait jamais nous n'avons réussi à mesurer de bonnes per-

Grille	Nombre d'exécutions	Meilleure perf.	Paramètres
15×14	33	67.9	N=76020, Nb=141, W03L2R8
14×15	10	71.8 (1) / 76.4	N=76935, Nb=141, W03L2R8
10×21	5	74.4	N=76500, Nb=141, W03L2R8

TAB. A.3 *Résultats des exécutions de Linpack sur 210 noeuds.*

formances. Au mieux nous avons eu 32.4 GFlops/s sur 43×5 noeuds, et 55 GFlops/s sur 5×43, soit 255 MFlops/s par nœud (soit 50% de la performance optimale séquentielle mesurée sur linpack, sur un nœud). Cela posa également un problème expérimental car de telles exécutions (la matrice était de l'ordre de 77000×77000) prenait de 1h30 à 2h38 pour ces performances : la machine était donc entièrement occupée, longtemps, pour produire des mauvais résultats.
On note la différence de performances entre la grille 5×43 et la grille 43×5. Cela est dû à la bonne utilisation des commutateurs de I-Cluster dans la configuration favorable. Ce point sera reprécisé plus bas.

– sur 210 noeuds : nous avons longtemps obtenu nos meilleurs résultats sur 210 noeuds, car $210 = 14 \times 15 = 15 \times 14$. C'est cette configuration qui nous a permis de comprendre l'influence de l'algorithme de diffusion en fonction des commutateurs. Nous avons effectué 10 exécutions sur 14×15 noeuds ; 33 sur 15×14 ; 5 sur 10×21 ; 1 sur 7×30 ; et 6 sur 30×7. La table A.3 montre les meilleures performances obtenues.
On note à nouveau que la meilleure performance obtenue, 76.4 GFlops/s sur 210 noeuds, donne à peu près 360 MFlops/s par nœud.
– sur 225 noeuds : l'idée fut donc de revenir à une grille carrée et il fallut ajouter 10 noeuds supplémentaires pour cela aux 215 prévus initialement pour la configuration de I-Cluster. $225 = 15 \times 15$ et nous attendions autour de 81 GFlops/s, à raison de 360 MFlops/s par nœud. Une première tentative fut de prendre des PC standards du laboratoire ID, de marque différente de ceux du Cluster, mais basés sur des Pentium III, pour compléter I-Cluster. L'expérience fut désastreuse (15 GFlops/s mesurés), à cause semble-t-il d'un léger manque de mémoire des nouveaux noeuds, qui ont swappé au début de l'exécution et ainsi bloqué tout le calcul.
Par contre une fois les 10 noeuds e-vectra reçus, identiques aux 215 autres, nous avons mesuré dès les premières expériences 81.4 GFlops/s, sur 225 noeuds, avec une matrice de taille N=80370. C'est cette mesure qui a servi de référence pour le top-500.

Taille de matrice n et taille de blocs n_b Ce sont les paramètres avec lesquels nous avons travaillé de façon très empirique. Les conseils généraux sont de prendre la matrice la plus grosse possible. Nous avons obtenu nos meilleures performances avec une matrice de taille allant de 200 Moctets (par nœud) à 220 Moctets (la mémoire requise par nœud étant bien sûr $M(N) = \frac{8n^2}{pq}$, à raison de 8 octets par réel double précision). Chaque nœud dispose de 256 Moctets de mémoire vive. Il semble que pour des matrices plus grandes

au moins une partie des noeuds commençaient à swapper au cours du test. Cela peut être dû aux buffers requis par MPI, et/ou à ceux réservés par le système. La taille de matrice traitée semble compatible avec les expériences effectuées sur d'autres machines.

En ce qui concerne la taille n_b des blocs, théoriquement nous aurions dû travailler ainsi :

(i). déterminer la taille $M \times N$ de la matrice traitée sur un nœud (sachant qu'elle prend autour de 210 Moctets) ;

(ii). déterminer expérimentalement la meilleure valeur de K pour l'opération dgemm des Blas, sur des produits de matrices $M \times K$ et $K \times N$ (*cf.* la section sur les tests des Blas ci-dessus) ;

(iii). fixer $n_b = K$ et n en fonction de n_b.

