Adaptation de l'algorithmique aux architectures parallèles

Alexandre Borghi

▶ **To cite this version:**

Alexandre Borghi. Adaptation de l'algorithmique aux architectures parallèles. Autre [cs.OH]. Université Paris Sud - Paris XI, 2011. Français. NNT : 2011PA112205 . tel-00694498

HAL Id: tel-00694498

https://theses.hal.science/tel-00694498

Submitted on 4 May 2012

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

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

UNIVERSITÉ PARIS-SUD XI

MANUSCRIT DE THÈSE

proposé pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ PARIS-SUD XI

préparé au LABORATOIRE DE RECHERCHE EN INFORMATIQUE
dans le cadre de ***l'Ecole Doctorale d'Informatique de Paris-Sud***

par

Alexandre BORGHI

soutenance prévue le 10 octobre 2011

Adaptation de l'algorithmique aux architectures parallèles

Directeurs de thèse :
M. Sylvain PEYRONNET et M. Jérôme DARBON

JURY

M. Philippe DAGUE	Professeur à l'Université Paris-Sud
M. Jérôme DARBON	Chargé de Recherche au CNRS
M. Daniel KROB	Directeur de Recherche au CNRS
M. Saïd LADJAL	Maître de Conférences à Telecom-ParisTech
M. Sylvain PEYRONNET	Maître de Conférences à l'Université Paris-Sud
M. Patrick SIARRY	Professeur à l'Université Paris-Est Créteil

Table des matières

Résumé **i**

Abstract **iii**

1 Introduction **1**
1.1 Cadre général 1
1.2 Motivations 2
1.3 Contributions 3

2 Eléments d'architectures et de programmation parallèles **5**
2.1 Architectures de processeurs parallèles 5
2.1.1 Processeur multi-coeur (CPU) 6
2.1.2 *Cell Broadband Engine Architecture* (CBEA ou Cell) 7
2.1.3 *Graphics Processing Unit* (GPU) 9
2.2 Architectures d'ordinateurs parallèles 11
2.2.1 Système multiprocesseur 11
2.2.2 *Cluster* 11
2.3 Programmation parallèle 12
2.3.1 *Multithreading* 12
2.3.2 Vectorisation 13
2.3.3 Mémoire 13

3 Modélisation de la factorisation entière par PLNE **15**
3.1 Description du problème 15
3.2 Etat de l'art 16
3.3 Formulations en PLNE 17
3.3.1 Première formulation 18
3.3.2 Seconde formulation 20
3.4 Implémentation et expériences 22
3.4.1 Implémentation 22
3.4.2 Expériences 23
3.5 Conclusion 26

4 *Compressive sensing* – Approche par prog. linéaire **29**
4.1 Description du problème 29
4.2 Décomposition de Dantzig-Wolfe 30
4.2.1 Cas général 30
4.2.2 Application au *compressive sensing* 34
4.3 Décomposition réduite 38
4.3.1 Présentation de la décomposition proposée 38
4.3.2 Equivalence avec le problème original 39
4.3.3 Point de vue des coûts réduits 40
4.3.4 Conséquences sur les règles de pivot 41
4.3.5 Coûts calculatoires 43
4.4 Implémentation et expériences 44
4.4.1 Implémentation 44
4.4.2 Expériences 45
4.5 Conclusion 50

5 *Compressive sensing* – Approche par prog. convexe **51**
5.1 Etat de l'art des résolutions approchées 52
5.2 Un algorithme basé sur le point proximal 52
5.2.1 Régularisation de Moreau-Yosida 52
5.2.2 Preuve de convergence 54
5.2.3 Choix de l'opérateur proximal 55
5.2.4 Algorithme proposé 56
5.3 Implémentations et expériences 59
5.3.1 Implémentations 59
5.3.2 Expériences 61
5.4 Conclusion 71

6 Vérification approchée sur architecture parallèle **73**
6.1 Contexte 74
6.1.1 Problème 74
6.1.2 Etat de l'art 74
6.1.3 Méthode de résolution APMC 74
6.2 Architecture logicielle d'APMC 3.0 75
6.3 Utilisation du modèle BSP pour parallélisation semi-automatique . . 77
6.3.1 Modèle BSP 78
6.3.2 Bibliothèque BSP++ 79
6.3.3 Support pour programmation hybride 81
6.3.4 Expériences 83
6.4 Adaptation d'APMC à l'architecture parallèle Cell 87
6.4.1 Nouvelle implémentation : APMC-CA 87
6.4.2 Expériences 89
6.5 Conclusion 91

7 Conclusion et perspectives **93**
7.1 Bilan 93

7.2 Perspectives 94

A Annexe **97**
A.1 Résultats complémentaires pour le chapitre 3 97
A.2 Résultats complémentaires pour le chapitre 4 98

Références bibliographiques **101**

Résumé

Dans cette thèse, nous nous intéressons à l'adaptation de l'algorithmique aux architectures parallèles. Les plateformes hautes performances actuelles disposent de plusieurs niveaux de parallélisme et requièrent un travail considérable pour en tirer parti. Les superordinateurs possèdent de plus en plus d'unités de calcul et sont de plus en plus hétérogènes et hiérarchiques, ce qui complexifie d'autant plus leur utilisation.

Nous nous sommes intéressés ici à plusieurs aspects permettant de tirer parti des architectures parallèles modernes. Tout au long de cette thèse, plusieurs problèmes de natures différentes sont abordés, de manière plus théorique ou plus pratique selon le cadre et l'échelle des plateformes parallèles envisagées.

Nous avons travaillé sur la modélisation de problèmes dans le but d'adapter leur formulation à des solveurs existants ou des méthodes de résolution existantes, en particulier dans le cadre du problème de la factorisation en nombres premiers modélisé et résolu à l'aide d'outils de programmation linéaire en nombres entiers.

La contribution la plus importante de cette thèse correspond à la conception d'algorithmes pensés dès le départ pour être performants sur les architectures modernes (processeurs multi-coeurs, Cell, GPU). Deux algorithmes pour résoudre le problème du *compressive sensing* ont été conçus dans ce cadre : le premier repose sur la programmation linéaire et permet d'obtenir une solution exacte, alors que le second utilise des méthodes de programmation convexe et permet d'obtenir une solution approchée.

Nous avons aussi utilisé une bibliothèque de parallélisation de haut niveau utilisant le modèle BSP dans le cadre de la vérification de modèles pour implémenter de manière parallèle un algorithme existant. A partir d'une unique implémentation, cet outil rend possible l'utilisation de l'algorithme sur des plateformes disposant de différents niveaux de parallélisme, tout en ayant des performances de premier ordre sur chacune d'entre elles. En l'occurrence, la plateforme de plus grande échelle considérée ici est le *cluster* de machines multiprocesseurs multi-coeurs. De plus, dans le cadre très particulier du processeur Cell, une implémentation a été réécrite à partir de zéro pour tirer parti de celle-ci.

Mots clefs

Parallélisation, vectorisation, architectures parallèles, multi-coeur, GPU, Cell, programmation linéaire, programmation convexe

Abstract

In this thesis, we are interested in adapting algorithms to parallel architectures. Current high performance platforms have several levels of parallelism and require a significant amount of work to make the most of them. Supercomputers possess more and more computational units and are more and more heterogeneous and hierarchical, which make their use very difficult.

We take an interest in several aspects which enable to benefit from modern parallel architectures. Throughout this thesis, several problems with different natures are tackled, more theoretically or more practically according to the context and the scale of the considered parallel platforms.

We have worked on modeling problems in order to adapt their formulation to existing solvers or resolution methods, in particular in the context of integer factorization problem modeled and solved with integer programming tools.

The main contribution of this thesis corresponds to the design of algorithms thought from the beginning to be efficient when running on modern architectures (multi-core processors, Cell, GPU). Two algorithms which solve the compressive sensing problem have been designed in this context: the first one uses linear programming and enables to find an exact solution, whereas the second one uses convex programming and enables to find an approximate solution.

We have also used a high-level parallelization library which uses the BSP model in the context of model checking to implement in parallel an existing algorithm. From a unique implementation, this tool enables the use of the algorithm on platforms with different levels of parallelism, while obtaining cutting edge performance for each of them. In our case, the largest-scale platform that we considered is the cluster of multi-core multiprocessors. More, in the context of the very particular Cell processor, an implementation has been written from scratch to take benefit from it.

Keywords

Parallelization, vectorization, parallel architectures, multi-core, GPU, Cell, linear programming, convex programming

CHAP. 1

INTRODUCTION

1.1 Cadre général

Dans cette thèse réalisée au Laboratoire de Recherche en Informatique (LRI) de l'Université Paris-Sud 11 et encadrée par Sylvain Peyronnet et Jérôme Darbon, nous nous intéressons à l'adaptation de l'algorithmique, et plus précisément des méthodes de résolution, aux architectures parallèles modernes. Les ordinateurs actuels proposent différents niveaux de parallélisme. Cela va de la présence de plusieurs processeurs, chacun disposant de plusieurs coeurs, qui eux-mêmes permettent de faire des calculs en parallèle grâce à des instructions vectorielles, à la présence de cartes graphiques composées de plusieurs centaines d'unités de calcul. Ces dernières disposent en effet d'unités de calcul dont la précision est désormais suffisante pour qu'elles puissent être utilisées dans le cadre du calcul scientifique.

Au-delà des ordinateurs classiques, qui disposent déjà d'une grande puissance de calcul, la plupart des superordinateurs sont architecturés autour d'ordinateurs correspondant à la description du paragraphe précédent reliés en réseau. Cependant, nous sortons ici du cadre du calcul parallèle pour entrer dans le domaine du calcul distribué, qui sera abordé en fin de manuscrit.

Les modèles de calcul permettent de déterminer d'un point de vue théorique les performances d'un algorithme. Les modèles classiques, tels que le modèle RAM (*Random Access Machine*), permettent ainsi d'analyser les performances des algorithmes sur les plateformes classiques. Cependant, depuis quelques années, les plateformes réelles s'éloignent significativement de ces modèles de calcul et ces derniers sont trop abstraits pour permettre de prédire les performances pratiques sur les ordinateurs modernes. Par exemple, dans le modèle RAM, l'accès à n'importe quel élément en mémoire est en temps constant, alors que les architectures actuelles disposent d'une hiérarchie de mémoires ayant chacune des caractéristiques très différentes. Le modèle PRAM (*Parallel Random Access Machine*) suppose quant à lui l'existence d'une unique mémoire et d'un unique pointeur de programme, ce

qui n'est absolument pas représentatif des architectures de plus en plus complexes des superordinateurs. Ainsi, certains algorithmes efficaces en théorie ne le sont pas en pratique. Deux problématiques émergent. La première est de déterminer si nous pouvons développer des algorithmes adaptés aux plateformes modernes. Nous parlons bien d'algorithme, et non pas d'implémentation. La seconde consiste en la mise en place d'un formalisme théorique, un modèle, qui corresponde aux caractéristiques des architectures modernes. Dans le cadre de ma thèse, je me suis attaché à travailler sur la première problématique : développer des algorithmes efficaces pour les plateformes modernes.

1.2 Motivations

Les ordinateurs actuels disposent d'une grande puissance de calcul, mais en tirer parti est aujourd'hui un grand défi. En effet, les architectures de processeurs sont très variées et même si elles reposent sur des caractéristiques communes, elles sont tout de même particulièrement différentes les unes des autres. De plus, les superordinateurs les plus performants ont désormais tendance à adopter une architecture hybride, c'est-à-dire à disposer de plusieurs types de processeurs différents, qui peuvent être spécifiquement conçus pour certains types de tâches, ou être plus versatiles.

Du point de vue du développement logiciel, il existe différents outils standards permettant de paralléliser un programme. Cependant, avoir recours à certains types de processeurs (tels que le Cell ou ceux présents sur les cartes graphiques, les GPU) demande d'utiliser des bibliothèques particulières, ou même des langages de programmation spécifiques. Ainsi, chaque plateforme requiert souvent une implémentation particulière pour atteindre de bonnes performances. Il existe néanmoins des implémentations d'outils classiques de parallélisation pour ces architectures, mais ces dernières étant très singulières, l'utilisation de ces outils occasionne très souvent une dégradation significative des performances.

L'hétérogénéité de ces architectures d'ordinateur et la variété des architectures de processeur demandent un travail précis au niveau de l'implémentation des algorithmes, ce qui est d'autant plus vrai quand ces derniers ne sont pas naturellement adaptés aux architectures. En effet, un grand nombre d'algorithmes sont intrinsèquement séquentiels ou scalaires, ce qui les désavantage très fortement sur ces architectures. De plus, certains algorithmes disposent d'une description de haut niveau, c'est-à-dire qu'ils utilisent des notions de mathématiques suffisamment abstraites pour nécessiter des choix conséquents au niveau de l'implémentation, notamment sur la représentation et les structures de données, ainsi que les sous-algorithmes utilisés.

Dans le cadre de cette thèse, nous nous intéressons à un tel travail d'adaptation de l'algorithmique aux architectures, dans le but d'obtenir des performances de premier ordre. Nous nous intéressons aussi à la reformulation de problèmes pour leur trouver une forme plus adaptée à leur résolution. Un autre point important de cette thèse réside dans la conception d'algorithmes pensés dès le départ pour être efficaces, parallèles et faciles à implémenter sur les architectures parallèles modernes.

1.3 Contributions

Tout au long de ce manuscrit, plusieurs problèmes de natures différentes sont abordés. Les différents travaux effectués sont présentés, du plus théorique au plus pratique. De plus, au fur et à mesure des chapitres, nous envisageons des architectures parallèles d'échelle de plus en plus grande.

Le chapitre 2 est un chapitre préliminaire. Nous y présentons des informations sur les différentes architectures de processeurs et d'ordinateurs envisagées tout au long du manuscrit, ainsi que des concepts de plus haut niveau qui en découlent. Ces concepts nous serviront lors de la conception d'algorithmes et d'implémentations efficaces pour ces architectures.

Dans le chapitre 3, nous nous intéressons à l'utilisation de solveurs disposant d'une implémentation parallèle et performante déjà existante. Le travail se situe donc au niveau de la modélisation du problème donnée en entrée de ces solveurs. En l'occurrence, nous travaillons sur la factorisation en nombres premiers formulée par programmation linéaire en nombres entiers. La modélisation la plus simple ne permet pas de gérer des nombres de taille intéressante à cause de restrictions au niveau des nombres à virgule flottante utilisés, et amène des problèmes d'instabilités numériques. Une seconde formulation est apportée pour résoudre ces problèmes, mais sa résolution est plus lente. Plusieurs solveurs performants et parallèles sont utilisés pour montrer l'influence des solveurs sur la résolution. Ce travail réside principalement en une série de reformulations qui ont été conduites dans le but d'adapter la forme du problème aux contraintes pratiques liées aux solveurs. On peut noter que le travail présenté dans ce chapitre a été réalisé de manière indépendante, sans collaborateurs, avec le consentement des encadrants. Ici, nous proposons un travail de haut niveau sur la modélisation d'un problème particulier et ses conséquences, mais sans une réelle implémentation de la méthode de résolution, étant donné que celle-ci est laissée à des solveurs tiers. Les chapitres suivants travaillent à l'échelle de l'implémentation, et traitent donc de sujets de plus bas niveau.

Dans le chapitre 4, nous abordons le problème du *compressive sensing*. Un premier algorithme est proposé pour résoudre ce problème de manière exacte et efficace, en utilisant une approche reposant sur la programmation linéaire. En général, les méthodes de résolution pour ce problème sont approchées, ce qui permet d'obtenir très rapidement une solution de bonne qualité. Ici, une résolution exacte permet entre autres de comparer les différentes méthodes selon des critères qualitatifs et non pas uniquement sur des critères de performances. Nous verrons que cette méthode propose une grande flexibilité dans les sous-algorithmes qui la composent et qu'elle est compatible avec l'utilisation de transformées rapides à la place de matrices (qui sont souvent utilisées en raison de leurs meilleures performances). Ce travail a donné lieu à un article accepté à la conférence *ISVC* [24] et à un article actuellement soumis à la revue *JSPS*, et a été réalisé en collaboration avec Jérôme Darbon du CMLA (ENS Cachan), en partie dans le cadre du projet MIDAS du programme COSINUS de l'ANR.

Dans le chapitre 5, nous proposons un second algorithme pour le problème du *compressive sensing*. Notre méthode de résolution est cette fois approchée et repose sur la programmation convexe. Cet algorithme a été conçu dans le but d'être efficace

et de permettre une implémentation rapide et simple sur différentes architectures de processeurs parallèles. En effet, l'approche retenue est particulièrement efficace dans le cadre de calculs parallèles (de la vectorisation au multiprocesseur), mais ne serait pas aussi adaptée à un cadre distribué. Plusieurs implémentations ont été développées suivant l'architecture ciblée (processeurs multi-coeurs, Cell ou GPU). De plus, nous présentons aussi une variante exacte de notre méthode de résolution, qui peut s'appliquer efficacement dans le cas de l'utilisation de matrices (par opposition aux transformées rapides). Ce travail a mené à la publication d'un article accepté à la conférence *SiPS* [21] ainsi qu'à celle d'un article dans la revue *TCS* [22], et a été réalisé en collaboration avec Jérôme Darbon, alors en poste à UCLA (USA).

Dans le chapitre précédent, des implémentations spécifiques ont été réalisées pour les différentes architectures de processeur considérées. Dans le chapitre 6, nous traitons principalement de l'utilisation d'outils génériques de haut niveau (BSP++) permettant de paralléliser pour différentes architectures d'ordinateurs et avec différentes sous-implémentations, en n'ayant à écrire qu'une seule et unique implémentation. Entre autres, cela permet l'utilisation de machines multiprocesseurs et/ou multi-coeurs (SMP), de clusters et de clusters de SMP, avec OpenMP, MPI ou MPI+OpenMP, ce dernier choix étant effectué à la compilation. Nous travaillons ici sur un problème de vérification de modèles, et plus particulièrement sur APMC, proposé par [90]. Cette méthode de résolution probabiliste et approchée dispose d'un parallélisme de haut niveau adapté au calcul distribué et permet donc de considérer une utilisation très variée de BSP++. Cependant, les méthodes de parallélisation de plus bas niveau (vectorisation) sont très peu adaptées ici, étant donné le côté intrinsèquement scalaire d'APMC. Une implémentation sur Cell utilisant la bibliothèque BSP++ est aussi considérée, mais elle ne permet ici que de traiter des instances de taille très faible et les performances sont mauvaises. Ainsi, à la fin de ce chapitre, nous proposons une nouvelle implémentation d'APMC dont le but est de tirer parti des caractéristiques du Cell. L'algorithme en lui-même n'est pas modifié, mais la représentation des données et la manière d'effectuer les opérations sont adaptées à cette architecture. Ce travail a donné lieu à deux articles acceptés aux conférences *PDMC* [71] et *QEST* [25]. En particulier, l'article [25] fut le résultat de travaux effectués en stage avec Sylvain Peyronnet et Thomas Hérault, ce dernier étant actuellement à UTK (USA), et l'article [71] a été réalisé en collaboration avec Pierre Esterie et Khaled Hamidouche, deux étudiants en thèse encadrés par Joël Falcou, tous trois au LRI, travaillant entre autres sur le modèle BSP.