Cette dernière étape aurait dû en toute rigueur suivre les règles suivantes : pn_b divise n et qn_b divise n ; c'est-à-dire $(\gcd(p,q) \times n_b)$ divise n.

L'ennui est que cette dernière contrainte est très forte : par exemple avec des paramètres réalistes, sur 210 noeuds, $p = 14, q = 15$, $\gcd(p,q) = 210$ et si on fixe $n_b = 141$ selon les tests des Blas, il faut choisir N comme multiple de $29610 = 210 \times 141$, ce qui ne laisse que peu de choix si l'on veut que la matrice tienne en mémoire, car avec $M(n) = 210Mb$ il vient $n \approx 76000$ sur 210 noeuds. Finalement on s'aperçoit que l'on peut uniquement prendre $N = 2 \times 29610 = 59220$ si l'on veut suivre toutes ces contraintes... et cette matrice nettement plus petite que $n \approx 76000$ ne va pas donner une performance optimale sur 210 noeuds.

Nous n'avons donc pas réellement suivi ce schéma, même si certaines étapes furent réalisées (mesure sur dgemm des meilleures tailles de blocs), pour les raisons suivantes :

(i). Utilisant 4 ou 5 versions des Blas, avec chacune des valeurs différentes pour les valeurs optimales de K et décider sur ces tests quelle version aurait été la meilleure, aurait pris beaucoup de temps pour un résultat peu fiable car d'autres facteurs interviennent dans le calcul total qui rendent l'analyse plus complexe que juste le calcul de la taille optimale de bloc pour dgemm ;

(ii). même en fixant $n_b = K$ de cette façon, les contraintes sur n, p et q auraient été trop grandes et nous auraient conduits à ne pas tester certaines valeurs de n et n_b qui se sont révélées correctes ;

(iii). finalement, en testant directement sur les exécutions de Linpack différentes valeurs de n et n_b (ce sont, chronologiquement, les derniers paramètres à avoir été fixés), parfois respectant le critère de division, parfois non, nous avons obtenu les meilleures performances sur des paramètres ne respectant pas ce critère (par exemple $n = 76230$ et $n_b = 141$ sur une grille 15×14).

Algorithme de diffusion Le test Linpack fournit 6 algorithmes différents pour l'algorithme de diffusion utilisé, qui sont des variantes autour de chaînes de "send/receive"

(chaîne linéaire, ou arbre binaire tronqué à une profondeur donnée...). Chacun a été testé sur presque toutes les configurations. Ce fut en réalité le paramètre clé pour gagner (ou perdre) des ordres de grandeur sur la performance. En particulier le passage à l'échelle de 98 à 210 noeuds impliqua un changement radical d'algorithme de diffusion.

Dans un premier temps nous avons testé les diffusions sur 32 noeuds, pour $n = 25000$. Sur ces exécutions, les diffusions 1, 3 et 5 sont ressortis comme étant les plus efficaces. Malheureusement ces expériences, faites au début du travail, ne prirent pas en compte l'influence des switches (commutateurs) de I-Cluster (*cf.* aussi la section ci-dessous sur la topologie du réseau). Les expériences ultérieures montrèrent l'influence prépondérante de ceux-ci sur les performances. D'autres mesures effectuées sur 32 noeuds regroupés sur un unique switch donnèrent comme meilleurs les diffusions 3 et 5. (À peu près 940 secondes sur $n = 25000$ et $n_b = 141$. Avec l'algorithme de diffusion 4 on obtenait quelques 955 secondes.)

Nous avons obtenu les meilleures performances sur 96 et 98 noeuds avec la diffusion de type 5.

En passant à 200 noeuds et plus, les résultats furent radicalement différents : les diffusions 2 et 3 ressortirent nettement plus (autour de 67 GFlops/s) que le 5 qui restait autour de 50 GFlops/s, les autres paramètres étant égaux par ailleurs ($n = 76230, n_b = 141, W0 * L2R8$, sur $210 = 14 \times 15$ noeuds). Ce choix fut confirmé quand les noeuds furent mieux alloués sur les switches. Le "blocage" à 50 GFlops/s, pendant plusieurs jours, à cause d'un changement nécessaire d'algorithme de diffusion, fut l'un des obstacles majeurs de ce travail.

Au vu de l'influence de cette diffusion nous avons également testé des algorithmes écrits "à la main" en MPI afin de les adapter à I-Cluster. Aucun n'apporta d'amélioration significative aux algorithmes pré-conçus dans le test Linpack.

Autres paramètres : La plupart des autres paramètres de Linpack, de nature séquentielle, furent fixés lors des tests intensifs sur 32 noeuds. De façon surprenante, les valeurs optimales n'ont pas toujours été celles préconisées par le "tuning guide".

- Depth (nombre des colonnes de blocs factorisées à l'étape A.1 avant de les diffuser) : sur 32 noeuds, Depth=0 a donné de 10.6 à 10.8 GFlops/s selon la diffusion, et de 10.2 à 10.6 pour Depth=1. Sur 210=15x14 noeuds, Depth=1 a donné 62.9 GFlops/s ($W13L2R8, n = 76230, n_b = 141$), vs. 76.4 GFlops/s avec les mêmes paramètres et Depth=0. Nous avons donc conclu que Depth=0 est la meilleure valeur sur I-Cluster.
- L2R8 : sur 32 noeuds le jeu de paramètres qui donna les meilleurs résultats fut : Rfact = Left (algorithme de factorisation récursive (séquentielle) de la colonne de blocs) ; PNBMINs = 8 (critère d'arrêt de la récursion de la factorisation) ; Fact = R (algorithme de factorisation utilisé une fois la récursion terminée) ; NDivs=2 (la colonne de blocs va être récursivement découpée en NDivs colonnes).

A.2.3 Topologie du réseau de I-Cluster

En dessous de la grille virtuelle utilisée par Linpack, on trouve bien sûr une allocation physique des processeurs de I-Cluster sur une topologie de réseau spécifique. Dans la configuration finale, les 225 noeuds sont répartis sur 5 commutateurs, chacun regroupant donc 45 noeuds.

La prise en compte de cette hiérarchie de communications fut cruciale pour gagner encore quelques GFlops/s sur Linpack. L'évolution de cette topologie fut d'ailleurs conditionnée par nos mesures : la machine a été adaptée au programme.

Durant les 10 premiers jours de tests sur 215 noeuds, ceux-ci étaient en effet répartis comme suit : les 3 premiers commutateurs hébergeaient chacun 40 noeuds. Le quatrième en avait 48 et le dernier 47. Ainsi nous pouvions utiliser 32 noeuds placés sur un unique switch. Par contre pour faire des tests plus gros, de taille réaliste, soit nous devions nous restreindre à utiliser 40 nœuds par switch, soit 200 noeuds au maximum, pour le mapping de la grille virtuelle ; soit nous déséquilibrions les schémas de communications par grille en utilisant, pour quelques calculs, des processeurs placés sur un switch différent de ceux de la "ligne" avec laquelle ils communiquaient. De fait, les meilleures performances furent longtemps obtenues avec la première approche, donc sur $5 \times 40 = 200$ noeuds, pour une grille adaptée de 10×20 noeuds qui a donné 65.7 GFlops/s en minimisant les communications trans-switches. Il a juste fallu spécifier à MPI une liste de processeurs ordonnée de façon à allouer les noeuds d'une manière qui respecte la topologie des switches.

Nous avons quand même réalisé 70 GFlops/s sur une grille de 14×15 noeuds, en ayant donc des communications trans-switches, mais en adaptant l'algorithme de diffusion pour en tenir compte.

Néanmoins le pas décisif fut l'ajout de 3 cartes sur les 3 switches sous-dimensionnés à 40 noeuds, afin de pouvoir accueillir jusqu'à 47 noeuds sur chaque switch. Ainsi nous avons pu déplacer les noeuds de façon à en avoir 43 par switch (soit 215 noeuds au total — puis 225 à terme). Nous avons donc pu faire des mesures sur des grilles de $215 = 15 \times 14 = 5 \times 43$ processeurs, optimales sur nos switches, à raison donc de 3×14 lignes de la grille virtuelle par switch. Le gain de cette manipulation de la topologie fut immédiat : nous sommes passés de 70 GFlops/s à 76.4 GFlops/s sur 215 noeuds. Cette performance fut la meilleure obtenue jusqu'à réception des 10 noeuds e-vectra supplémentaires pour compléter la grappe à 225 noeuds.