CHAP. 2

Eléments d'architectures et de programmation parallèles

Cette thèse a pour axe principal l'adaptation des méthodes de calcul aux architectures modernes. Les ordinateurs actuels disposent de plusieurs niveaux de parallélisme. En effet, un ordinateur peut disposer de plusieurs processeurs, qui eux-mêmes sont composés de plusieurs coeurs, et ces coeurs disposent d'unités de calcul vectoriel. De plus, les cartes graphiques modernes sont composées d'un grand nombre d'unités de calcul, qui peuvent être utilisées en parallèle et dont la précision est devenue suffisante pour qu'elles soient utilisées à des fins de calcul scientifique. Pour concevoir des méthodes de résolution qui puissent tirer parti efficacement (et si possible facilement) des architectures parallèles, il faut garder en tête un certain nombre de concepts généraux et de caractéristiques liés au calcul parallèle sur machines à mémoire partagée, qui pour certains peuvent aussi s'appliquer au cas du calcul distribué.

Dans ce chapitre, nous rassemblons ainsi un certain nombre d'informations sur lesquelles reposent les chapitres suivants. Dans la section 2.1, nous décrivons tout d'abord les architectures de processeurs parallèles considérées. Puis, dans la section 2.2, nous considérons un plus haut niveau de parallélisme, en nous intéressant aux architectures d'ordinateurs parallèles. Enfin, dans la section 2.3, nous introduisons les concepts généraux qui résultent de ces architectures et qui serviront à concevoir des algorithmes et des implémentations parallèles efficaces dans les chapitres suivants.

2.1 Architectures de processeurs parallèles

Cette section présente les architectures parallèles actuelles. En particulier, on en considère trois : les processeurs multi-coeurs avec unités vectorielles, le processeur Cell et les GPU. Ces trois plateformes sont les plus répandues actuellement et partagent un certain nombre de caractéristiques communes. Tout d'abord, elles disposent de

plusieurs coeurs : elles sont parallèles. Par ailleurs, elles ont toutes une mémoire partagée, c'est-à-dire une mémoire centrale qui est accessible par tous les coeurs. Cependant et bien qu'elles soient toutes parallèles, elles n'implémentent pas le parallélisme de la même manière. Ceci a pour conséquence des caractéristiques très différentes, en termes de puissance de calcul et de transfert mémoire, ce qui a de lourdes répercussions sur les performances globales.

2.1.1 Processeur multi-coeur (CPU)

Un processeur multi-coeur classique est composé de plusieurs coeurs, généralement 2 à 8. Chaque coeur est superscalaire, *out-of-order* (c'est-à-dire que les instructions sont réordonnées pour être exécutées dans un ordre favorisant les performances) et dispose d'un certain nombre d'unités de calcul vectoriel (*Single Instruction Multiple Data* ou SIMD), telles que celles pouvant être utilisées par l'intermédiaire des jeux d'instructions AltiVec [48], SSE[1] ou AVX[2]. D'un point de vue de calcul parallèle, les différences majeures entre les processeurs disponibles sont l'interconnexion entre les différents coeurs, la hiérarchie de cache et la présence ou l'absence de contrôleur mémoire intégré.

La figure 2.1 montre une représentation schématique des Intel Core 2 et Intel Core i7 dans leur configuration à 4 coeurs.

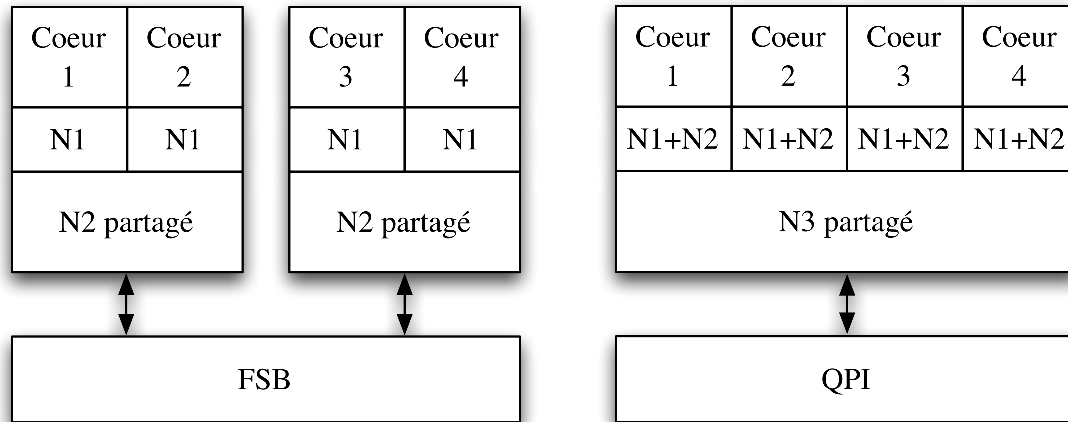

Figure 2.1: Représentation schématique des processeurs Intel à 4 coeurs (à gauche : Core 2 Quad; à droite : Core i7).

Le processeur Intel Core 2 dispose de coeurs regroupés par 2 (ou *cluster*). Chaque coeur a son propre cache de niveau 1 (L1) et chaque groupe de 2 coeurs a un cache de niveau 2 (L2) partagé. Les 2 *clusters* de la version 4 coeurs du Core 2 (le Core 2 Quad) sont reliés entre eux par le FSB par l'intermédiaire du *northbridge* de la carte

[1] http://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms

[2] http://www.intel.com/software/avx

mère. Ce dernier intègre le contrôleur mémoire, qui permet au processeur d'accéder à la RAM. Comme le Core 2 Quad ne dispose pas de cache partagé entre ses 4 coeurs et n'a pas de contrôleur mémoire intégré, la cohérence de cache est maintenue par l'intermédiaire du *northbridge* de la carte mère et du FSB. Cela implique des communications beaucoup plus lentes entre coeurs de *clusters* différents, ce qui se répercute très souvent sur les performance générales.

Au contraire, le processeur Intel Core i7 a un unique *cluster* pour ses 4 coeurs. De plus, il dispose d'un cache de niveau 3 (L3) partagé entre ses 4 coeurs, en plus d'un cache privé de niveau 2 pour chacun de ses coeurs. Grâce au cache partagé, une meilleure utilisation globale du cache apparaît quand plusieurs *threads* travaillent sur les mêmes données. Le FSB est remplacé par le QPI, une interconnexion point-à-point, et le contrôleur mémoire est intégré désormais au sein du processeur. Ceci permet non seulement de réduire la latence, mais aussi d'augmenter la bande passante. De plus, le Core i7 dispose de deux fonctionnalités qui ont des répercussions sur le calcul parallèle : le *Turbo Boost* (TB) et l'*Hyper-Threading* (HT). La première consiste à augmenter dynamiquement la fréquence au delà de la fréquence nominale du processeur pour les coeurs actifs quand d'autres coeurs sont inactifs. Quand cette fonctionnalité est activée, elle peut engendrer un biais dans les résultats expérimentaux. La seconde fonctionnalité, l'HT, permet au système d'exploitation de voir 2 coeurs logiques par coeur physique (SMT 2 voies). Ceci peut permettre une meilleure utilisation des unités de calcul : comme 2 *threads* ont des suites d'instructions indépendantes, le pipeline du coeur peut être rempli plus efficacement et le moteur *out-of-order* peut donner de meilleurs résultats. Cependant, gérer 2 *threads* par coeur peut aussi nuire aux performances comme cela accroît la pression sur le cache (plus de défauts de cache), sur les unités de calcul, la RAM, etc.

Ces deux modèles de processeurs permettent théoriquement d'atteindre 48 GFLOPS en double précision ou 96 GFLOPS en simple précision à 3GHz.

Plusieurs possibilités existent pour écrire du code parallèle dans le cadre classique d'architectures SMP (*Symmetric Multiprocessing*) : on peut par exemple utiliser les bibliothèques pThread ou BOOST::Thread[3] (cette dernière ayant l'avantage d'être orientée objet). On peut aussi utiliser l'API OpenMP [35, 41] ou MPI [60]. Pour utiliser les unités vectorielles, il faut utiliser le jeu d'instructions correspondant, soit directement en assembleur, soit en passant par des fonctions *intrinsics*, qui permettent entre autres de s'affranchir de la gestion des registres au prix de performances plus modestes.

2.1.2 *Cell Broadband Engine Architecture* (CBEA ou Cell)

Le CBEA est une architecture développée par Sony, Toshiba and IBM [107]. Celle-ci a été conçue pour disposer de performances élevées, tout en ayant un bon rendement énergétique. En particulier, la bande passante dont dispose l'interconnexion entre les coeurs est très élevée. La figure 2.2 représente l'architecture du processeur Cell.

Le Cell repose sur un coeur PowerPC 64 bits *in-order* (aussi appelé *Power Processing Element* ou PPE) avec unité vectorielle AltiVec, qui contrôle plusieurs coeurs vectoriels

[3] http://www.boost.org

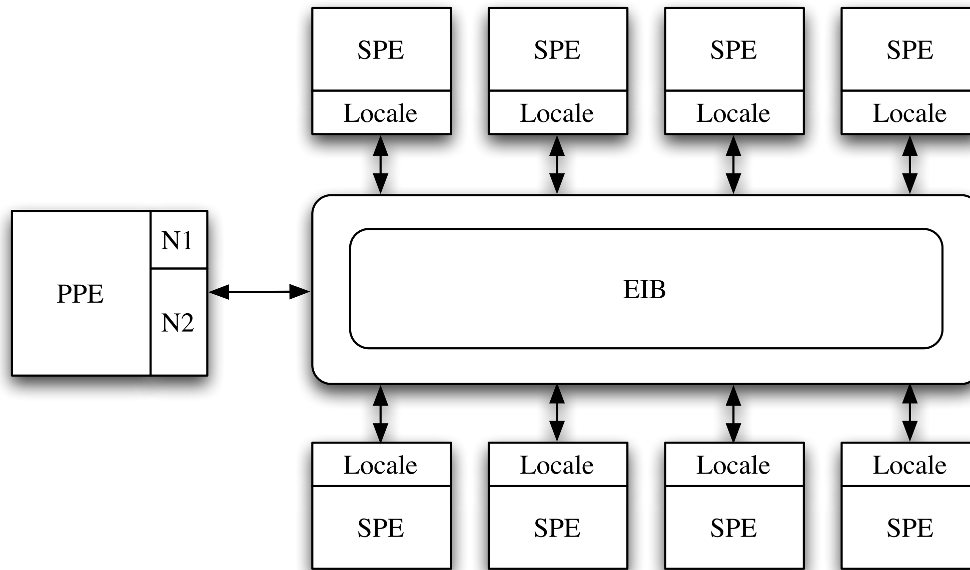

Figure 2.2: Représentation schématique du Cell.

(appelés *Synergistic Processing Element* ou SPE). Un SPE est un coeur *in-order* qui regroupe une unité vectorielle (SIMD) de 128 bits et 256Ko de mémoire locale. Les implémentations actuelles de l'architecture CBEA ont jusqu'à 8 SPE. Le PPE et les SPE sont reliés entre eux par un bus en anneaux disposant d'une très grande bande passante (bus appelé *Element Interconnect Bus* ou EIB).

Comme un SPE dispose de seulement 256Ko de mémoire locale, il est inévitable de devoir réaliser un grand nombre de transferts à travers l'EIB. Néanmoins, grâce à sa bande passante très élevée, des calculs qui sont généralement limités par la mémoire, comme la multiplication matrice/vecteur, ne le sont pas sur Cell. Par conséquent, un certain nombre de nouvelles opérations peuvent être implémentées efficacement de manière parallèle sur Cell [88]. Comme aucune unité scalaire n'est présente sur les SPE, il est particulièrement important de tirer profit au maximum de la vectorisation, sinon les unités de calcul seraient sous-exploitées.

L'implémentation la plus commune de l'architecture CBEA se retrouve dans la Playstation 3 (PS3) de Sony. En effet, celle-ci possède un processeur Cell à 3,2GHz qui dispose de 7 SPE et de 256Mo de RAM. Cependant, seuls 6 SPEs et 192Mo de RAM sont disponibles, le reste étant réservé au système d'exploitation. Un SPE à 3,2GHz dispose d'une puissance de calcul de 25,6 GFLOPS en simple précision. Le Cell de la PS3 peut donc atteindre 153,6 GFLOPS en simple précision avec ses 6 SPE, c'est-à-dire approximativement 50% de mieux qu'un processeur Intel 4 coeurs à 3GHz. En ce qui concerne les performances en double précision, un SPE du Cell de la PS3 n'atteint que 1,6 GFLOPS, c'est-à-dire 9,6 GFLOPS pour les 6 SPE. Cette sévère limitation a été corrigée avec l'arrivée du PowerXCell 8i d'IBM qui, avec 8 SPE à 3,2GHz, atteint 102,4 GFLOPS en double précision, c'est-à-dire 12,8 GFLOPS par SPE

(ou 204,8 GFLOPS en simple précision pour les 8 SPE). De plus, il faut noter que le Cell de la PS3 a un PPE qui atteint tout de même 6,4 GFLOPS en double précision (et 25,6 GFLOPS en simple précision). Comme les SPE du Cell ne disposent pas d'unité de calcul scalaire, ce processeur est beaucoup plus adapté au code vectoriel (SIMD). Enfin, les branchements conditionnels sont très coûteux sur les SPE.

Les performances élevées du Cell viennent cependant au prix d'une programmation très pénible à mettre en oeuvre, comme nous pouvons l'observer dans [56, 121]. En effet, dans le cadre de l'utilisation très classique de la bibliothèque libSPE2 avec pThread développée par IBM dans son SDK[4], beaucoup d'opérations usuelles sont assez complexes à mettre en oeuvre. Par exemple les transferts DMA doivent être réalisés à la main. Beaucoup d'efforts ont été faits pour améliorer la facilité d'utilisation du Cell, notamment au travers d'implémentations spécifiques d'OpenMP [105] et de MPI [87].

2.1.3 *Graphics Processing Unit* (GPU)

Les GPU modernes (tels que ceux de NVIDIA) reposent sur un nombre conséquent de processeurs de flux qui partagent une mémoire commune [100]. Les GPU peuvent aussi être vus comme des *clusters* d'unités SIMD. Ces processeurs à coeurs homogènes ont initialement été conçus pour le rendu de graphismes en 3D, même si, au fur et à mesure des nouvelles versions, de plus en plus de fonctionnalités sont orientées calcul scientifique. Notamment, la précision des calculs est devenue telle que les GPU peuvent être utilisés dans ce nouveau cadre : le *General-Purpose* GPU (GPGPU) [106]. Il est toutefois à noter que certains calculs n'ont pas la précision que l'on peut obtenir sur CPU, et que le support des flottants double précision demande un sacrifice très grand en termes de performances. Il faut aussi noter que les GPU ne fonctionnent pas de manière indépendante, comme c'est le cas pour les CPU ou le Cell. En effet, le GPU est obligatoirement piloté par un CPU. De plus, un même système peut disposer de plusieurs GPU.

Les architectures de GPU G80 et G200 de NVIDIA sont représentées dans la figure 2.3. Contrairement aux coeurs de CPU, les coeurs de GPU sont scalaires et correspondent plus à des unités de calcul de CPU. Ils disposent tout de même d'une hiérarchie de cache de plusieurs niveaux, mais très différente de ce qu'on peut retrouver sur CPU. En effet, les coeurs sont groupés en *clusters* (multiprocesseurs) qui disposent chacun d'un cache de niveau 1. Les *clusters* sont subdivisés en groupes de 8 coeurs qui partagent une mémoire locale. Ces 8 coeurs exécutent la même instruction sur des données différentes, d'où le côté SIMD avec une instruction sur un vecteur de 8 éléments, chacun géré par une unité scalaire (ici abusivement appelée coeur). Le nombre de coeurs par *cluster* peut varier suivant les modèles : 16 coeurs sur l'architecture G80 et 24 sur G200, qui est une version améliorée du G80.

Grâce à un bus dédié, les coeurs de GPU ont accès à une mémoire de plusieurs centaines de Mo particulièrement rapide. En particulier, cette VRAM est beaucoup plus rapide que la RAM centrale. Cela permet d'utiliser efficacement la grande

[4] https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine

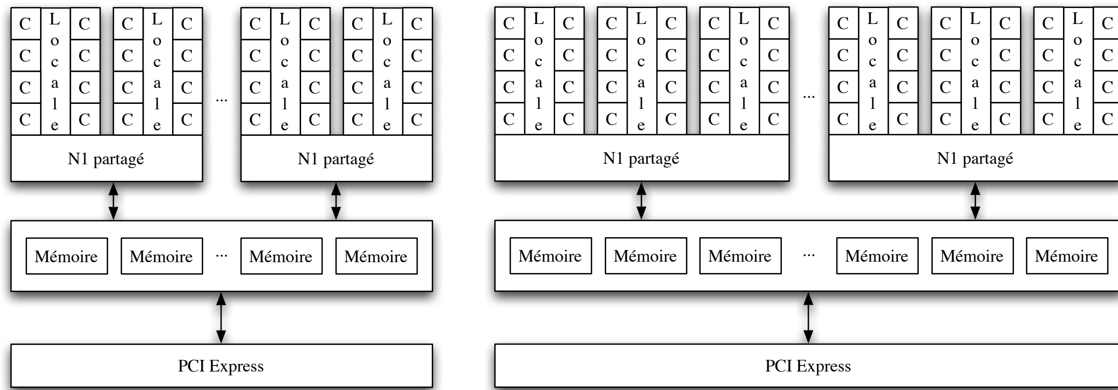

Figure 2.3: Représentation schématique des GPU NVIDIA (à gauche : G80; à droite : G200).

quantité d'unités de calcul qui sont particulièrement consommatrices en bande passante. La RAM centrale peut aussi être accédée par l'intermédiaire du bus PCI Express. Néanmoins, les transferts entre la VRAM et RAM sont très coûteux, comme la bande passante du bus PCI Express est limitée. Ainsi, de tels transferts doivent être évités au maximum. Enfin, les branchements conditionnels sont très coûteux sur GPU.

Grâce à leur architecture radicalement différente, les GPU ont des performances théoriques qui sont un ordre de grandeur plus grandes que celles des CPU modernes. Par exemple, une carte graphique NVIDIA 8800 GTS (architecture G80) avec 96 coeurs à 1,2GHz a une puissance de calcul de 345 GFLOPS en simple précision et une NVIDIA GTX 275 (architecture G200) avec 240 coeurs à 1,4GHz a une puissance de 1010 GFLOPS (toujours en simple précision). Rappelons que les processeurs Intel à 4 coeurs présentés précédemment ont une puissance de calcul de 96 GFLOPS en simple précision à 3GHz.