Enfin, deux architectures furent testées pour les liaisons entre les switches : une connexion sous forme d'étoile (graphe complet) entre les 5 switches avec des liens à 2 Gbits/s ; et une interconnexion sous forme d'anneau. De façon surprenante nous n'avons pas noté d'amélioration significative avec la formation en étoile.

Janvier 2001	: 28.4 GFlops/s sur 96 noeuds, $n = 50004$ (303 MFlop/s par nœud).
Lundi 19 mars	: début des mesures pour le top-500.
Jeudi 22 mars	: optimisations et nouvelles mesures sur 96 / 98 noeuds : 35 GFlops/s.
. . . jusqu'au 23 mars	: mesures sur 32 noeuds.
Lundi 26 mars	: Linux 2.4.2 et nouveau pilote.
Vendredi 30 mars	: installation de 20 nouveaux noeuds.
Lundi 2 avril	: installation du cluster avec 215 noeuds.
Dimanche 08 avril	: 67 GFlops/s sur 210 noeuds.
Lundi 09 avril	: 70 GFlops/s sur 210 noeuds.
Mardi 10 avril	: 71.8 GFlops/s avec un bon mapping des 210 noeuds sur les switches.
Mercredi 11 avril	: 43 noeuds par switch.
Jeudi 12 avril	: Meilleure perf. sur 210 noeuds (76.4 GFlops/s on 210 nodes).
Vendredi 13 avril	: échecs sur 215 et 225 noeuds (15 GFlops/s)
Dimanche 15 avril	: Soumission.
Mercredi 18 avril	: réception de 10 noeuds e-Vectra, installation de 225 noeuds sur I-Cluster, mesure de 81.4 GFlops/s.

TAB. A.4 *Chronologie des résultats sur le benchmark Linpack.*

A.2.4 Pilotes et noyau Linux

Nous avons utilisé deux pilotes pour les cartes réseau 3Com de I-Cluster. Chaque pilote a été installé et testé sous Linux 2.2.17 et Linux 2.4.2. Les améliorations mesurées ont été perturbées par les tests des autres paramètres mais sur un série de 12 mesures nous avons obtenu des tests durant 1038 secondes en moyenne avec le premier pilote, contre 989 secondes avec le deuxième (sous le noyau 2.4.2). (mesures effectuées sur 32=8x4 noeuds sur un switch, avec une configuration W05L2R8, $n = 25000, n_b = 141$.)

Après ces tests nous sommes restés sous Linux 2.4.2 et ce dernier pilote.

A.3 Conclusion : chronologie des résultats obtenus et contributions principales des paramètres étudiés

La chronologie du travail et des résultats principaux est présentée dans la table A.4.

En résumé, et avec des précautions car ces paramètres sont largement interdépendants, on peut lister par ordre d'importance les paramètres optimisés :

– algorithme de diffusion : passage de 50 à 67 GFlops/s sur 210 noeuds, soit à peu près 30 % de gain ;

- paramètres algorithmiques de Linpack : passage de 28.4 à 35 GFlops/s sur 96 noeuds (ceci comprend également quelques améliorations dûes aux Blas), soit à peu près 20% de gain ;
- prise en compte des switches dans l'allocation des processeurs dans la grille virtuelle : gain de 5 GFlops/s sur 70, soit 7% de la performance ;
- Blas/Lapack : le gain entre les différentes versions est moins évident à quantifier car il dépend de chaque test considéré (produit matriciel, linpack-1000…). Sur le test linpack-1000, Atlas 3.2.0 affiche 425 MFlops/s contre 333 pour MKL (*cf.* table A.2), soit un gain de 27 %. Sur les mesures effectuées sur Linpack parallèle avec l'une ou l'autre des versions des Blas, nous avons cité ci-dessus des mesures donnant autour de 950 secondes avec Atlas/Henry, contre 1030 pour Atlas 3.2.0 seul, soit 8 % d'amélioration. Bien sûr on peut également avoir des performances catastrophiques avec de mauvaises librairies ;
- pilotes/noyau Linux : gain de 50 secondes sur 1000, soit 5% de la performance ;

Remarquons enfin que l'importance de ces paramètres varie également selon le degré de parallélisme de la machine : l'importance des switches apparaît pour les exécutions massivement parallèles. Au contraire les Blas vont sans doute jouer un rôle plus visible sur les exécutions séquentielles.