Comme les GPU étaient initialement utilisés pour des rendus 3D, ils ne pouvaient être accédés qu'à travers des bibliothèques graphiques, telles que Microsoft Direct3D[5] ou SGI OpenGL[6]. Dans le but de pouvoir implémenter et exécuter des algorithmes numériques sur GPU, NVIDIA a développé CUDA [7], qui repose sur un compilateur pour un langage sur-ensemble du C, ainsi qu'un ensemble d'outils et de bibliothèques. NVIDIA a récemment fourni une implémentation d'OpenCL pour ses GPU[8].

[5]http://msdn.microsoft.com/fr-fr/directx
[6]http://www.opengl.org
[7]http://developer.nvidia.com/category/zone/cuda-zone
[8]http://developer.nvidia.com/opencl

2.2 Architectures d'ordinateurs parallèles

2.2.1 Système multiprocesseur

Un système multiprocesseur repose sur plusieurs processeurs au sein d'un même ordinateur. Chaque processeur peut être lui-même multi-coeur. Les processeurs sont reliés entre eux par une interconnexion qui est différente suivant les modèles. Par exemple, la figure 2.4 représente schématiquement un ordinateur à base de 4 Opterons. Ici, les 4 Opterons sont reliés par une connexion point-à-point à base de liens HyperTransport où chaque processeur est directement relié à deux autres. Chaque processeur utilise donc deux liens HyperTransport dans le cadre des communications entre processeurs. Ainsi, quand un processeur A veut communiquer avec le processeur B, il peut réaliser un ou deux sauts suivant le dit processeur B.

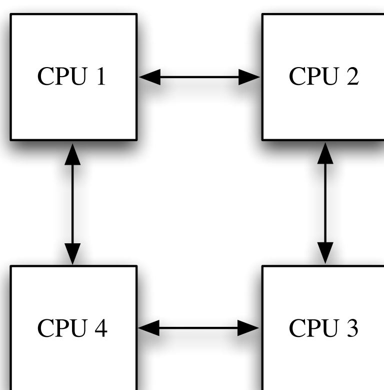

Figure 2.4: Représentation schématique d'un ordinateur à base de 4 processeurs Opteron.

D'un point de vue du logiciel, les mêmes outils que ceux utilisés pour la programmation de processeurs multi-coeurs peuvent être utilisés. Le lecteur peut donc se référer à la section 2.1.1 pour de plus amples détails.

2.2.2 *Cluster*

Un *cluster*, ou une grappe d'ordinateurs, regroupe plusieurs ordinateurs fonctionnant de concert. Chaque ordinateur est appelé un *noeud*. Les noeuds sont reliés entre eux par l'intermédiaire d'un réseau (par exemple un réseau Ethernet classique). La mémoire n'est donc plus ici partagée entre les différents processeurs mais privée à chaque noeud. Nous sortons donc du cadre du calcul à proprement dit parallèle, pour entrer dans celui du calcul *distribué*.

La figure 2.5 représente une configuration de *cluster* à 16 noeuds.

Chaque noeud peut être lui-même un système multiprocesseur où chaque processeur est multi-coeur, ainsi que multi-GPU, par exemple.

Dans le cadre du développement logiciel en vue d'une exécution sur *cluster*, MPI est l'outil le plus couramment rencontré. En effet, les méthodes à base de *threads* ou

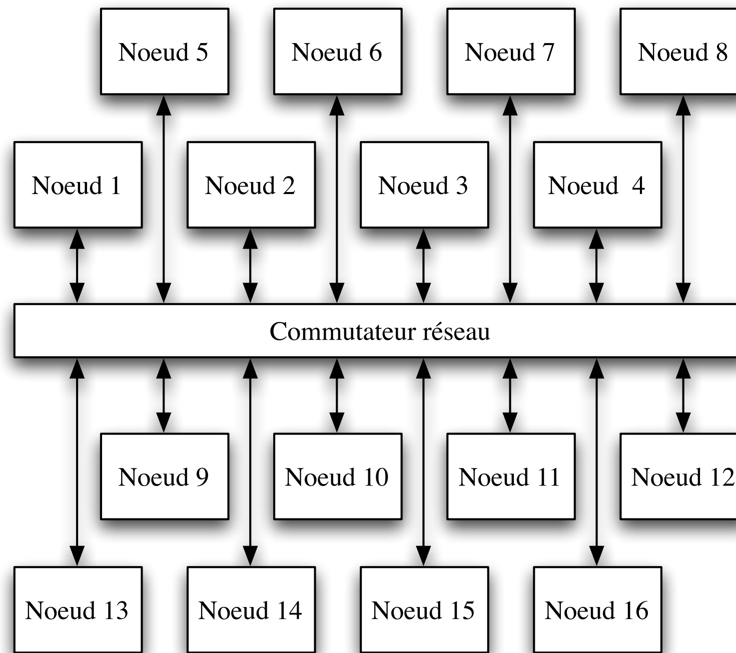

Figure 2.5: Représentation schématique d'un *cluster* à 16 noeuds.

d'OpenMP ne sont plus utilisables, sauf si les noeuds disposent de plusieurs coeurs, auquel cas un modèle hybride MPI+OpenMP peut être choisi à la place d'un modèle plus classique, reposant uniquement sur MPI.

2.3 Programmation parallèle

Dans cette section, nous décrivons les caractéristiques à prendre en compte d'un point de vue général dans le cadre d'une conception *et* d'une implémentation efficaces sur les architectures parallèles modernes (et en particulier celles décrites précédemment).

2.3.1 *Multithreading*

Les plateformes modernes disposent de plusieurs coeurs, et parfois même de plusieurs processeurs. Les différentes tâches à exécutér peuvent alors être réparties sur ces différents coeurs dans le but d'accélérer le calcul. Cependant, il faut prendre garde à ce que les différents *threads* n'interfèrent pas entre eux. Par exemple, comme la mémoire est partagée, deux *threads* peuvent écrire à un même emplacement mémoire. Pour éviter cette situation, on peut utiliser un verrou qui assurera qu'un seul *thread* à la fois accède à la ressource. L'utilisation de verrous peut entraîner certaines situations où un *thread* A attend une ressource verrouillée par un *thread* B, et le *thread* B attend une ressource verrouillée par le *thread* A. Ce problème classique est dénommé interblocage. En d'autres termes, un interblocage est occasionné quand

deux *threads* s'attendent mutuellement. L'utilisation de verrous peut ainsi amener à des situations particulièrement complexes, et occasionne souvent des pertes de performance, comme ils rompent temporairement le parallélisme (et donc des *threads* peuvent passer du temps à attendre l'accès à la ressource). Il est souvent préférable de modifier la manière dont la mémoire est gérée ou dont les calculs sont effectués pour s'assurer que chaque *thread* n'interfère pas avec un autre sans avoir recours aux verrous. Un ordonnancement approprié des différentes tâches peut aussi aider à ce qu'elles n'interfèrent pas entre elles. La synchronisation des différents *threads* est un problème majeur en calcul parallèle.

2.3.2 Vectorisation

Il existe un niveau de parallélisation plus fin. Il consiste à traiter les données comme des vecteurs, pour pouvoir ainsi utiliser les unités vectorielles (SIMD) des architectures. Les unités vectorielles actuelles peuvent traiter des vecteurs de 128 bits (SSE, AltiVec), voire même 256 bits (AVX). Ainsi, avec AVX une seule instruction permet de traiter 8 flottants simple précision (32 bits) ou 4 flottants double précision (64 bits), ce qui permet un gain en performance particulièrement important. La description classique d'un algorithme ne met cependant pas toujours en évidence une structure pour les données qui permet de vectoriser facilement. Il y a souvent un travail à effectuer pour adapter les structures de données utilisées et faire apparaître des vecteurs de données adéquats par rapport aux opérations à effectuer (et donc au jeu d'instructions SIMD), d'autant plus si du *multithreading* est aussi utilisé. Ce genre de situations amène souvent à devoir faire des choix, par exemple utiliser des tableaux de structures ou des structures de tableaux, et de tels choix peuvent avoir de grandes répercussions sur les performances. L'alignement des données est aussi important d'un point de vue performances, comme les données sont stockées sous forme de vecteurs. Il faut noter que certains algorithmes sont particulièrement mal adaptés à la vectorisation, comme ils ne sont pas fondés sur des primitives vectorielles, mais plutôt sur des primitives intrinsèquement scalaires. Une utilisation optimale de la vectorisation nécessite ainsi une réflexion dès la conception de l'algorithme, et non pas seulement pendant l'implémentation.

2.3.3 Mémoire

La mémoire est un facteur limitant particulièrement fort pour les architectures modernes. En particulier, leur quantité, leur bande passante, leur latence, ou celles des différentes mémoires caches qui permettent d'y accéder efficacement peuvent être des facteurs limitants. En effet, lors d'un accès mémoire pour effectuer un calcul, les données sont transférées de la RAM à un certain nombre de caches (suivant l'architecture) pour pouvoir être manipulées par le processeur, puis les caches sont mis à jour et ensuite la RAM. Pour des calculs plus rapides, il est important de prendre en compte les caractéristiques liées aux caches (en particulier leur hiérarchie, le fait qu'elles soient partagées ou non, leur quantité, etc.) et d'effectuer des opérations en accord avec celles-ci. Ces différentes caractéristiques peuvent en effet avoir de grandes répercussions sur les performances. Un algorithme et son implémentation

peuvent aider le processeur à maintenir efficacement son cache (c'est-à-dire utiliser sa taille limitée à bon escient) en utilisant des schémas d'accès à la mémoire particuliers, qu'il sache gérer efficacement, ou, si ce n'est pas applicable, en utilisant un *prefetch* explicite. Dans le cadre du *multithreading*, il faut faire en sorte de ne pas mettre en défaut le protocole de cohérence de cache utilisé (qui gère le cas où une même donnée en mémoire a des valeurs différentes dans le cache de 2 coeurs). De plus, plusieurs coeurs d'un processeur peuvent accéder simultanément à la mémoire par l'intermédiaire d'un unique contrôleur mémoire, auquel cas la bande passante de chaque coeur est réduite. Pour éviter ce genre de situations, on peut par exemple ordonnancer les accès mémoire et les calculs de telle sorte que les coeurs soient le moins possible en phase d'attente, tout en se partageant correctement les ressources. Pour des transferts plus rapides, leur fréquence et leur taille doit être ajustée suivant les caractéristiques de l'architecture. Par exemple, dans le cas du processeur Cell, des transferts DMA de taille 16Ko sont optimaux, c'est-à-dire que le ratio temps de transfert sur taille transférée est le plus faible. Dans le cadre des GPU, les transferts entre RAM et VRAM passent par le bus PCI Express, dont la bande passante limite très souvent les performances. Ainsi, de tels transferts doivent être évités au maximum. Toujours dans le cadre des GPU, il faut éviter les situations de conflits de bancs mémoires, c'est-à-dire quand plusieurs *threads* accèdent à un même banc mémoire de la VRAM en même temps. Par exemple, [2] implémente un gestionnaire de mémoire pour éviter de tels conflits. Au final, une utilisation adéquate de la mémoire est un facteur déterminant pour les bonnes performances.

CHAP. 3

Modélisation de la factorisation entière par PLNE

Dans ce chapitre, nous présentons une méthode de factorisation entière en nombres premiers fondée sur la résolution d'un programme linéaire en nombres entiers (PLNE). Plusieurs modélisations sont proposées pour ce problème et l'idée est de laisser la résolution à un solveur de PLNE, dont un certain nombre d'implémentations efficaces existent. Le but premier n'est pas ici les performances : des algorithmes rapides spécifiques à ce problème existent et disposent d'implémentations efficaces. L'intérêt se situe surtout autour de la modélisation du problème et de ses conséquences sur la résolution avec des solveurs de PLNE classiques. La première modélisation proposée est simple mais ne permet pas de factoriser de grands nombres, à cause de certaines limitations au niveau des nombres à virgule flottante et peut engendrer des instabilités numériques. La seconde modélisation résout ces problèmes, mais demande en contrepartie plus de variables et plus de contraintes, ce qui a pour conséquence une résolution plus lente.

Tout d'abord, le problème de la factorisation entière en nombres premiers est décrit rapidement dans la section 3.1. Ensuite, un état de l'art des méthodes classiques pour ce problème est dressé en section 3.2. Cet état de l'art fait aussi état d'une méthode basée sur la formulation d'un problème de satisfiabilité (SAT). La section 3.3 décrit ensuite les deux formulations proposées et discute de leurs avantages et inconvénients respectifs. Enfin, un certain nombre d'expériences sont conduites en section 3.4 pour comparer les deux formulations avec différents solveurs et appuyer les allégations de la section précédente.

3.1 Description du problème

Le problème de la factorisation entière en nombres premiers consiste à décomposer un nombre en un produit de facteurs premiers. Le théorème fondamental de

l'arithmétique affirme que tout entier supérieur ou égal à 2 dispose d'une telle décomposition en facteurs premiers et que celle-ci est unique, à l'ordre des facteurs près.

Sans perte de généralités, on peut formuler ce problème pour uniquement deux facteurs comme suit : étant donné un entier c supérieur ou égal à 2, trouver deux diviseurs a et b de c, tels que $c = a \times b$.

3.2 Etat de l'art

Plusieurs algorithmes existent pour le problème de la factorisation entière en nombres premiers. Cependant, seulement peu d'entre eux sont suffisamment efficaces pour être utilisés sur de grands nombres. Il existe néanmoins des applications pour lesquelles on cherche des facteurs de nombres de faible à moyenne taille. En particulier, [98] présente différentes utilisations et compare plusieurs algorithmes dans ce cadre.

Pour factoriser de très grands nombres, l'algorithme de factorisation par crible sur les corps de nombres généralisé (ou GNFS pour *General Number Field Sieve*) [92] est actuellement le plus rapide avec sa complexité heuristique en temps : $O\left(\exp\left(\left(\frac{64}{9}\log c\right)^{\frac{1}{3}}(\log\log c)^{\frac{2}{3}}\right)\right)$, où $c \in \mathbb{N}$ est le nombre à factoriser. Sa complexité en temps dépend uniquement de la taille du nombre à factoriser. C'est pourquoi cet algorithme est particulièrement adapté à la factorisation de nombres RSA (Rivest Shamir Adleman), des nombres semi-premiers (c'est-à-dire avec exactement deux facteurs premiers) avec des facteurs de même taille. Ces nombres prennent leur nom de la méthode de cryptographie à clé publique RSA et se retrouvent dans son utilisation [74]. Cette méthode de chiffrement est très largement répandue et est utilisée par exemple dans le cadre de protocoles de commerce électronique.
En pratique, l'algorithme GNFS est actuellement le plus rapide pour factoriser des nombres de taille supérieure à 110 chiffres en décimal. Le record de factorisation entière en nombres premiers pour les algorithmes qui ne tirent pas parti de la forme spécifique du nombre à factoriser est un nombre RSA de 768 bits (232 chiffres en décimal) et a été réalisé en utilisant cet algorithme [84]. Leur auteurs expliquent que l'étape de sélection des polynômes a duré la moitié d'une année sur 80 processeurs et que l'étape de crible en lui-même a duré presque 2 ans sur plusieurs centaines d'ordinateurs. Ils estiment que l'étape de crible aurait duré 1500 ans sur un unique processeur AMD Opteron 2,2GHz simple coeur avec 2Go de RAM. Il existe d'autres implémentations efficaces, ainsi que parallèles et/ou distribuées, par exemple [123, 124].

Comme expliqué précédemment, la complexité en temps du GNFS ne dépend que de la taille du nombre à factoriser. Au contraire, certains algorithmes ont une complexité en temps qui dépend de la taille des facteurs premiers, comme par exemple l'algorithme de factorisation en courbe elliptique (ou ECM pour *Elliptic Curve Factorization Method*) [93] pour lequel la complexité en temps dépend de la taille du plus petit facteur de c. En effet, sa complexité en temps est : $O\left(\exp\left(\sqrt{2}+O(1)\sqrt{\log a \log\log a}\right)\right)$, où a est le plus petit facteur de c. Cette propriété fait du ECM le meilleur algorithme

pour trouver de petits facteurs à partir de très grands entiers composés. En effet, actuellement le plus grand facteur trouvé par l'algorithme ECM est de 73 chiffres en décimal et est un facteur du nombre $2^{1181} - 1$ (367 chiffres en décimal)[1]. Parmi les implémentations notables, nous pouvons citer GMP-ECM[2] avec son implémentation parallèle, distribuée et qui repose sur la bibliothèque GMP[3], qui tire parti de la vectorisation, ainsi qu'une implémentation sur GPU [13].

Ces algorithmes sont particulièrement complexes et donc difficiles à implémenter efficacement. De plus, leur utilisation demande une compréhension profonde des mécanismes qui les régissent. En effet, ils sont souvent fondés sur des calculs reposant sur des mathématiques avancées et disposent de paramètres dont la détermination doit être très précise pour que le résultat soit obtenu efficacement, tout en restant correct. Pour plus de détails sur les algorithmes séquentiels et parallèles de factorisation en nombres premiers, le lecteur intéressé peut se référer à [28].

Nous pouvons aussi faire mention d'une méthode qui s'éloigne particulièrement des approches classiques et qui repose sur une reformulation en un problème SAT (*boolean SATisfiability*) [79]. Le but était ici de générer des instances difficiles pour le problème SAT, ce qui permet de comparer différents solveurs. Les auteurs de ce travail ont factorisé des nombres semi-premiers avec des facteurs de même taille. A partir d'un nombre de 70 bits, ils obtiennent une instance de SAT avec 9347 variables et 59448 clauses. Ils estiment que, pour un nombre de 256 bits, l'instance de SAT correspondante aurait comporté 22165 variables et 142344 clauses. Bien que ce travail n'ait pas été réalisé dans le but d'obtenir des performances comparables à celle des algorithmes présentés précédemment, nous pouvons noter l'existence de plusieurs implémentations efficaces et parallèles de solveurs SAT[4].

3.3 Formulations en PLNE

L'idée principale est de reformuler le problème de factorisation entière en nombres premiers comme un PLNE, de telle sorte que ce programme puisse être résolu en utilisant un solveur classique.

Les deux formulations décrites dans cette section reposent sur la recherche de a et b tels quel $c = a \times b$ avec c le nombre à factoriser. Autrement dit, trouver les facteurs a et b de c en explorant leurs valeurs potentielles. Dans le cas général, c peut disposer de plus de 2 facteurs premiers. Ici, cela signifie que a et b ne sont pas forcément premiers et que, pour trouver la décomposition en nombres entiers, il faudrait répéter le processus sur a et sur b, et ainsi de suite, jusqu'à ce que plus aucun facteur ne soit trouvé (à part les facteurs triviaux 1 et le nombre à factoriser). Ce processus peut être accéléré en testant la primalité du facteur trouvé si sa taille le permet rapidement ou en effectuant des divisions successives jusqu'à une certaine

[1] Voir http://www.loria.fr/ zimmerma/records/top50.html pour les 50 plus grands facteurs trouvés par ECM.

[2] https://gforge.inria.fr/projects/ecm

[3] http://www.gmplib.org

[4] Voir http://www.satcompetition.org pour les performances des solveurs SAT.

valeur pour éliminer un certain nombre de facteurs potentiels, avant de relancer le processus de factorisation.

Pour des raisons de clarté dans les formulations qui vont suivre, on suppose que si une variable est utilisée alors qu'elle dispose d'un indice qui dépasse son ensemble de définition, cette variable est implicitement égale à 0. Nous supposons aussi que $\lfloor log_2(c) \rfloor = \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor$ pour éviter d'avoir à écrire des cas particuliers, même si cette propriété n'est absolument pas requise par les formulations proposées ici. En effet, ces formulations ne reposent sur aucune propriété particulière, ni sur le nombre à factoriser, ni sur les facteurs. De plus, on peut spécifier la taille des facteurs recherchés pour accélérer la méthode, dans le cas où cette information est connue.

Dans tout ce chapitre, a_0 représente le bit de poids faible de a.

3.3.1 Première formulation

La première formulation utilise l'algorithme classique de multiplication d'entiers en $O(n \times m)$ avec n (resp. m) la taille du multiplicateur (resp. du multiplicande), ainsi qu'une représentation binaire, pour l'un des deux facteurs : le facteur a subit une décomposition binaire alors que b reste en décimal. Nous obtenons alors :

$$(F_1) \begin{cases} \min z \\ s.c.\ \sum_{i=0}^{\lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1} \left(2^i a_i b\right) + z = c \\ \qquad a_i \in \{0,1\}, \forall i \in \{0, \ldots, \lfloor log_2(a) \rfloor\} \\ \qquad b \in \{0, \lfloor \sqrt{c} \rfloor\} \end{cases} \tag{3.1}$$

Nous pouvons aussi noter qu'il n'y a pas besoin de forcer les deux facteurs à être supérieurs ou égaux à 2, grâce à l'hypothèse faite dans la section 3.3. La formulation (F_1) dispose d'une fonction objectif quadratique qui est alors reformulée en utilisant la linéarisation classique binaire/entier de McCormick [97]. Il s'ensuit que :

$$(F_1^{'}) \begin{cases} \min z \\ s.c.\ \sum_{i=0}^{\lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1} \left(2^i p_i\right) + z = c \\ \qquad p_i - \lfloor \sqrt{c} \rfloor a_i \leq 0 \\ \qquad p_i - b \leq 0 \\ \qquad p_i - \lfloor \sqrt{c} \rfloor a_i - b + \lfloor \sqrt{(c)} \rfloor \geq 0 \\ \qquad p_i \in [0, \lfloor \sqrt{c} \rfloor], \forall i \in \{0, \ldots, \lfloor log_2(a) \rfloor\} \\ \qquad a_i \in \{0,1\}, \forall i \in \{0, \ldots, \lfloor log_2(a) \rfloor\} \\ \qquad b \in \{0, \lfloor \sqrt{c} \rfloor\} \end{cases} \tag{3.2}$$

Notons que dans $(F_1^{'})$, les variables p_i sont définies comme continues et qu'elles correspondent au produit de a_i par b, ce qui contraint les p_i à être implicitement entières. $(F_1^{'})$ est un programme linéaire mixte avec $\lfloor log_2(a) \rfloor$ variables binaires, une variable entière, $\lfloor log_2(a) \rfloor$ variables continues et $3\lfloor log_2(a) \rfloor$ contraintes. Nous pouvons aussi aussi noter que b pourrait être lui aussi représenté en binaire mais que cela signifierait approximativement 2 fois plus de variables et un nombre de

contraintes quadratiquement plus élevé. C'est pourquoi b est ici laissé dans sa représentation décimale.

Cette formulation a pour désavantage principal qu'elle introduit de grandes constantes qui sont des puissances de 2. Les solveurs classiques de PLNE, tels que ILOG CPLEX[5], Gurobi [6], COIN-OR CBC [7] ou GLPK [8], utilisent un algorithme de type *Branch and Cut* [10], qui repose sur l'algorithme du simplexe, pour calculer des relaxations continues et sur la génération de coupes pour améliorer ces relaxations. En d'autres termes, ils reposent de manière très prononcée sur des calculs d'algèbre linéaire et toutes les implémentations utilisent des nombres flottants double précision (64 bits) pour stocker et calculer. Les grandes constantes induites par la formulation $(F_1^{'})$ peuvent être stockées dans un flottant double précision, tel que défini par la norme IEEE 754 jusqu'à 2^{1023} (c'est-à-dire approximativement $8,98847 \times 10^{307}$ ou 308 chiffres en décimal) comme il n'a qu'un seul bit à 1. Cependant, ces flottants utilisent une mantisse de 52 bits et la formulation $(F_1^{'})$ ne demande pas seulement de stocker ces constantes, mais aussi d'effectuer des calculs à partir de celles-ci. Une mantisse de 52 bits signifie qu'il ne peut pas y avoir plus de 50 bits d'écart entre le premier bit à 1 et le dernier. Nous pouvons noter que les valeurs stockées peuvent néanmoins dépasser 2^{52}, comme des bits dont la valeur est 0 peuvent suivre le dernier bit à 1.
Cependant, il y a une réelle limite dans ce qui peut être stocké *et* effectivement utilisé lors de calculs. Même dans le cas de puissances de 2 relativement petites, le fait que la formulation $(F_1^{'})$ engendre des calculs entre de petits nombres (variables $0-1$ relaxées en variables $[0,1]$) et des nombres significativement plus grands (constantes puissances de 2) peut entraîner des instabilités numériques. Pour ces raisons, en pratique, même si les solveurs classiques de PLNE savent gérer un certain nombre d'instabilités numériques de manière efficace, cette formulation a une limite dans la taille des nombres qu'elle permet de factoriser, bien plus faible que 52 bits (la section 3.4 revient plus en détails sur cela).
Il est intéressant de noter que si $(F_1^{'})$ est modifié pour que les deux facteurs utilisent une représentation binaire, des valeurs moins grandes entrent en jeu en ce qui concerne les variables. Néanmoins, ce sont les grandes constantes puissances de 2 qui sont le coeur du problème et celles-ci sont toujours présentes.
Pour résoudre ce problème, une bibliothèque multiprécision telle que GMP peut être utilisée. Cependant, cela impliquerait non seulement un surcoût élevé au niveau des performances, mais aussi de devoir modifier et recompiler les solveurs de PLNE. Or, les développeurs des solveurs commerciaux ne donnent pas accès à leur code. Ainsi, cette solution empêcherait d'utiliser, entre autres, CPLEX et Gurobi. De plus, si une bibliothèque multiprécision était utilisée, il faudrait déterminer la précision des calculs à effectuer de telle sorte que les calculs soient corrects et efficaces, ce qui pourrait être long et contraignant, suivant les différents algorithmes utilisés par les solveurs.
Une solution plus élégante consiste à utiliser une formulation en PLNE qui ne ferait

[5] http://www-01.ibm.com/software/integration/optimization/cplex-optimizer
[6] http://www.gurobi.com
[7] https://projects.coin-or.org/Cbc
[8] http://www.gnu.org/software/glpk

pas apparaître de constantes ou de variables trop grandes. Cela permettrait alors de pouvoir factoriser de grands nombres, sans avoir à modifier le code des solveurs. La section suivante décrit une telle formulation.

3.3.2 Seconde formulation

L'idée principale est ici d'éviter la conversion en décimal du nombre à factoriser, qui utilise des puissances de 2. Pour cela, de nouvelles notations sont introduites :

$$I = \{0, \dots, \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1\}$$

$$S(i) = \{(j,k) \in \{0, \lfloor log_2(a) \rfloor\} \times \{0, \lfloor log_2(b) \rfloor\} | j + k = i\}$$

Désormais, les deux facteurs utilisent une représentation binaire et l'algorithme classique de multiplication binaire (quadratique) est utilisé, par l'intermédiaire d'une contrainte pour chaque bit. Cette contrainte décompose chaque somme de la multiplication en un bit de résultat et un entier de retenue. Cette étape requiert l'introduction de nouvelles variables : y_i stocke le $i^{\text{ème}}$ bit du produit $a \times b$ (somme$\%2$) et x_i stocke la $(i+1)^{\text{ème}}$ retenue (somme$/2$). Nous avons alors :

$$(F_2) \begin{cases} \min \sum_{i=0}^{l_c} \left| c_i - y_i \right| \\ s.c. \ \sum_{(j,k) \in S(i)} a_j b_k + x_{i-2} = 2x_{i-1} + y_i, \forall i \in I \\ \quad x_i \in \mathbb{R}, \forall i \in \{0, \dots, \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 3\} \\ \quad y_i \in \{0,1\}, \forall i \in \{0, \dots, \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1\} \\ \quad a_i \in \{0,1\}, \forall i \in \{0, \dots, \lfloor log_2(a) \rfloor\} \\ \quad b_i \in \{0,1\}, \forall i \in \{0, \dots, \lfloor log_2(b) \rfloor\} \end{cases} \quad (3.3)$$

Il est important de remarquer que, comme expliqué précédemment dans la section 3.3, certaines variables ont des indices qui excèdent leur ensemble de définition. Ces variables sont implicitement égales à 0 et existent uniquement pour éviter d'écrire des cas particuliers et ainsi ne pas charger les formulations. En l'occurence, ici, la première contrainte de (F_2) utilise des variables dont l'indice excède son ensemble de définition, et donc pour $i = 0, x_{-2} = 0$, pour $i = 1, x_{-1} = 0$ et pour $i = \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1, x_{\lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 2} = 0$.

Comme pour la formulation (F_1), (F_2) est reformulée en utilisant la linéarisation classique binaire/entier de McCormick [97]. Nous en profitons aussi pour linéariser les valeurs absolues de manière classique. Il vient :

$$(F_2^{'}) \begin{cases} \min \sum_{i=0}^{l_c} z_i \\ s.c. \ \sum_{(j,k) \in S(i)} p_{jk} + x_{i-2} = 2x_{i-1} + y_i, \forall i \in I \\ \quad z_i \geq y_i, \forall i \in I \\ \quad z_i \geq -y_i, \forall i \in I \\ \quad p_{jk} - a_j \leq 0, \forall i \in I, \forall (j,k) \in S(i) \\ \quad p_{jk} - b_k \leq 0, \forall i \in I, \forall (j,k) \in S(i) \\ \quad p_{jk} - a_j - b_k + 1 \geq 0, \forall i \in I, \forall (j,k) \in S(i) \\ \quad z_i \in [0,1], \forall i \in I \\ \quad p_{jk} \in [0,1], \forall i \in I, \forall (j,k) \in S(i) \\ \quad x_i \in \mathbb{R}, \forall i \in \{0, \ldots, \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 3\} \\ \quad y_i \in \{0,1\}, \forall i \in \{0, \ldots, \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1\} \\ \quad a_i \in \{0,1\}, \forall i \in \{0, \ldots, \lfloor log_2(a) \rfloor\} \\ \quad b_i \in \{0,1\}, \forall i \in \{0, \ldots, \lfloor log_2(b) \rfloor\} \end{cases} \tag{3.4}$$

Dans $(F_2^{'})$, les variables p_{jk} sont définies comme continues et correspondent au produit de a_j par b_k alors que les variables z_i sont elles aussi définies comme continues et correspondent aux valeurs absolues des y_i. Les variables p_{jk} (resp. z_i) sont implicitement des variables $0-1$ (resp. entières), grâce aux contraintes (resp. la fonction à minimiser). Les variables x_i sont aussi définies comme continues mais sont implicitement entières, grâce à la première contrainte pour $i = 1$ (ou $i = \lfloor log_2(a) \rfloor + \lfloor log_2(b) \rfloor - 1$), $\forall i \in I, x_i \in \mathbb{N}$.

$(F_2^{'})$ est un programme linéaire mixte avec $2\lfloor log_2(a) \rfloor + 2\lfloor log_2(b) \rfloor + 2$ variables binaires, $2\lfloor log_2(a) \rfloor + 2\lfloor log_2(b) \rfloor - 2 + \lfloor log_2(a) \rfloor \times \lfloor log_2(b) \rfloor$ variables continues et $3\lfloor log_2(a) \rfloor + 3\lfloor log_2(b) \rfloor + 3\lfloor log_2(a) \rfloor \times \lfloor log_2(b) \rfloor$ contraintes, comme $\sum_{i \in I} |S(i)| = \lfloor log_2(a) \rfloor \times \lfloor log_2(b) \rfloor$.

Théoriquement, cette nouvelle formulation $(F_2^{'})$ permet de travailler sur des nombres à factoriser qui peuvent atteindre 2^{52} bits, au lieu de 52 bits pour la précédente formulation $(F_1^{'})$, et sans avoir recours à aucune modification au niveau des solveurs existants (telle que l'utilisation d'une bibliothèque multiprécision). La formulation $(F_2^{'})$ permet aussi d'échapper aux problèmes d'instabilités numériques qui pouvaient être rencontrés précédemment, étant donné qu'il n'y a plus que des variables dont les valeurs sont raisonnablement faibles qui entrent en jeu (les plus grandes valeurs rencontrées étant les retenues).

Telle qu'elle a été écrite, la formulation $(F_2^{'})$ dispose de plus de symétries dans ses solutions que $(F_1^{'})$. En effet, comme la variable b est inférieure ou égale à $\lfloor \sqrt{c} \rfloor$ dans $(F_1^{'})$, a et b ne peuvent pas échanger leurs valeurs (sauf si $a = b$). Contrairement à cela, à l'optimum, a et b peuvent être échangés dans $(F_2^{'})$ et la solution sera toujours valide comme la multiplication entière est commutative et qu'aucune contrainte ne force un facteur à être au moins aussi grand que l'autre. Du fait de la représentation des deux facteurs en binaire dans $(F_2^{'})$ on ne peut pas forcer b à être supérieur ou égal à a de la même manière que dans $(F_1^{'})$ sans introduire de grandes constantes ou un nombre élevé de variables. Cependant et sans perte de généralité, on peut ajouter une contrainte qui implique que b ait au moins le même nombre de bits à 1 que a,

ce qui est moins fort que d'être supérieur ou égal mais qui permet tout de même de supprimer une grande partie des symétries. En effet, dans le cas où a et b ont le même nombre de bits à 1, la contrainte ne supprime pas la symétrie. Cette contrainte s'écrit :

$$\sum_{i=0}^{\lfloor log_2(a) \rfloor} a_i \leq \sum_{i=0}^{\lfloor log_2(b) \rfloor} b_i \tag{3.5}$$

Pour les deux formulations, on peut noter que si l'utilisateur choisit explicitement les tailles des facteurs à des valeurs trop grandes, les facteurs triviaux 1 et c peuvent être trouvés. Cependant, on peut aisément ajouter une contrainte pour éviter cette situation. En effet, il suffit de forcer chaque facteur à avoir un nombre de bits à 1 supérieur ou égal à 2 (en supposant que c est impair, ce qui est facile à vérifier).

$$\sum_{i=0}^{\lfloor log_2(a) \rfloor} a_i \geq 2 \qquad \text{et} \qquad \sum_{i=0}^{\lfloor log_2(b) \rfloor} b_i \geq 2 \tag{3.6}$$

Avec une telle contrainte sur chacun des deux facteurs, si aucune solution, à part la solution triviale 1 et c, n'existe, c'est-à-dire si c est un nombre premier, la résolution ne trouvera pas de solution.

Pour finir, des étapes de prétraitement peuvent permettre de fixer la valeur de certaines variables. Par exemple, on peut facilement vérifier que le nombre à factoriser est impair. Si c'est le cas, on peut fixer $a_0 = 1$ et $b_0 = 1$. Si ce n'est pas le cas, on peut diviser le nombre à factoriser par 2. De plus, si des tailles maximales pour les facteurs à trouver sont spécifiées par l'utilisateur, on peut aussi vérifier les bits de poids forts potentiels de a et b et fixer certaines de leurs valeurs. Par exemple, si les tailles des facteurs sont trop grandes par rapport à celle du nombre à factoriser, on peut fixer des bits de poids fort à 0.

3.4 Implémentation et expériences

Dans cette section, nous décrivons, d'une part, l'implémentation du programme qui génère la formulation choisie et la résout avec le solveur spécifié, et les différentes expériences conduites.

3.4.1 Implémentation

L'implémentation est écrite en C++ et peut utiliser Gurobi 3.0.1, COIN-OR CBC 2.4 ou GLPK 4.43 comme solveur tiers de PLNE. Ces solveurs utilisent un algorithme de type *Branch and Cut* [10] pour résoudre les PLNE. CBC et GLPK sont compilés avec ICC 11.1, alors que Gurobi est distribué sous forme de bibliothèques précompilées.

Le programme crée un fichier dans le format LP[9] à partir du nombre à factoriser. L'utilisateur peut spécifier la taille maximale des facteurs à trouver, si cette information est connue. Dans le cas contraire, des bornes sur les tailles des facteurs peuvent

[9] http://www.lix.polytechnique.fr/ liberti/teaching/xct/cplex/reffileformatscplex.pdf

être déduites du nombre à factoriser et ces informations sont ajoutées au problème. La propagation des bits dont les valeurs ont été déduites est laissée à la discrétion de la phase de prétraitement du solveur utilisé. Le programme appelle alors le solveur spécifié pour résoudre le problème.

3.4.2 Expériences

Nos expériences considèrent des nombres semi-premiers avec des facteurs de la même taille comme nombres à factoriser, ce qui correspond à la forme des nombres RSA.

Pour générer des instances où le nombre à factoriser est le produit de deux facteurs premiers de même taille en binaire, on utilise le crible d'Atkin [5]. Celui-ci génère la liste des nombres premiers jusqu'à une certaine valeur fixée. Nous choisissons alors aléatoirement dans cette liste deux nombres premiers de la taille voulue et on vérifie que leur produit correspond bien à la taille recherchée. Dans ce cas, le produit est gardé, sinon le processus est répété. Pour les expériences qui s'intéressent aux temps d'exécution, chaque valeur reportée est la moyenne des temps d'exécution de 10 instances. Les résultats complets concernant ces expériences se trouvent en annexe A.1.

Conditions expérimentales

Les expériences sont réalisées sur un ordinateur qui fonctionne sous Linux (noyau 2.6.31) et qui dispose d'un processeur Intel Core i7 920 2,66GHz avec 4×256Ko de cache de niveau 2 et 8Mo de cache de niveau 3 partagé, ainsi que de 6Go de RAM.

Limites de la première formulation

Cette expérience (figure 3.1) reflète les limitations pratiques de la formulation $(F_1^{'})$ (voir section 3.3.1) vis-à-vis des trois solveurs utilisés : Gurobi, GLPK and CBC.

Les résultats sont clairement différents pour les trois solveurs. Pour GLPK, les instances au-delà de 34 bits ne sont plus résolues correctement. Pour CBC, tout se passe correctement jusqu'à 40 bits, où seulement 2 instances sur 10 sont résolues correctement (en fait, les 2 dont les nombres à factoriser sont les plus petits). Pour Gurobi, la limite apparaît à 24 bits où, pour certaines instances, des instabilités numériques ont lieu. A partir de 30 bits, plus aucune instance testée n'a été résolue correctement avec Gurobi. En effet, la solution trouvée a une valeur de fonction objectif qui est bien 0, mais elle n'est pas réalisable. Les solutions à partir de 50 bits ont une valeur de fonction objectif différente de 0.