Bibliographie

[1] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice-Hall International Editions, 1989.

[2] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 1–12, jul 1995.

[3] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 12–23, Newport, June 1997.

[4] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. *ACM SIGPLAN Notices*, 31(6) :213–225, June 1996.

[5] Robert D. Blumofe. *Executing multithreaded programs efficiently*. Phd thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 1995.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley. C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli C. E. Zhou. Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8) :207–216, August 1995.

[7] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1) :202–229, February 1998.

[8] OpenMP Architecture Review Board. Openmp c and c++ application program interface. Technical Report 004-2229-001, Computer Science Department, Carnegie Mellon University, October 1998.

[9] C. Boeres, V. E. F. Rebello, and A. P. Nascimento. Scheduling arbitrary task graphs on logp machines. In *5th International Euro-Par Conference on Parallel Processing (EUROPAR'99)*. Toulouse, France, septembre 1999.

[10] S. Browne, J. Dongarra, Garner N., K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of SC'2000*, November 2000.

[11] Gerson G.H. Cavalheiro. *Athapascan-1 : interface générique pour l'ordonnancement dans un environnement d'exécution parallèle*. PhD thesis, INPG, Novembre 1999.

[12] Andréa Schwertner Charão. *Multiprogrammation parallèle générique des méthodes de décomposition de domaine*. PhD thesis, INPG, Septembre 2001.

[13] Claude Cohen-Tannoudji. *Mécanique Quantique, I et II*. Hermann, 1973.

[14] David E. Culler, Richard M Karp, David A. Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Eric Schauser, Ramesh Subramonian, and Thorsten Von Eicken. Logp : a practical model of parallel computation. *Communications of the ACM*, 39(11) :78–85, nov 1996.

[15] David E. Culler, Richard M Karp, David A. Patterson, Abhijit Sahay, Klaus Eric Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten Von Eicken. Logp : Towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7) :1–12, jul 1993.

[16] Jérôme Devémy. Utilisation de pvm pour paralléliser un logiciel de chimie quantique. *Calculateurs parallèles*, 8(2) :215–228, 1996.

[17] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM - Software/Environments/tools, 1998.

[18] Mathias Doreille. *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. PhD thesis, INPG, Décembre 1999.

[19] Gustavo Fernandes, Nicolas Maillard, and Yves Denneulin. Parallelizing a dense matching region growing algorithm for an image interpolation application. In *Proceedings of PDPTA'2001, Las Vegas*, June 2001.

[20] Guy Fishman. Quasi-cube et würtzite : application au gan. In *École thématique du CNRS*, Montpellier, 1997.

[21] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118. San Diego, California, may 1978.

[22] Ian T. Foster, Jeffrey L. Tilson, Albert F. Wagner, Robert J. Shepard, Ron L. Harrison, Rick A. Kendall, and Rik J. Littlefield. Toward high-performance computational chemistry : I. scalable fock matrix construction algorithms. *Journal of Computational Chemistry*, 17(1) :109–123, 1996.

[23] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5) :212–223, May 1998.

[24] François Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. PhD thesis, INPG, Septembre 1999.

[25] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide do the Theory of NP-Completeness*. W.H. Freeman and Company, New-York, 1979.

[26] R.L. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Tech J.*, 45 :1563–1581, 1966.

[27] M. F. Guest, P. Sherwood, and J.A. Nichols. Massive parallelism : the hardware for computational chemistry ? In *Proceedings of Supercomputing, Collision Processes and Applications*. Plenum Publishing Corporation, New-York, Sep 1999.

[28] M. Head-Gordon, J.A. Pople, and M.J. Frisch. MP2 energy evaluation by direct methods. *Chem. Phys. Lett.*, 153 :503–506, 1988.

[29] J.-J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2) :244–257, April 1989.

[30] Jing Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2) :244–257, apr 1989.

[31] Joseph Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[32] Claude-Pierre Jeannerod and Nicolas Maillard. Using computer algebra to diagonalize some kane matrices. *Journal of Physics A*, 33 :2857–2870, 2000.