Comparaison des performances des deux formulations

Cette expérience (figure 3.2 ou tableau A.1) est une comparaison en termes de temps d'exécution entre les formulations $(F_1^{'})$ et $(F_2^{'})$. Comme CBC et GLPK sont les solveurs qui permettent de résoudre les instances les plus grandes pour la formulation $(F_1^{'})$, seuls ces deux solveurs sont ici utilisés. En effet, Gurobi ne permet

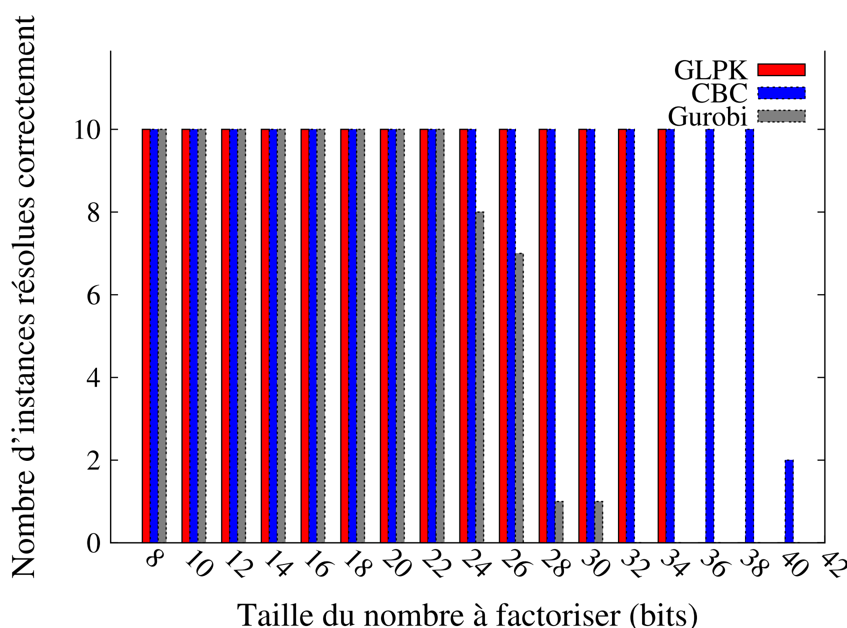

Figure 3.1: Nombre d'instances, sur 10, dont le résultat retourné est correct pour la formulation $(F_1^{'})$, avec GLPK, CBC et Gurobi.

d'obtenir des résultats fiables que jusqu'à 22 bits, ce qui est trop faible dans le cadre de cette comparaison. Rappelons que CBC (resp. GLPK) permet de factoriser des nombres de 38 bits (resp. 34 bits) avec la formulation $(F_1^{'})$. Il est important de noter que CBC est ici utilisé en mode séquentiel (c'est-à-dire uniquement un *thread*), pour effectuer une comparaison juste avec GLPK qui ne dispose pas d'une implémentation parallèle.

Comme on peut le voir, pour les deux solveurs CBC et GLPK, il y a une forte pénalité en termes de temps de calcul à utiliser la formulation $(F_2^{'})$ au lieu de la formulation $(F_1^{'})$. De plus, comme $(F_2^{'})$ a beaucoup plus de variables et de contraintes que $(F_1^{'})$, l'étape de prétraitement des solveurs peut avoir un impact élevé sur les performances. En effet, pour des tailles suffisamment grandes (ici, après 24 bits), des variations très élevées peuvent apparaître pour des instances de même taille. Pour les deux formulations, GLPK est de très loin plus rapide que CBC, excepté pour les instances de taille 36 bits dans la formulation $(F_2^{'})$.

Performances de la seconde formulation

Cette dernière expérience (figure 3.3 ou tableau A.2) utilise des instances de tailles plus élevées que précédemment pour montrer que la formulation $(F_2^{'})$ fonctionne bien au-delà de 38 bits, qui était la limite de la formulation $(F_1^{'})$ pour CBC et qui était utilisée comme taille d'instance maximale dans l'expérience précédente, ainsi que les performances de formulation dans ce contexte. Les temps d'exécution correspondent

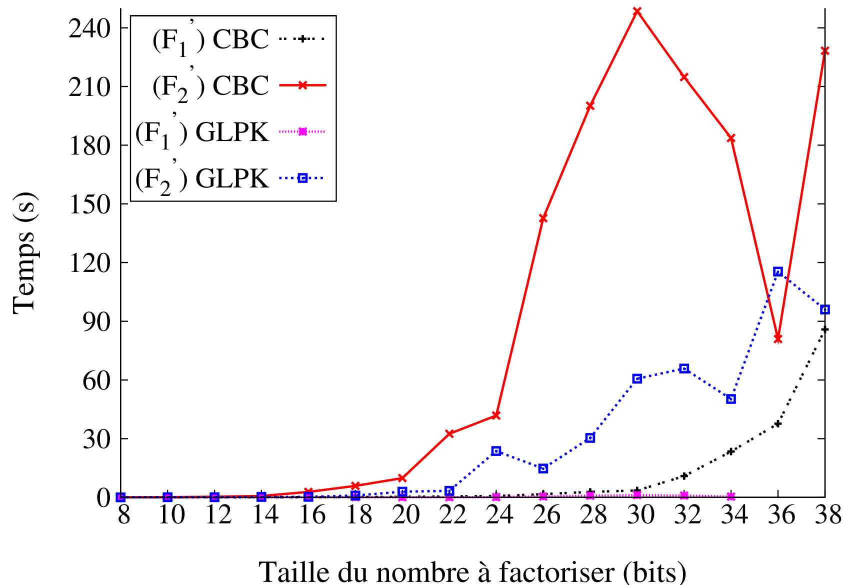

Figure 3.2: Comparaison des temps d'exécution de (F_1') et (F_2') avec CBC séquentiel et GLPK sur un Core i7 920. Résultats issus de la moyenne de 10 instances.

cette fois aux versions parallèles de CBC et de Gurobi pour pouvoir observer le comportement des résolutions sur des instances plus grandes et dans des conditions plus réalistes (en particulier, l'utilisation de solveurs parallèles sur des ordinateurs dotés de processeurs multi-coeurs). Le processeur utilisé est un Core i7 920 qui dispose de 4 coeurs, chacun gérant 2 coeurs logiques. Ainsi, ce processeur peut gérer jusqu'à 8 *threads* simultanément de manière matérielle. Par conséquent, CBC et Gurobi ont tous deux été exécutés avec 8 *threads*. Même si les 8 coeurs ne sont pas physiques mais uniquement logiques, la résolution avec 8 *threads* est généralement plus rapide qu'avec 4. Il faut cependant noter qu'utiliser 8 *threads* au lieu de 4 demande plus de mémoire.

Tout d'abord, la formulation (F_2') permet de résoudre toutes les instances de toutes les tailles considérées. Il est intéressant de noter que CBC ne lance pas les 8 *threads* dès le début de la résolution par *Branch and Cut*. En effet, CBC a besoin d'effectuer plusieurs relaxations continues avant de lancer un nouveau *thread*. Pour les plus grandes instances considérées ici, plusieurs secondes peuvent être nécessaires avant le lancement des 8 *threads*. Pour les instances les plus petites, le temps d'exécution peut être plus rapide avec l'exécution séquentielle, même si en moyenne ce n'est pas le cas. Cependant, pour les instances de taille élevée, le bénéfice d'utiliser plusieurs *threads* en lieu et place d'un seul est très clair : on peut observer une accélération superlinéaire, c'est-à-dire qu'utiliser p coeurs à la place d'un seul permet d'être plus que p fois plus rapide. Néanmoins, la version parallèle de CBC a pour gros

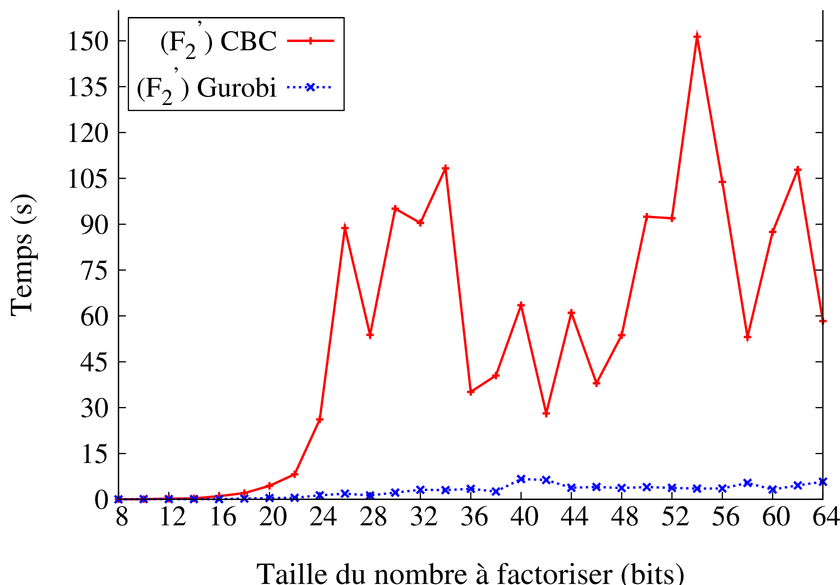

Figure 3.3: Performance de (F_2') avec CBC parallèle et Gurobi parallèle (8 *threads*) sur un Core i7 920. Résultats issus de la moyenne de 10 instances.

désavantage de ne pas être déterministe dans son exécution, c'est-à-dire qu'une même instance peut être résolue en des temps très variables. Ce phénomène accroît sensiblement les variations dans les temps d'exécutions qui étaient déjà importantes dans l'expérience séquentielle après 24 bits. Nous pouvons noter qu'au contraire, la version parallèle de Gurobi a une exécution déterministe et, une fois la première relaxation obtenue, les 8 *threads* se lancent simultanément. Dans cette expérience, Gurobi est beaucoup plus rapide que CBC. Nous pouvons noter que Gurobi en mode séquentiel serait plus rapide que CBC en mode parallèle pour les instances et la formulation considérées.

3.5 Conclusion

Dans ce chapitre, nous avons modélisé le problème de la factorisation entière en nombres premiers sous forme de PLNE, de telle sorte qu'un solveur classique puisse être utilisé pour sa résolution. En l'occurrence, une première formulation a été décrite, qui a l'avantage d'être simple, mais qui ne permet pas de travailler sur des nombres à factoriser suffisamment grands, à cause de certaines limitations au niveau des nombres flottants. Une seconde formulation a été apportée pour corriger cet inconvénient majeur, mais en contrepartie, cette dernière est plus lente à résoudre. Différentes expériences ont été conduites et analysées pour montrer les limites de la première formulation et comparer les performances des deux formulations, en

utilisant différents solveurs de PLNE classiques.

Ce travail permet de mettre en évidence l'importance de la modélisation sur la résolution, non seulement pour les performances mais aussi pour les limites des différents solveurs. En particulier, nous avons vu que l'utilisation de nombres à virgule flottante peut engendrer des instabilités numériques ou des limites dans ce qui peut être stocké et utilisé pour des calculs. Ces problèmes surviennent pour des tailles d'instances différentes suivant les solveurs et une nouvelle modélisation a été nécessaire pour pouvoir utiliser ces solveurs performants sur les instances considérées.

CHAP. 4

Compressive sensing – Approche par programmation linéaire

Dans ce chapitre nous décrivons une méthode de résolution exacte pour le problème du *compressive sensing* (CS). La méthode proposée repose sur des outils de *programmation linéaire* et plus particulièrement sur une modification de la décomposition de Dantzig-Wolfe spécifique au CS. Celle-ci permet un gain substantiel en performance par rapport aux approches exactes classiques.

L'intérêt majeur d'une méthode exacte pour ce problème est de permettre de comparer d'un point de vue qualitatif les différentes méthodes approchées. En effet, dans la pratique, une résolution approchée est très souvent suffisante et a l'avantage de trouver une solution proche de la solution optimale particulièrement rapidement. Cependant, comparer des solutions approchées issues de différentes méthodes est complexe sans solution de référence. L'intérêt d'un tel travail est donc méthodologique pour la communauté de recherche autour du CS.

Ce chapitre commence par une introduction rapide au problème du CS (4.1). La décomposition classique de Dantzig-Wolge est alors décrite dans un contexte général (4.2.1), pour ensuite être appliquée au CS (4.2.2). La section 4.3 traîte alors des modifications apportées à cette décomposition pour obtenir une nouvelle décomposition plus efficace dans le cadre du CS. Enfin, des expériences comparatives sont réalisées et analysées dans la section 4.4.

Ce travail a mené à la publication d'un article accepté à la conférence *SiPS* [21] ainsi qu'à celle d'un article dans la revue *TCS* [22].

4.1 Description du problème

Avant d'introduire le *compressive sensing* (CS), intéressons-nous au problème plus général du *basis pursuit*. Le problème du *basis pursuit* [36] correspond à minimiser la norme l_1 d'un signal sous certaines contraintes linéaires. Etant donné une matrice

$A \in \mathbb{R}^{m\times n}$ et des données observées $f \in \mathbb{R}^m$, on cherche la solution $u^* \in \mathbb{R}^n$ du problème de minimisation suivant :

$$\begin{cases} \min_u \|u\|_1 \\ s.c.\ Au = f \end{cases} \tag{4.1}$$

où le nombre de contraintes m est beaucoup plus petit que la taille n du signal à reconstruire (en pratique le plus souvent entre 4 et 32 fois, suivant les instances). Ce problème a reçu beaucoup d'attention grâce à ses applications dans différents domaines, tels que le traitement du signal et d'image, la compression de données, et plus récemment le *compressive sensing* [29–31, 50, 115]. Pour cette dernière application, des matrices spécifiques sont utilisées et ont pour propriété de permettre une reconstruction exacte du signal original, supposé creux, dans une base donnée, à partir du signal observé. Plus précisément, ces matrices doivent vérifier la propriété d'isométrie restreinte, c'est-à-dire que toutes leurs valeurs propres appartiennent à $[1-\nu, 1+\nu]$ pour $\nu > 0$ et proche de 0. De telles matrices sont proposées dans [7, 29, 31, 47, 50]. Nous pouvons citer entre autres les matrices Gaussiennes, les matrices de Bernoulli, ainsi que les matrices de DFT ou DCT partielles. Ces matrices de contraintes sont *denses*. De plus, la solution optimale est *unique*.

Pour un aperçu des méthodes de résolution approchée couramment utilisées pour résoudre le CS, voir la section 5.1.

4.2 Décomposition de Dantzig-Wolfe

La méthode proposée repose sur la décomposition de Dantzig-Wolfe. Dans cette section, nous introduisons tout d'abord d'un point de vue général cette décomposition classique, puis cette dernière est appliquée au problème du CS.

4.2.1 Cas général

Dans cette section, nous décrivons la méthode classique de décomposition de Dantzig-Wolfe [43, 44, 89]. Tout d'abord, nous décrivons les grandes lignes de cette méthode de résolution. L'idée principale est de reformuler un programme linéaire dont la matrice de contraintes est angulaire par blocs en un programme linéaire équivalent. Ce nouveau programme est alors résolu par *génération de colonnes*. Dans le programme linéaire reformulé, aussi dénommé *programme maître* (PM), chaque solution est une combinaison convexe des points extrêmes du polyèdre défini par les contraintes du programme linéaire original. Ainsi, le nombre de variables (c'est-à-dire de colonnes) dans la reformulation est égal au nombre de sommets de ce polyèdre, c'est-à-dire un nombre exponentiel par rapport au nombre de variables du programme original. C'est pourquoi pour résoudre efficacement le très grand programme linéaire engendré, un processus itératif à deux niveaux est utilisé (génération de colonnes). Les paragraphes suivants expliquent en détail cette méthode de décomposition.

A chaque itération, un sous-problème du programme maître est considéré en ne prenant en compte qu'un petit sous-ensemble de ses variables (colonnes). A

partir de ce problème restreint, des sous-problèmes généralement solubles en temps polynomial sont utilisés pour obtenir les *multiplicateurs du simplexe* optimaux. Ceux-ci sont alors utilisés pour générer une colonne dont le *coût réduit* est négatif (c'est-à-dire, qui va permettre d'obtenir une meilleure solution à l'itération suivante), qui va alors permettre de résoudre les nouveaux sous-problèmes correspondants. En répétant ce processus, l'optimum est atteint quand il n'existe plus de colonne de coût réduit négatif [44].

La décomposition de Dantzig-Wolfe tire parti de la structure particulière de la matrice de contraintes du programme à résoudre. Cependant, il n'en résulte pas systématiquement un gain au niveau des performances. En effet, cette méthode peut être dans certains cas plus lente qu'une résolution directe par *simplexe*. De plus, le phénomène d'amélioration rapide de la solution au début du processus puis de ralentissement significatif au fur et à mesure de l'optimisation est bien connu. D'un point de vue pratique, cette méthode peut aussi être complexe à implémenter suivant le problème à résoudre. Cependant, un certain nombre de problèmes bénéficient clairement de cette approche [53,89,112]. Un regain d'intérêt pour la méthode de décomposition de Dantzig-Wolfe a eu lieu avec l'avènement de la résolution par *Branch and Price* dans le cadre de la programmation linéaire en nombres entiers [8,9,117].

La décomposition de Dantzig-Wolfe provient de la reformulation d'un programme linéaire dont la matrice de contraintes est angulaire par blocs. Soit $x \in \mathbb{R}_+^n$ les variables de ce programme et $c \in \mathbb{R}^n$ les coefficients de sa fonction objectif linéaire. Les contraintes peuvent être de deux types : couplantes ou non-couplantes. Si elles sont couplantes, c'est-à-dire qu'elles corrèlent un grand nombre de variables, elles sont représentées par $Ax = b$ où A est une matrice de taille $m \times n$. Si elles sont non-couplantes, elles sont représentées par l'un des l blocs angulaires indépendants des contraintes, notés ici $D_{(i)}$ et travaillant sur les variables vectorielles $x_i \in \mathbb{R}_+^{k_i}$ où $n = \sum_{i=1}^{l} k_i$. Avec ces dernières notations, $x = (x_1, \ldots, x_l)^T$ et $c = (c_1, \ldots, c_l)^T$. Le programme linéaire initial peut ainsi être formulé de la sorte :

$$\begin{cases} \min_x c^T x \\ s.c.\ Ax = b \\ \qquad D_{(i)} x_i = b_i \\ x \geq 0 \end{cases} \tag{4.2}$$

La matrice de contraintes est alors de la forme suivante :

$$\begin{pmatrix} A_{(1)} & \ldots & A_{(l)} \\ D_{(1)} & & \\ & \ddots & \\ & & D_{(l)} \end{pmatrix}$$

Notons $x(i)$ la $i^{\text{ème}}$ composante du vecteur x et $A_{(i)}$ le bloc de colonnes de la matrice A qui correspond aux vecteurs x_i. A partir du programme linéaire original (4.2), la reformulation issue de la décomposition de Dantzig-Wolfe donne un programme linéaire équivalent avec pour vecteur-variables u tel quel $\forall i, u(i) \in \mathbb{R}_+$.

Le programme maître de la décomposition de Dantzig-Wolfe s'écrit alors [44] :

$$\text{(Programme Maître)} \begin{cases} \min_u \sum_j \theta(j)u(j) \\ s.c. \;\; \sum_j \gamma_j u(j) = b \\ \qquad \sum_j u(j) = 1 \\ u(j) \geq 0 \end{cases} \tag{4.3}$$

où $\gamma_j = Ap_j$ et $\theta(j) = c^T p_j$, avec $p_j \in \mathbb{R}^n$ les points extrêmes du polyèdre défini par les contraintes non-couplantes du programme linéaire (4.2) (c'est-à-dire les contraintes $D_{(i)}x_i = b_i$).

Notons que la contrainte $\sum_j u(j) = 1$ avec $\forall j, u(j) \geq 0$ du programme linéaire (4.3) est une contrainte de convexité. Cela signifie que les solutions du programme linéaire original (4.2) s'écrivent au niveau du programme maître comme des combinaisons convexes des points extrêmes p_j.

Il est intéressant de remarquer que le nombre de variables du programme linéaire (4.3) est le même que le nombre de points extrêmes du polyèdre de contraintes de (4.2), c'est-à-dire un nombre exponentiel par rapport au nombre de variables de (4.2). De plus, le programme linéaire (4.3) possède une contrainte de plus. Ainsi, au lieu d'utiliser une résolution classique par simplexe qui serait beaucoup plus lente sur le programme maître (4.3) que sur le programme original (4.2), une approche par génération de colonnes est utilisée.

Intéressons-nous au processus de génération de colonnes. Soit S_i le polyèdre convexe borné défini par :

$$S_i = \{x_i | D_{(i)}x_i = b_i, x_i \in \mathbb{R}_+^{k_i}\}$$

L'ensemble S est le produit cartésien de tous les S_i et l'ensemble de tous les points extrêmes du programme (4.2). S est un ensemble polyédrique convexe borné.

Soit μ les multiplicateurs du simplexe associés à la contrainte de convexité du programme (4.3) (c'est-à-dire de la contrainte $\sum_j u(j) = 1$) et π le vecteur des multiplicateurs du simplexe associés aux contraintes restantes. Pour choisir la variable $u(\bar{j})$ qui entre dans la base (c'est-à-dire la nouvelle colonne à générer à partir du point extrême $p_{\bar{j}}$), on considère la règle usuelle de Dantzig. Cela correspond à choisir la variable avec le coût réduit minimum. Ici, cela signifie que la variable $u(\bar{j})$ correspond au point extrême $p_{\bar{j}}$ solution de :

$$\min_{p_j} (c - \pi A)^T p_j - \mu \tag{4.4}$$

En partant de l'hypothèse que le programme (4.4) est un problème d'optimisation borné, ces solutions optimales sont des points extrêmes de son ensemble réalisable et le programme (4.4) est alors équivalent à :

$$\begin{cases} \min_x \langle c - \pi^T A, x \rangle \\ s.c. \; D_{(i)}x_i = b_i, \forall i \\ x_i \geq 0, \forall i \end{cases} \tag{4.5}$$

La solution du problème (4.5) est le point extrême qui correspond à la variable qui entre en base. La pivot du simplexe classique est utilisé pour obtenir la variable qui sort de la base.

La fonction objectif du programme (4.5) est additivement séparable et ses contraintes sont indépendantes. Ainsi, le problème (4.5) peut être transformé en les l sous-problèmes indépendants suivants :

$$\forall i \in \{1, \ldots, l\}, \begin{cases} \min_{x_i} \langle c_i - \pi^T A_{(i)}, x_i \rangle \\ s.c.\ D_{(i)} x_i = b_i \\ x_i \geq 0 \end{cases} \tag{4.6}$$

Le fait que les l sous-problèmes soient indépendants est de la plus haute importance car c'est cette propriété qui permet de les résoudre efficacement en parallèle, sans la moindre communication. A chaque itération, les sous-problèmes (4.6) peuvent être généralement calculés en temps polynomial grâce à leur forme particulière. Ainsi, on peut obtenir un élément particulier d'un ensemble de taille exponentielle en temps polynomial.

Supposons qu'une base réalisable initiale pour le programme (4.3) est connue. Notons B la matrice de base associée, B^{-1} son inverse et (π, μ) les multiplicateurs du simplexe. A_i est $i^{\text{ème}}$ colonne de la matrice A. L'algorithme suivant utilise la décomposition de Dantzig-Wolfe pour résoudre le problème original (4.2) au travers du problème reformulé (4.3) :

Algorithme 1 Algorithme de Dantzig-Wolfe [44]

---

1. Résoudre les l sous-problèmes (4.6) en utilisant les multiplicateurs du simplexe π. Nous obtenons ainsi leurs l solutions optimales $\hat{x}(i)$ ainsi que leurs l valeurs $w(i)$ de fonction objectif.
2. Calculer $w = \sum_{i=1}^{n} w(i) - \mu$.
3. Si $w \geq 0$, une solution optimale est trouvée. Cette solution du problème (4.2) est alors : $\sum_{u(j)>0} u(j) p_j$.
4. Si $w < 0$, la nouvelle colonne à entrer en base est :

$$\begin{pmatrix} \sum_i A_i \hat{x}(i) \\ 1 \end{pmatrix}$$

5. Après multiplication par B^{-1}, appliquer l'opération de pivot du simplexe pour obtenir la variable qui quitte la base. La nouvelle matrice de base B est alors obtenue ainsi que son inverse B^{-1} et les nouveaux multiplicateurs du simplexe (π, μ).
6. Répéter à partir de l'étape 1.

La nouvelle base obtenue d'une itération sur l'autre ne diffère de la précédente que par une unique colonne. Cette caractéristique permet de calculer B^{-1} en multipliant la matrice carrée η suivante par la précédente matrice de base inverse :

$$\begin{pmatrix} 1 & & & -\frac{d(1)}{d(r)} & & & \\ & \ddots & & \vdots & & & \\ & & 1 & -\frac{d(r-1)}{d(r)} & & & \\ & & & \frac{1}{d(r)} & & & \\ & & & -\frac{d(r+1)}{d(r)} & 1 & & \\ & & & \vdots & & \ddots & \\ & & & -\frac{d(m+1)}{d(r)} & & & 1 \end{pmatrix}$$

où $d = B^{-1} \times col$ avec col la colonne de la variable qui entre en base. Nous voyons que la matrice η est formée par la somme d'une matrice unitaire de taille $(m+1) \times (m+1)$ et d'une matrice avec uniquement une colonne non-nulle de la même taille. Grâce à la forme particulière de la matrice η, cette multiplication peut être réalisée en seulement $O(m^2)$ opérations. Ceci réduit considérablement le temps nécessaire à l'obtention de la nouvelle matrice inverse par rapport à un calcul classique d'inversion de matrice (ainsi que par rapport à une multiplication classique matrice/matrice). Pour de plus amples détails sur ce procédé standard du simplexe, le lecteur peut se référer à [43].

Si le programme maître (4.3) est non-dégénéré, chaque itération décroît strictement la valeur de la fonction objectif. Comme il existe un nombre fini de bases possibles et qu'aucune n'est répétée, la méthode permet d'obtenir une solution optimale en un nombre fini d'itérations. La solution est alors exprimée comme combinaison convexe des points extrêmes du programme linéaire (4.2). En outre, comme pour le simplexe, l'algorithme est en temps exponentiel par rapport à la taille de l'entrée. La complexité d'une itération est $O(mn)$.

4.2.2 Application au *compressive sensing*

Dans cette section, le problème du CS est reformulé de telle sorte que la décomposition de Dantzig-Wolfe décrite dans la section 4.2.1 puisse être utilisée. Cela signifie que le programme linéaire doit disposer d'une matrice de contraintes angulaire par blocs et que les solutions des sous-problèmes engendrés forment un ensemble polyédrique convexe borné.

La formulation (4.1) du problème du CS se réécrit selon le programme linéaire suivant [19] :

$$\text{(CS Linéaire)} \begin{cases} \min_y \langle y, \mathbb{1} \rangle \\ s.c.\ Ax = f \\ \qquad x \leq y \\ \quad -x \leq y \\ x \in \mathbb{R}^n, y \in \mathbb{R}^n_+ \end{cases} \tag{4.7}$$

La décomposition de Dantzig-Wolfe présuppose que l'ensemble des solutions des sous-problèmes induits sont des ensembles polyédriques convexes explicitement

bornés (comme décrit dans la section 4.2.1). Cependant, dans notre cas, cette supposition n'est pas vérifiée, comme l'ensemble S engendré par le programme (4.7) n'est pas borné. Ainsi, la méthode ne peut pas être directement utilisée. Les problèmes de minimisation de norme l^1 (et de norme l^∞) sont connus pour avoir ce problème dans le cadre de méthodes de décomposition. En effet, les variables $x(i)$ sont libres (et $y(i)$ non-négatives) donc la notion de points extrêmes n'a pas de sens. Cependant, $Ax = f$ peut être résolu pour obtenir une solution initiale x^0. Alors, une borne K sur les variables $y(i)$ peut en être déduite en prenant la valeur $||x^0||_1$. Dans le cas d'une minimisation de l^∞, K peut alors prendre la valeur $||x^0||_\infty$. Remarquons que chaque variable $y(i)$ pourrait être bornée indépendamment, même si ici l'on dispose d'une seule et même borne pour toutes les variables. Dans tous les cas, on obtient alors un programme linéaire avec des variables bornées et la notion de points extrêmes est explicitement et naturellement définie, ce qui va nous permettre d'utiliser la décomposition de Dantzig-Wolfe.

Réordonnons les variables $x(i)$ et $y(i)$ de la sorte :

$$(x(0), y(0), x(1), y(1), \ldots, x(n), y(n))$$

Cela permet d'obtenir une matrice de contraintes de la forme :

$$\begin{pmatrix} A_{1,1} & 0 & \ldots & A_{1,n} & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ A_{m,1} & 0 & \ldots & A_{m,n} & 0 \\ 1 & -1 & & & \\ -1 & -1 & & & 0 \\ 0 & 1 & & & \\ & & \ddots & & \\ & & & 1 & -1 \\ 0 & & & -1 & -1 \\ & & & 0 & 1 \end{pmatrix}$$

Cette matrice de contraintes possède la structure angulaire par blocs qui est nécessaire à l'utilisation de la décomposition de Dantzig-Wolfe (voir section 4.2.1). Cette matrice est formée de la contrainte couplante $A^{'}$ et de n contraintes angulaires par blocs identiques. En l'occurrence, la matrice $A^{'}$ est la matrice A dans l'espace des variables réordonnées, c'est-à-dire $A^{'} = A \otimes [1, 0]$. De plus, chaque bloc de contraintes incorpore en son sein une contrainte de borne sur ses variables.

En suivant la reformulation (4.3), on a $\gamma_j \in \mathbb{R}^{2n}$ et $\theta \in \mathbb{R}^{2n}$ tels que $\gamma_j = A^{'} p_j$ et $\theta(j) = c^T p_j = \sum_i p_j(i)$ (simplifiés grâce à la norme l^1, qui implique que $\forall i \in \{1, \ldots, n\}, c(2i-1) = 0$ et $c(2i) = 1$). Nous obtenons alors le programme suivant en variables $u(j) \in \mathbb{R}^{2n}$:

$$\begin{cases} \min_u \sum_j \theta(j) u(j) \\ s.c. \ \sum_j \gamma_j u(j) = f \\ \qquad \sum_j u(j) = 1 \\ u_j \geq 0 \end{cases} \tag{4.8}$$

Décrivons maintenant le processus de génération de colonnes pour le problème du CS. Soit S_i l'ensemble polyédrique convexe borné qui correspond au $i^{\text{ème}}$ bloc angulaire de contraintes, c'est-à-dire :

$$S_i = \{x(i), y(i) | x(i) - y(i) \leq 0, x(i) + y(i) \geq 0, y(i) \leq K, x(i) \in \mathbb{R}, y(i) \in \mathbb{R}_+\}$$

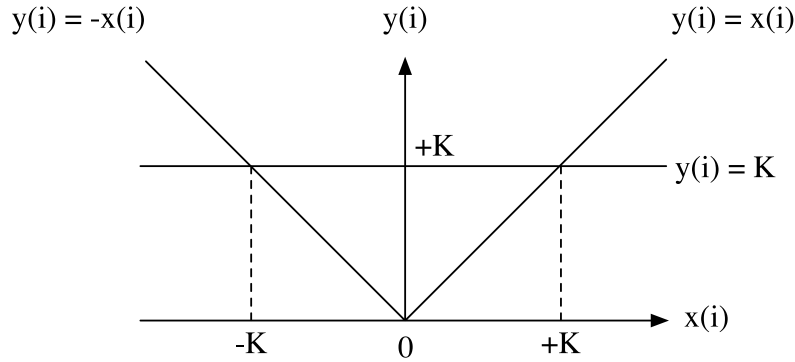

Figure 4.1: Représentation graphique de l'ensemble S_i.

Pour faciliter la compréhension, l'ensemble S_i est représenté graphiquement sur la figure 4.1. L'ensemble L est le produit cartésien de tous les S_i et forme un ensemble polyédrique convexe borné.

Dans notre cas de minimisation de la norme l^1, les n programmes linéaires indépendants (4.6) qui expriment les sous-problèmes de la décomposition peuvent se simplifier en la formule suivante :

$$\forall i \in \{1, \dots, n\}, \begin{cases} (0,0) & \text{si } |\bar{x}(i))| \leq 1 \\ (-sg(\bar{x}(i)) \times K, K) & \text{sinon} \end{cases} \tag{4.9}$$

où $\bar{x}(i)$ est le coût réduit de la variable $x(i)$ et $sg(x) = 1$ si $x \geq 0$, -1 sinon. La formule (4.9) peut générer 3 valeurs différentes par sous-problème : $(0,0)$, $(-K, K)$, (K, K). Comme il y a n sous-problèmes à résoudre à chaque itération, l'ensemble L des solutions possède 3^n éléments. Les vecteurs de L sont notés p_j. Il est à remarquer que la solution donnée par la formule (4.9) est calculée à chaque itération en temps linéaire (n sous-problèmes indépendants en temps constant). Cette amélioration est spécifique au CS. Rappelons qu'en général une telle solution s'obtient en temps polynomial (voir section 4.2.1). La formulation (4.7) possède une matrice de contraintes *creuse* de taille $(m + 2n) \times (2n)$ et la formulation (4.8) possède une matrice de contraintes *dense* de taille $(m + 1) \times (3^n)$.

Comme les variables $y(i)$ sont uniquement utilisées pour exprimer la norme l^1 de la fonction objectif de manière linéaire, les seules variables d'intérêt au niveau des contraintes sont les variables $x(i)$. Ainsi, l'ensemble polyédrique convexe borné S_i qui correspond aux n blocs de contraintes peut être changé en :

$$S_i = \{x(i) | -K \leq x(i) \leq K, x(i) \in \mathbb{R}\}$$

La formule (4.9) définit désormais l'ensemble L de la sorte : $L = \{-K, 0, K\}^n$. Alors, p_j et γ_j peuvent désormais appartenir à $\mathbb{R}^n$ au lieu de $\mathbb{R}^{2n}$. Pour cela, γ_j est dorénavant défini par $\gamma_j = Ap_j$. Comme $p_j \in \mathbb{R}^n$, l'expression de $\theta(j)$ doit être elle aussi adaptée. Les composantes de c, c'est-à-dire les coefficients de la fonction objectif, sont égaux à 0 pour les variables $x(i)$ et à 1 pour les variables $y(i)$, et $y(i) = |x(i)|$. Avec $p_j \in \mathbb{R}^n$, il vient $\theta(j) = \sum_i |p_j(i)|$. Désormais, $\forall j, p_j \in \mathbb{R}^n, \theta(j) = \sum_i |p_j(i)|$ et $\gamma_j = Ap_j$.

Comme un grand nombre de programmes linéaires rencontrés en pratique, le programme linéaire (4.2) est dégénéré. Cela vient du fait que sa solution optimale est elle-même dégénérée. En effet, celle-ci dispose d'une unique solution optimale qui dispose de moins de m valeurs non-nulles, où m correspond au nombre de contraintes. Cela implique que les algorithmes de résolution basés sur le simplexe pourraient cycler, c'est-à-dire ne pas converger. Une analyse des situations où le simplexe peut cycler est effectuée dans [63]. En pratique et malgré leurs dégénérescences, de tels programmes linéaires peuvent tout de même être résolus de manière efficace en utilisant un simplexe prenant en compte l'existence de dégénérescence. Dans notre cas, en pratique, la dégénérescence apparaît uniquement quand l'optimum est atteint : l'énergie décroît strictement jusqu'à l'optimum. Cependant, la convergence n'est pas détectée, bien qu'atteinte, et le processus ne s'arrête pas de lui-même (c'est-à-dire que la condition $w \geq 0$ de l'étape 3 de l'algorithme de Dantzig-Wolfe présenté dans la section 4.2.1 n'est jamais vérifiée) alors que la valeur de la fonction objectif ne diminue plus. Ce problème peut être résolu en ajoutant un test sur la stricte décroissance de la valeur prise par la fonction objectif. Si ce test est satisfait, les conditions de Karush-Kuhn-Tucker (KKT) [82,86] peuvent être vérifiées dans le but de s'assurer que la solution est bien optimale. En pratique, le test seul sur la stricte décroissance de la fonction objectif suffit pour obtenir la solution optimale.

Décrivons maintenant succinctement comment obtenir une solution de base réalisable initiale pour le problème (4.8). Une telle solution peut être calculée à partir d'une solution de base réalisable de $Ax = f$. En effet, chaque solution de $Ax = f$ peut être représentée comme une solution u de $\sum_i (Ap_i)^T u(i) = f$ avec $\sum u(i) = 1$ et $\forall i = 1, \ldots, 3^n, u(i) \geq 0$, c'est-à-dire une solution du programme maître. Résoudre $Ax = f$ se fait généralement en introduisant m variables d'écart $z(i)$ (une par contrainte) et en minimisant $\sum_i z_i$. Un simplexe peut alors être utilisé pour résoudre ce problème dont la solution de base initiale est $z = f$ et $x = (0, \ldots, 0)^t$. Comme on sait que $Ax = f$ dispose d'au moins une solution, on peut résoudre $Ax = f$ en introduisant une seule variable d'écart z pour toutes les contraintes et en minimisant z. Utiliser une unique variable d'écart pour résoudre $Ax = f$ au lieu de m est généralement plus rapide et donne de meilleures bornes. Cependant, généralement la solution obtenue dispose de plus de 0 (c'est-à-dire potentiellement plus de dégénérescences) et ne dispose pas de solution initiale triviale. Néanmoins, comme la méthode de résolution est basée sur le simplexe, la qualité de la solution initiale a un impact important sur les performances. Le lecteur intéressé peut se référer à [26,43,89] pour plus de détails sur la manière de trouver une solution initiale pour l'algorithme du simplexe.
A partir d'une solution x^0 de $Ax = f$, une base initiale peut être obtenue en utilisant des éléments de L avec exactement une composante non-nulle. La solution de base

réalisable qui y correspond s'obtient facilement, ainsi que les multiplicateurs du simplexe (π, μ). Alors, l'algorithme de Dantzig-Wolfe de la section 4.2.1 peut être initialisé.