[33] C.F. Joerg. *The Cilk system for parallel multithreaded computing*. PhD thesis, Massachussets Institute of Technology, january 1996.

[34] Claudine Kahane. *Mécanique Quantique — Notes de cours*, 1996.

[35] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, 16 :517–542, 1996.

[36] Philip Klein, Ajit Agrawa, R. Ravi, and Satish Rao. Approximation through multicommodity flow. In *Proceedings of the 31st annual IEEE Symposium on Foundations of Computer Science*, pages 726–737. IEEE Computer Society, 1990.

[37] Iskander Kort. *Ordonnancement et Modèle d'exécution : cas des Graphes Spécifiques*. PhD thesis, INPG, juillet 1998.

[38] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1) :95–132, mar 1990.

[39] Théodor R. Lascaux P. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*. Masson, 1987.

[40] R. B. Lehoucq and J. A. Scott. An evaluation of software for computing eigenvalues of sparse nonsymmetric matrices. Technical Report MCS-P547-1195, Argonne National Laboratory, Jan 1996.

[41] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *Arpack User's Guide : Solution of large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, Oct 1997.

[42] Ira N. Levine. *Quantum Chemistry*. Prentice Hall, 1991.

[43] Zhen Liu. A note on Graham's bound. *Information Processing Letters*, 36 :1–5, 1990.

[44] B. M. Maggs, L. R. Matheson, and R. E. Trajan. Models of parallel computation : a survey and synthesis. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 61–71. El-Rewini et Shriver, jan 1995.

[45] Nicolas Maillard, Jean-Louis Roch, and Pierre Valiron. Parallélisation du calcul ab-initio de l'énergie de corrélation électronique par la méthode mp2. In *RenPar'9 — 9èmes rencontres francophones du parallélisme*, May 1997.

[46] Istvan Mayer and Pierre Valiron. Second order møller-plesset perturbation theory without basis set superposition error. *Journal of Chemical Physics*, 109(9) :3360–3373, September 1998.

[47] M. Middendorf, W. Löwe, and Zimmermann W. Scheduling inverse trees under the communication model of the logp machine. *Theoretical Computer Science*, 125 :137–168, 1999.

[48] Slaman M.J. and A. Aziz. *Journal of Chemical Physics*, 94(12) :8047, 1991.

[49] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation : A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3) :264–280, July 1991.

[50] Ida M.B. Nielsen and Edward T. Seidl. Parallel direct implementation of second-order perturbation theories. *J. Comp. Chem.*, 16(10) :1301–1313, 1995.

[51] Sven Ortoli and Jean-Pierre Pharabod. *Le Cantique des Quantiques*. La Découverte, 1985.

[52] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2) :322–328, apr 1990.

[53] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Inc., 1980.

[54] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings 28th Annual Symposium on Foundations of Computer Science*, pages 185–192. IEEE, 1987.

[55] Rayward-Smith. Uet scheduling with unit interprocessor communication delay. *Discrete Applied Mathematics*, 18 :55–71, 1987.

[56] B. Richard, P. Augerat, N. Maillard, S. Derr, S. Martin, and C. Robert. I-cluster : Reaching top500 performance using mainstream hardware. Technical Report HPL-2001-206 20010831, HP Laboratories Grenoble, Aug 2001.

[57] Jean-Louis Rivail. *Éléments de Chimie Quantique à l'usage des chimistes*. Inter-Éditions/CNRS Éditions, 1994.

[58] Youssef Saad. *Numerical Methods for Large Eigenvalue Problems*. John Wiley and Sons, New York, 1992.

[59] David J. Simpson and F. Warren Burton. Space efficient execution of deterministic parallel programs. *IEEE Transactions on Software Engineering*, 25(6) :870–882, November 1999.

[60] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : the Complete Reference*. Massachusetts Institute of Technology, 1996.

[61] D. C. Sorensen. Implicit apllication of polynomial filters in a k-step arnoldi method. *SIAM Journal on Matrix Analysis Applied*, 13(1) :357–385, January 1992.

[62] Danny C. Sorensen. Implicitly restarted arnoldi/lanczos methods for large scale eigenvalue calculations. Technical Report TR-96-40, Rice University, May 1996.