A chaque itération, la décomposition de Dantzig-Wolfe sur le problème du CS amène à explorer 3^n éléments, où n est la taille du signal. Même si l'exploration est réalisée implicitement en temps linéaire grâce à l'équation (4.9) la résolution par décomposition de Dantzig-Wolfe peut être particulièrement lente. En effet, les solutions disposent de nombreuses symétries et il faut un grand nombre d'itérations pour obtenir la solution optimale (voir section 4.4.2 pour plus de détails).

Rappelons que, dans cette section, nous avions besoin d'un ensemble de solutions des sous-problèmes explicitement borné. Une telle propriété a été induite d'une considération sur la solution initiale. Dans la section suivante, nous allons voir comment utiliser la manière spécifique dont nous avons défini l'ensemble de solutions borné des sous-problèmes pour améliorer la décomposition et ses performances.

4.3 Décomposition réduite

Dans cette section, nous présentons une modification de la méthode de décomposition de Dantzig-Wolfe précédemment appliquée au problème du CS (voir section 4.2.2). Cette variante permet de réduire drastiquement le nombre de bases à explorer et augmente ainsi les performances. De plus, cette nouvelle décomposition, dite *réduite*, permet d'utiliser toutes les règles classiques de pivot du simplexe.

4.3.1 Présentation de la décomposition proposée

L'idée principale consiste à décomposer le problème (4.7) de telle sorte que les solutions soient des combinaisons convexes de vecteurs avec au plus une composantes non-nulle (notées q_i) en lieu et place de points extrêmes généraux (notés p_i). Cette décomposition réduite de Dantzig-Wolfe permet alors de travailler sur seulement $2n + 1$ vecteurs de la forme précitée, ce qui décroît significativement les symétries du problème. En effet, les solutions du problème (4.8) sont exprimées comme combinaison convexe des éléments de $L = \{-K, 0, K\}^n$. Parmi tous les vecteurs de L, on considère seulement ceux avec au plus une composante non-nulle. En d'autres termes, ces vecteurs s'écrivent $(0, \ldots, 0, \pm K, 0, \ldots, 0)^t$ ou $(0, \ldots, 0)^t$. Ces vecteurs forment l'ensemble M, avec $|M| = 2n + 1$.

De plus, le nombre de points extrêmes considérés est réduit d'exponentiel à linéaire, ce qui permet d'utiliser toutes les règles classiques de pivot du simplexe, et non plus seulement une forme particulière de la règle de Dantzig tirant parti de la structure du problème (que nous nommerons ici règle de Dantzig-Wolfe). En effet, les règles de pivot du simplexe peuvent ici choisir parmi $2n$ colonnes différentes là où la règle de Dantzig-Wolfe sur la décomposition standard peut choisir parmi 3^n colonnes différentes, c'est-à-dire que potentiellement $\binom{2n}{m}$ bases peuvent être inspectées, à la place de $\binom{3^n}{m+1}$.

4.3.2 Equivalence avec le problème original

Rappelons que le problème initial du CS possède des contraintes $Ax = f$. En supposant l'absence de dégénérescence, toute combinaison convexe de vecteurs q_i avec m composantes non-nulles correspond à une solution de $Ax = f$ avec m composantes non-nulles, exactement comme pour la décomposition standard de Dantzig-Wolfe (voir section 4.2).

De plus, chaque solution de $Ax = f$ peut être représentée comme solution v de $\sum_{i=1}^{2n+1} v(i)Aq_i = f$ avec $\sum_{i=1}^{2n+1} v(i) = 1$ et $\forall i = 1, \ldots, 2n+1, v(i) \geq 0$, c'est-à-dire une solution de la décomposition réduite de Dantzig-Wolfe.
Rappelons que la matrice de base du programme (4.8) est de taille $(m+1) \times (m+1)$. La solution de base réalisable initiale x^0 du problème original, qui est solution de $Ax = f$, permet de calculer K de manière suivante : $K = ||x^0||_1$. K borne ainsi le problème au sens de la décomposition.
Notons δ_i le vecteur qui appartient à $\mathbb{R}^n$ avec toutes les composantes nulles sauf la $i^{\text{ème}}$ composante, qui est égale à 1. Alors, les éléments de M s'écrivent :

$$\begin{cases} q_{2i-1} = K\delta_i & \forall i = 1, \ldots, n, \\ q_{2i} = -K\delta_i & \forall i = 1, \ldots, n, \\ q_{2n+1} = (0, \ldots, 0)^t \end{cases} \tag{4.10}$$

La solution de base initiale v^0 pour (4.8) se calcule à partir de la solution de x^0 comme suit :

$$\begin{cases} v^0(2i-1) = \frac{\max(0, x^0(i))}{||x^0||_1} & \forall i = 1, \ldots, n, \\ v^0(2i) = \frac{\max(0, -x^0(i))}{||x^0||_1} & \forall i = 1, \ldots, n, \\ v^0(2n+1) = 0 \end{cases} \tag{4.11}$$

Pour toute itération t et pour tout $i \in \{1, \ldots, n\}$ on ne peut avoir $v^t(2i-1) \neq 0$ et $v^t(2i) \neq 0$ en même temps. Avec les q_i et $v^0(i)$ précédemment définis, $x^0 = \sum_{i=1}^{2n+1} v^0(i)Aq_i$ et v^0 possède exactement m composantes non-nulles.
Comme la borne K est calculée à partir de x^0, solution réalisable initiale utilisée par le processus d'optimisation, la décomposition réduite n'est valide que dans le cadre des solutions strictement meilleures que la solution réalisable initiale. En supposant l'absence de dégénérescence, les règles de pivot du simplexe usuelles assurent cette propriété à chaque itération et garantissent ainsi l'optimalité de la solution de la décomposition réduite au sens du problème original.
Il est important de noter que le vecteur nul est *nécessaire* au bon fonctionnement de la décomposition réduite et qu'il doit être présent dans la base à chaque itération.
Notons $x^t, t > 1$ la solution calculée à la $t^{\text{ème}}$ itération. Comme les règles de pivot garantissent des solutions meilleures au fur et à mesure des itérations, on a :

$$\forall t, \sum_i |x^{t+1}(i)| < \sum_i |x^t(i)| \tag{4.12}$$

En termes de solutions de la décomposition réduite, comme $x^t = \sum_{i=1}^{2n+1} v^t(i)Aq_i$,

cette dernière équation est équivalente à :

$$\forall t, \sum_{i=1}^{m} v^{t+1}(i)|Aq_i^{t+1}| < \sum_{i=1}^{m} v^t(i)|Aq_i^t| \tag{4.13}$$

et

$$\forall t, \sum_{i=1}^{m+1} v^t(i) = 1 \tag{4.14}$$

La suite $\{\sum_{i=1}^{m} v^t(i)\}$ est strictement décroissante car, pour tout i, la quantité $|Aq_i|$ est indépendante des itérations t. Ainsi, $v^t(m+1)$ agit comme un terme accumulateur et permet à $\sum_{i=1}^{m+1} v^t(i)$ de ne jamais dépasser 1 au fur à mesure de l'amélioration de la solution au sens de la norme l^1. La combinaison convexe est assurée durant le processus d'optimisation grâce au vecteur nul, qui est toujours présent dans la base. Cela montre que toute solution de base réalisable (sauf la solution initiale) a $v(2n+1) \neq 0$. Ainsi, le vecteur nul doit être en base à chaque itération. En d'autres termes, on force ce vecteur à rester en base durant toute l'optimisation.
Comme expliqué plus en détails dans la section suivante, le coût réduit du vecteur nul est 0. Ainsi, comme les règles de pivot ne considèrent que les variables strictement non-positives, le vecteur nul n'est jamais sélectionné. Si ce vecteur n'est pas forcé à rester dans la base durant l'optimisation, on obtient une solution optimale de la décomposition réduite qui est toujours réalisable dans le problème du CS original, mais pas nécessairement optimal.

Nous avons vu que les $2n+1$ éléments de M sont suffisants pour exprimer toutes les solutions du problème du CS original, si le vecteur nul est forcé à être en base. La section suivante montre la décomposition réduite aux vecteurs de M d'un point de vue des coûts réduits.

4.3.3 Point de vue des coûts réduits

Tous les éléments de L peuvent s'écrire comme la somme des éléments de M. Soit $p_{i,j}$ le vecteur de L avec exactement deux composantes non-nulles qui est la somme de $q_i \in L$ et $q_j \in L$. Nous avons $s_{q_i} = -1$ ou $s_{q_i} = 1$ suivant le signe de la composante non-nulle du vecteur q_i. Le coût réduit $\bar{v}(i)$ de la variable $v(i)$ qui correspond au vecteur q_i de la décomposition est alors :

$$\bar{v}(i) = c^T q_i - \pi^T A q_i - \mu \tag{4.15}$$

Ce qui peut se réécrire en :

$$\bar{v}(i) = (c(i) - s_{q_i}\pi^T A_i)K - \mu \tag{4.16}$$

et le coût réduit $\bar{u}(i,j)$ of $p_{i,j}$ de la formulation décomposée est alors :

$$\bar{u}_{i,j} = (c(i) + c(j)) - \pi^T (s_{q_i} A_i + s_{q_j} A_j)K - \mu \tag{4.17}$$

$$\bar{u}(i,j) = \bar{v}(i) + \bar{v}(j) + \mu \tag{4.18}$$

Selon la section 4.3.2, on a $\mu = 0$ comme le vecteur nul est toujours en base. La généralisation aux vecteurs avec plus de deux composantes non-nulles est simple.

Les variables de base ont un coût réduit de 0. Grâce à la propriété (4.18), on sait qu'à chaque élément de L (c'est-à-dire chaque point extrême de la décomposition originale de Dantzig-Wolfe) de coût réduit négatif, il existe au moins un élément de M (c'est-à-dire un point extrême avec exactement une composante non-nulle) de coût réduit lui aussi négatif. Plus précisément, trois cas peuvent survenir :

(i) $(\bar{v}(i) \geq 0$ et $\bar{v}(j) \geq 0) \Leftrightarrow \bar{u}(i,j) \geq 0$, c'est-à-dire si aucun élément de M ne peut être choisi, aucun élément de L ne peut être choisi non plus.

(ii) $(\bar{v}(i) \leq 0$ et $\bar{v}(j) \leq 0) \Leftrightarrow \bar{u}(i,j) \leq 0$, c'est-à-dire si un élément de L peut être choisi, l'élément $v(\arg\min(\bar{v}(i), \bar{v}(j)))$ de M peut aussi être choisi.

(iii) $(\bar{v}(i) \leq 0$ et $\bar{v}(j) \geq 0)$, l'élément $v(\arg(\bar{v}(i)))$ de M peut être choisi.

Ces trois cas nous montrent que, si un point extrême avec plus d'une composante non-nulle a un coût réduit strictement négatif (autrement dit, peut être choisi), alors au moins un point extrême avec exactement une composante non-nulle peut être choisi aussi. Si ce point extrême n'existe pas, alors il n'existe pas non plus de point extrême de coût réduit strictement négatif et ainsi la solution optimale est trouvée.

En conclusion, toute solution réalisable de la décomposition classique de Dantzig-Wolfe strictement meilleure que la solution initiale est aussi solution réalisable de la décomposition réduite.

4.3.4 Conséquences sur les règles de pivot

Dans la décomposition de Dantzig-Wolfe classique, à chaque itération la règle de pivot de Dantzig-Wolfe est utilisée pour trouver une nouvelle variable (colonne) qui va entrer en base parmi les 3^n éléments. Comme dans la décomposition réduite, ce nombre est réduit à $2n+1$, les règles classiques de pivot du simplexe peuvent être utilisées efficacement. Le tableau 4.1 liste les règles de pivot les plus répandues. En ce qui concerne la variable sortante, le test de ratio minimum classique est utilisé [43]. Si plusieurs solutions sont possibles, celle d'indice minimum est choisie.

règle de pivot	variable entrante
Dantzig	$\arg\min_i \{\bar{v}(i) \mid \bar{v}(i) < 0\}$
Steepest edge	$\arg\min_i \{\frac{\bar{v}(i)}{\sqrt{1+\sum_j (B^{-1} \times A)_{ij}}} \mid \bar{v}(i) < 0\}$
Bland	$\min\{i \mid \bar{v}(i) < 0\}$

Tableau 4.1: Règles classiques de pivot du simplexe

Nous présentons ici ces trois règles dans un contexte général et dans le cadre de la décomposition réduite :

- *Règle de Dantzig [42] :* la règle de Dantzig correspond à la règle utilisée par l'algorithme du simplexe original, c'est-à-dire que la variable choisie pour entrer dans la base est celle de coût réduit le plus négatif. Cela correspond à visiter $O(n)$ variables, c'est-à-dire $O(nm)$ opérations pour calculer tous les coûts réduits.
  Pour la décomposition réduite, le cas (ii) de la section 4.3.3 amène à choisir un point extrême qui peut être moins intéressant d'un point de vue du coût réduit que celui qui aurait été choisi dans le cadre de la décomposition classique de Dantzig-Wolfe. Cela vient du fait qu'une itération du simplexe fait entrer une nouvelle variable et en fait sortir une autre. Comme une variable dans la décomposition classique de Dantzig-Wolfe correspond à la somme de plusieurs variables dans la décomposition réduite, atteindre la même amélioration sur la solution demande plusieurs itérations successives. Cependant, comme les règles de pivot du simplexe travaillent localement, itération après itération, une base dans la décomposition réduite à une itération donnée a une plus grande liberté pour améliorer la solution. En effet, grâce à la diminution significative des symétries dans les représentations des solutions et dans le nombre de bases à visiter, moins d'itérations sont requises pour obtenir la solution optimale. En d'autres termes, en moyenne une itération améliore plus la solution dans la décomposition réduite que dans la décomposition classique de Dantzig-Wolfe.

- *Steepest Edge [59, 67] :* cette règle est généralement la plus efficace. Par rapport à la règle de Dantzig, elle a cependant les désavantages de demander plus de temps de calcul pour obtenir la variable entrante et de nécessiter plus de mémoire. En effet, cette règle nécessite tous les coefficients de la matrice $B^{-1} \times A$, où B est la matrice de base courante. En contrepartie, cette règle amène à une variable entrante de meilleure qualité que celle retournée par la règle de Dantzig, ce qui permet de diminuer drastiquement le nombre d'itérations requises pour obtenir la solution optimale. Sa complexité en temps est la même que celle de la règle de Dantzig : $O(n)$ variables sont visitées pour $O(nm)$ opérations. Nous pouvons aussi noter l'existence d'une variante de cette règle qui utilise une approximation pour obtenir plus rapidement la variable entrante : Devex [73].
  Dans le cadre de la décomposition réduite, les mêmes commentaires que pour la règle de Dantzig s'appliquent ici, pour les mêmes raisons.

- *Règle de Bland [18] :* la règle de Bland correspond à choisir la variable d'indice le plus faible ayant un coût réduit strictement négatif. Cette règle permet d'éviter que le simplexe cycle et assure ainsi un nombre fini d'itérations, même en présence de dégénérescence. La complexité en pire cas de cette règle est la même que celle de Dantzig, c'est-à-dire que toutes les variables doivent être inspectées. Cependant, cette règle demande en pratique très peu de temps pour calculer la variable entrante (seul un très faible nombre de variables

est effectivement inspecté), mais cette dernière est généralement d'une faible qualité.
Dans le cadre de la décomposition, on peut voir qu'utiliser la règle de Bland sur la décomposition classique de Dantzig-Wolfe est équivalent à utiliser cette règle sur la décomposition réduite, si le bon ordre sur les variables est choisi. Choisissons alors un ordre arbitraire, tel que la propriété suivante sur les points extrêmes de la décomposition classique de Dantzig-Wolge soit respectée : les éléments sont triés par ordre croissant de leur nombre de composantes non-nulles. A partir des trois cas de la section 4.3.3, on voit que, si une variable doit être choisie parmi les 3^n, elle sera choisie parmi les $2n + 1$ premières variables, c'est-à-dire un élément de M. Etant donné cet ordre et cette règle, cela montre bien que ces $2n + 1$ vecteurs sont suffisants pour obtenir la solution optimale.

4.3.5 Coûts calculatoires

Les opérations les plus consommatrices en temps dans la décomposition classique de Dantzig-Wolfe sont celles liées au calcul de la variable entrante et au calcul du nouveau vecteur γ associé à cette variable (calculée par multiplication matrice-vecteur). Ces deux opérations on une complexité en temps de $O(mn)$. Dans la décomposition réduite, le calcul de γ a une complexité de $O(m)$ comme le vecteur de taille n de la multiplication matrice-vecteur est réduit à une unique valeur non-nulle. Les 3 règles de pivot du tableau 4.1 ainsi que la règle de Dantzig-Wolfe ont la même complexité en pire cas de $O(n)$ variables à visiter, chacune nécessitant $O(m)$ opérations à chaque itération pour calculer son coût réduit. De plus, la règle *Steepest Edge* requiert de maintenir le tableau du simplexe itération après itération, c'est-à-dire de calculer $B^{-1} \times A$. Nous pouvons noter que cette matrice peut être mise à jour d'une itération sur l'autre au lieu de la recalculer entièrement à chaque fois, de la même manière que pour B^{-1} (voir section 4.2.1). Cela permet ainsi d'obtenir cette matrice en seulement $O(mn)$. Comme A est ici dense, calculer cette matrice peut être particulièrement coûteux et en pratique on verra que le bénéfice d'une meilleure variable entrante (et donc d'un nombre d'itérations plus faible) n'est pas suffisant pour compenser le temps de calcul plus élevé par rapport à la règle de Dantzig.

Nous pouvons noter que, pour les règles qui nécessitent de calculer tous les coûts réduits (c'est-à-dire toutes celles présentées ici, à part la règle de Bland), $2n$ variables sont invariablement inspectées à chaque itération, mais pour un coût qui peut se ramener à seulement n calculs de coûts réduits. En effet, l'opération la plus consommatrice en temps dans le calcul du coût réduit est le produit scalaire πA_i (voir formule (4.16)). Ce calcul est indépendant du signe et peut ainsi être réalisé une seule fois pour les deux vecteurs dont la seule différence est le signe.