[63] P. R. Taylor. *International Journal of Quantum Chemistry*, 31 :521, 1987.

[64] Leslie G. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8) :103–111, August 1990.

[65] Leslie G. Valiant. General purpose parallel architectures. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 944–971. Elsevier, 1990.

[66] Pierre Valiron and Jozef Noga. Towards high accuracy ab-initio quantum chemistry modeling : Open-mp strategies for accelerating an explicitly correlated coupled-cluster code. Technical report, IBM Advanced Computing Technology Center (ACTC), Mai 2000.

[67] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. www.netlib.org, 2000.

[68] Tao Yang and Apostolos Gerasoulis. Scheduling of structured and unstructured computation. In Arnold Rosenberg D. Frank Hsu, editor, *DIMACS Book Series, 1994 DIMACS Workshop on Interconnections Networks and Mapping and Scheduling Parallel Computation*. American Math. Society, 1995.

Résumé

Calcul Haute-Performance et Mécanique Quantique :
analyse des ordonnancements en temps et en mémoire

Ce travail présente l'apport de l'ordonnancement pour la programmation parallèle performante d'applications numériques en mécanique et chimie quantique. Nous prenons deux exemples types de résolution de l'équation de Schrödinger — Boîte Quantique (BQ) et Méthode des Perturbations d'ordre 2 (MP2) — qui nécessitent de grosses ressources en calcul et mémoire. La programmation traditionnelle (échange de messages et/ou multithreading) des machines parallèles (distribuées ou SMP) est illustrée par les performances obtenues avec le benchmark Linpack sur la grappe I-cluster (INRIA). Le manque de portabilité du code hautement performant obtenu montre l'importance d'un environnement de programmation parallèle permettant de découpler le codage de l'algorithme de son ordonnancement sur la machine cible. Nous introduisons alors ATHAPASCAN, qui repose sur l'analyse du flot de données, pour calculer dynamiquement des ordonnancements prouvés efficaces.

Un premier critère d'efficacité est le temps de calcul. Sur certains modèles de machines, la théorie et l'expérience montrent que ATHAPASCAN permet des ordonnancements qui garantissent des exécutions efficaces pour certains algorithmes adaptés à BQ, de type itératif (méthode de Lanczos). Un deuxième critère fondamental est l'espace mémoire requis pour les exécutions parallèles en calcul numérique ; c'est particulièrement critique pour MP2. Nous proposons d'annoter le Graphe de Flot de Données (GFD) manipulé par ATHAPASCAN pour prendre en compte la mémoire et permettre des ordonnancements dynamiques efficaces en mémoire. Pour MP2, dont le GFD est connu statiquement, un ordonnancement efficace en temps et en mémoire est donné.

Mots-clés : programmation parallèle, ordonnancement, calcul numérique, mécanique quantique.

Abstract

High-Performance Computing and Quantum Mechanics :
Analysis of Time and Memory Efficient Scheduling

This work shows the importance of scheduling for the programming of high performance numerical applications of quantum physics and chemistry. We focus on two methods for the Schrödinger equation : the Quantum Box (QB) and the order two Møller-Plesset (MP2) perturbation algorithm. Both require very large amounts of computing time and memory. The traditional programming (message passing and/or multithreading) of parallel architectures (distributed or SMP) is illustrated through the performances obtained with the Linpack benchmark on the I-Cluster machine of INRIA. The lack of portability of the high-performance code obtained shows the need for an environment for parallel programming that allows the disconnecting of algorithm coding from its scheduling on the target machine. We then introduce ATHAPASCAN, that relies on the analysis of the data flow to dynamically compute some schedules of proven efficiency.

For the execution time on some machine models, both theory and experiments show that ATHAPASCAN provides some schedules that guarantee good performance for algorithms adapted to QB, of iterative nature (Lanczos algorithm). Another aim is to bound the memory required by parallel executions in numerical computing ; this is especially true for MP2. We propose to annotate the Data Flow Graph (DFG) to take the memory into account and allow scheduling that is time and memory efficient. For MP2, whose DFG is statically known, a memory and time efficient scheduling is given.

Keywords : parallel programming, scheduling, high-performance computing, quantum mechanics.