Jusqu'ici, A a toujours été décrite explicitement. En d'autres termes, tous les coefficients de la matrice A sont stockés en mémoire et les calculs sur celle-ci sont réalisés par des opérations matricielles classiques. Dans le contexte du CS, la matrice A peut être une sous-matrice d'une transformée telle que la transformée de Fourier discrète (DFT) ou la transformée en cosinus discrète (DCT). Ainsi, des transformées rapides peuvent être utilisées en lieu et place des opérations matricielles. Comme

la complexité en temps pour calculer une transformée de Fourier rapide (FFT) est $O(n \log n)$ [12], cela permet de réduire la complexité d'une itération de $O(mn)$ à $O(\max(m^2, n\ log\ n))$. De plus, cela économise en mémoire la place dédiée au stockage de A.

La décomposition classique de Dantzig-Wolfe permet de calculer les n sous-problèmes en parallèle sans communication grâce à leur propriété d'indépendance (voir section 4.2.1). De plus, toutes les opérations en $O(mn)$ dans la décomposition réduite peuvent être effectuées en parallèle (multiplication matrice-vecteur et pré-multiplication par la matrice η pour la règle *Steepest Edge*). Dans le cas de l'utilisation de FFT, celles-ci peuvent aussi être réalisées en parallèle.

4.4 Implémentation et expériences

Dans cette section, nous présentons l'implémentation réalisée, ainsi que les différentes expériences conduites pour valider notre approche.

4.4.1 Implémentation

La plupart des solveurs de programmes linéaires travaillent sur des représentations creuses de matrices et disposent d'algorithmes rapides adaptés à celles-ci. Ainsi, beaucoup de travaux ont été conduits dans ce sens. Pour une vue d'ensemble des différents aspects théoriques et pratiques pour la résolution de programmes linéaires, le lecteur peut s'intéresser à [17].

Dans le cadre du CS, les matrices sont denses. Cependant, la linéarisation (4.7) de la forme la plus couramment rencontrée pour le CS fait apparaître une matrice de contraintes creuse, pour laquelle les implémentations classiques du simplexe sont adaptées. Néamoins, les décompositions présentées n'impliquent, elles, que des matrices denses de taille maximum $m \times n$ et des calculs d'au plus $O(mn)$ opérations sur celles-ci. Contrairement au cas creux, toutes ces opérations peuvent facilement être implémentées de manière efficace. En particulier, cela permet de tirer parti de la vectorisation et de certaines optimisations au niveau du cache. Ainsi, en plus des améliorations théoriques discutées dans la section 4.3.5, l'implémentation de notre décomposition réduite tire parti de ces optimisations. De plus, un schéma hybride *float-double* est utilisé dans le but d'augmenter les performances, tout en évitant toute instabilité numérique. En effet, utiliser uniquement des flottants simple précision amène à des instabilités. Pour résoudre ce problème, seules les parties de l'algorithme les plus consommatrices en temps (c'est-à-dire celles en $O(mn)$) utilisent des flottants simple précision. Les autres utilisent des flottants double précision. Toutes ces optimisations sont aussi mises en oeuvre au niveau de l'implémentation de la décomposition classique de Dantzig-Wolfe. L'implémentation est écrite en C++ et est compilée avec ICC (Intel C Compiler) 11.1 et les options d'optimisation classiques.

Dans le cas des DCT, une version utilisant des transformées rapides a aussi été implémentée. Les DCT partielles sont alors calculées par des FFT complexe-complexe avec des étapes de pré et post-traitements en $O(n)$ pour convertir les résultats à partir

ou vers des réels. En effet, les implémentations de DCT (donc réelles) sont sujettes à des problèmes de performance et ne sont donc pas utilisées ici. Utiliser des FFT complexe-complexe pour calculer des DCT implique un surcoût de 4 fois la mémoire requise par rapport aux DCT classiques, mais permet de conserver des temps de calculs raisonnables. Les calculs de FFT sont effectués grâce à la bibliothèque FFTW 3.2.2, compilée avec les options par défaut. Plus de détails sur cette bibliothèque peuvent se trouver dans [62].

4.4.2 Expériences

Conditions expérimentales

L'ordinateur utilisé pour les expériences dispose d'un processeur Intel Core i7 920 2,66GHz avec 4×256Ko de cache de niveau 2 et 8Mo de cache de niveau 3 partagé, ainsi que 6Go de RAM et fonctionne sous Linux (noyau 2.6.31). Comme point de référence, 3 solveurs classiques de programmation linéaire sont utilisés : GLPK 4.43[1] et COIN-OR CLP 1.11[2] qui sont deux solveurs *open source* compilés avec ICC 11.1 et les options classiques d'optimisation, et Gurobi 3.0[3] qui est un solveur commercial précompilé. Nous comparons notre méthode au schéma de résolution le plus rapide pour ces 3 solveurs sur la formulation (4.7) : le simplexe dual avec la règle *Steepest Edge*, sans *scaling* ni prérésolution. Sur des problèmes similaires et en utilisant un autre solveur de programmes linéaires, [61] a obtenu cette même variante du simplexe comme la plus efficace.

Nous considérons deux types de matrices explicites pour le CS : des matrices Gaussiennes orthogonalisées et des matrices de DCT partielles. Les premières ont leurs éléments générés en utilisant des distributions normales $\mathcal{N}(0, 1)$ et leurs lignes sont orthogonalisées. Les secondes sont générées en choisissant aléatoirement, avec échantillonnage uniforme, m lignes d'une matrice de DCT complète.

Dans le cadre des expériences liées à des représentations matricielles, seules celles réalisées avec des matrices Gaussiennes orthogonalisées sont présentées, étant donné que celles avec des matrices de DCT partielles montrent un comportement identique et qu'en général, pour ces dernières, des transformées rapides sont utilisées.

Les signaux f ont $\frac{1}{10}^{\text{ème}}$ de composantes non-nulles qui sont soit -1, soit $+1$. Deux ratios $\frac{m}{n}$ sont utilisés : $\frac{1}{16}$ et $\frac{1}{8}$. Les résultats présentés correspondent à la moyenne des résultats obtenus à partir de 10 instances différentes.

Il est à noter que toutes les figures ont un axe des ordonnées dont l'échelle est logarithmique. Les résultats concernant les expériences (temps d'exécution et nombre d'itérations) pour les deux types de matrices de contraintes (Gaussiennes orthogonalisées et DCT partielles) et les deux types de représentations pour les DCT partielles (matrices et transformées rapides) sont regroupés sous forme de tableaux dans l'annexe A.2.

---

[1] http://www.gnu.org/software/glpk
[2] https://projects.coin-or.org/Clp
[3] http://www.gurobi.com

Matrices Gaussiennes orthogonalisées

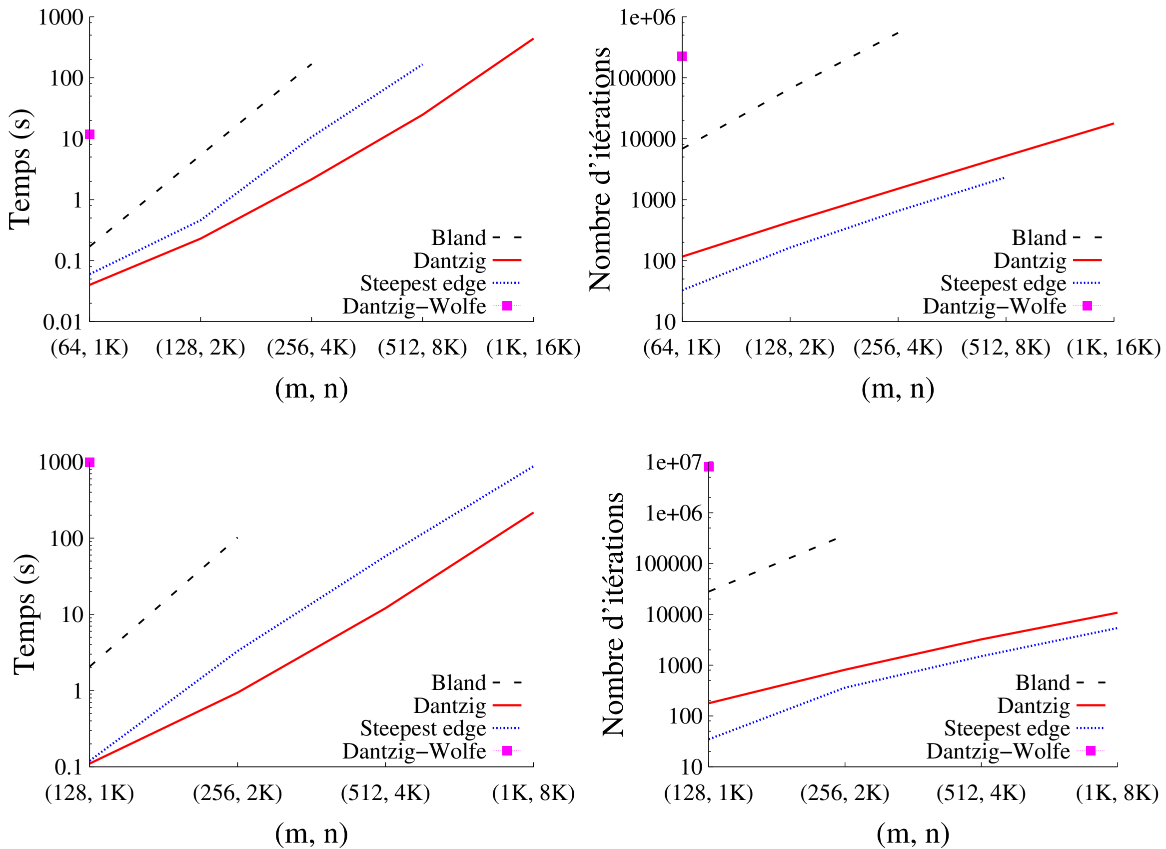

Figure 4.2: Comparaison entre notre décomposition réduite avec différentes règles de pivot et la décomposition classique de Dantzig-Wolfe. Résultats issus de la moyenne de 10 instances utilisant des matrices Gaussiennes orthogonalisées avec différents ratios $\frac{m}{n}$ (à gauche : temps (s); à droite : nombre d'itérations; première ligne : $\frac{m}{n} = \frac{1}{16}$; seconde ligne : $\frac{m}{n} = \frac{1}{8}$). Les résultats ne sont pas présentés quand le temps d'exécution dépasse 1500 secondes.

La figure 4.2 montre comment les différentes décompositions se comportent : la décomposition classique de Dantzig-Wolge est comparée à la décomposition réduite avec les règles de pivot décrites dans la section 4.3.4. Nous observons qu'en termes de temps de calcul et de nombre d'itérations, la décomposition classique de Dantzig-Wolfe est le compétiteur le plus faible. Pour la décomposition réduite, la règle *Steepest Edge* engendre moins d'itérations que la règle de Dantzig, qui elle-même requiert moins d'itérations que la règle de Bland. En termes de temps d'exécution, la règle de Dantzig donne de meilleurs résultats que la règle de Bland. Cependant, même si pour les plus grosses instances considérées, la règle *Steepest Edge* diminue le nombre d'itérations d'un facteur légèrement supérieur à 2 par rapport à la règle de Dantzig, elle demande 4 à $6,6$ fois plus de temps d'exécution au total. Ainsi, une itération demande 8 à 15 fois plus de temps avec cette règle qu'avec la règle de Dantzig. En

effet, comme le CS utilise des matrices denses et que la méthode considérée ne fait apparaître que des matrices elles aussi denses, la règle de pivot *Steepest Edge* ne peut tirer parti des optimisations spécifiques au cas creux et est alors très consommatrice en temps d'exécution. Le gain en nombre d'itérations n'est ici pas suffisant pour contrebalancer cette perte et son utilisation n'est donc pas avantageuse par rapport à la règle de Dantzig. Au final, dans le cadre de notre décomposition réduite, la règle la plus intéressante en termes de nombre d'itérations est *Steepest Edge* et la plus intéressante en termes de temps d'exécution est la règle de Dantzig.

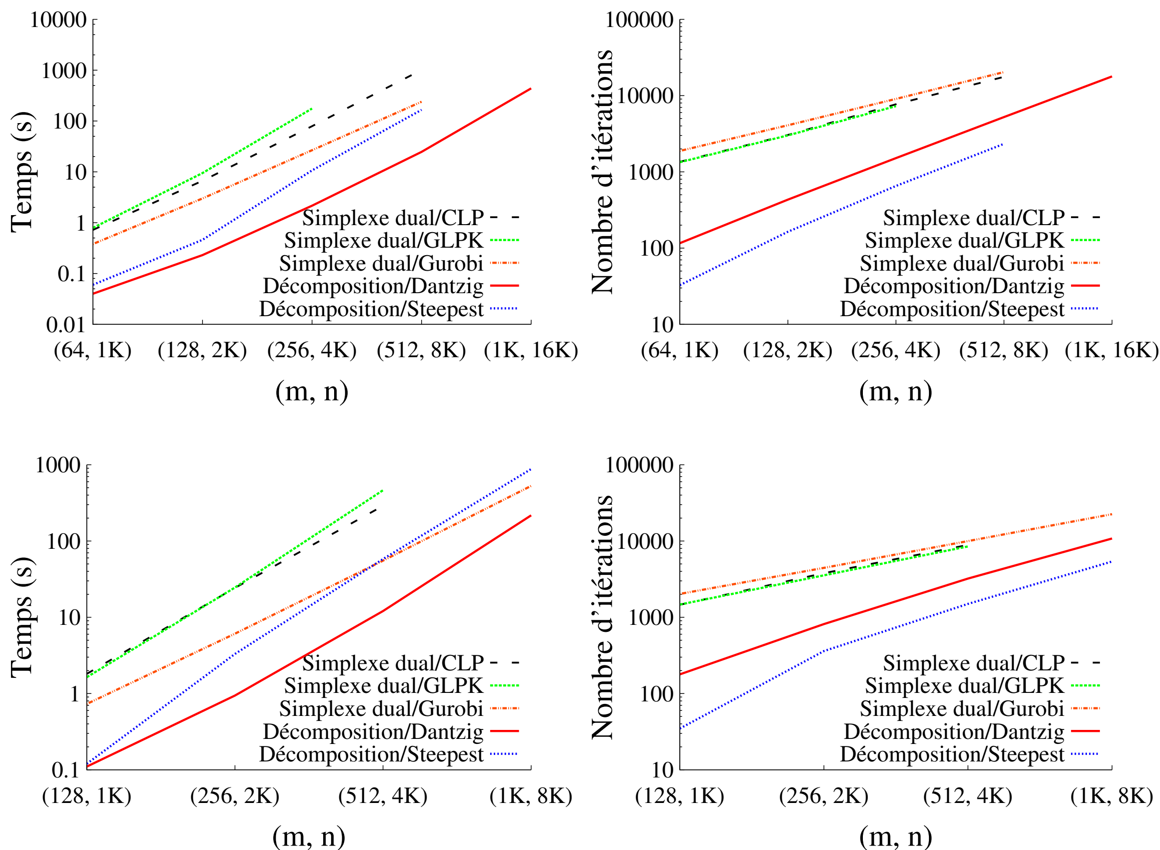

Figure 4.3: Comparaison entre notre décomposition réduite avec ses meilleures règles de pivot et les simplexes duaux de GLPK, CLP et Gurobi avec leur meilleure règle de pivot (*Steepest Edge*). Résultats issus de la moyenne de 10 instances utilisant des matrices Gaussiennes orthogonalisées avec différents ratios $\frac{m}{n}$ (à gauche : temps (s); à droite : nombre d'itérations; première ligne : $\frac{m}{n} = \frac{1}{16}$; seconde ligne : $\frac{m}{n} = \frac{1}{8}$). Les résultats ne sont pas présentés quand le temps d'exécution dépasse 1500 secondes.

La figure 4.3 montre une comparaison entre notre décomposition réduite avec ses meilleures règles de pivot et les simplexes duaux de GLPK, CLP et Gurobi avec leur meilleure règle de pivot (*Steepest Edge*). Sur les instances considérées, les

trois simplexes duaux montrent un comportement similaire en termes de nombre d'itérations. Gurobi nécessite cependant légèrement moins d'itérations pour atteindre l'optimum, mais demande surtout beaucoup moins de temps que GLPK et CLP. CLP est globalement un peu plus rapide que GLPK et le plus grand saut se situe pour $(m, n) = (256; 4096)$, où CLP est à peu près 2 fois plus rapide que GLPK (même si ce dernier a besoin de légèrement moins d'itérations pour obtenir la solution optimale). Gurobi est de très loin plus rapide que CLP : pour $(m, n) = (512; 8192)$ Gurobi est $4,2$ fois plus rapide que CLP et pour $(m, n) = (512; 4096)$ il est $5,2$ fois plus rapide. En termes de nombre d'itérations et de temps d'exécution la décomposition réduite avec la règle de Dantzig donne toujours de meilleurs résultats que les simplexes duaux de GLPK, CLP et Gurobi. Néanmoins, même si la décomposition réduite avec la règle *Steepest Edge* est toujours plus rapide que les simplexes duaux de GLPK et CLP, Gurobi est plus rapide pour $\frac{m}{n} = \frac{1}{16}$. Il est aussi intéressant de noter que les simplexes duaux sont toujours plus rapides que la décomposition classique de Dantzig-Wolfe. Ainsi, en termes de nombre d'itérations et de temps de calcul, la décomposition standard de Dantzig-Wolfe est la plus lente des méthodes considérées.
En termes de temps d'exécution, notre décomposition réduite avec la règle de Dantzig est le meilleur compétiteur pour toutes les tailles d'instances considérées. En effet, pour $(m, n) = (512; 4096)$, notre décomposition réduite avec la règle de Dantzig est $4,5$ fois plus rapide que le simplexe dual de Gurobi et demande $3,1$ fois moins d'itérations. La décomposition réduite est aussi $24,1$ fois plus rapide que CLP et demande $2,8$ fois moins d'itérations. Pour $(m, n) = (512; 8192)$, la décomposition réduite est $9,7$ fois plus rapide que Gurobi (son meilleur opposant) et a besoin de $3,9$ fois moins d'itérations pour obtenir la solution optimale. En considérant CLP à la place de Gurobi, la décomposition réduite est $40,9$ fois plus rapide et demande $3,4$ fois moins d'itérations.
Il est intéressant de noter que pour $(m, n) = (1024; 8192)$ Gurobi est seulement $2,4$ fois plus lent que notre décomposition réduite mais utilise 877Mo de mémoire là où notre décomposition utilise seulement 327Mo avec la règle de Dantzig et 462Mo avec *Steepest Edge*. Pour $(m, n) = (1024; 16384)$, Gurobi utilise $1,7$Go de mémoire, alors que notre décomposition réduite utilise 841Mo avec la règle de Dantzig et 879Mo avec *Steepest Edge*.

Le plus faible nombre d'itérations par rapport au simplexe dual montre les qualités intrinsèques de notre décomposition réduite. L'amélioration en temps d'exécution par itération montre surtout l'avantage lié aux optimisations permises au niveau de l'implémentation par cette décomposition. La comparaison entre la décomposition classique de Dantzig-Wolfe et notre décomposition réduite permet d'observer les conséquences de la réduction significative des symétries dans l'expression des solutions, et ainsi du nombre de bases potentielles à explorer. De plus, on peut noter que, pour un m donné, notre décomposition réduite semble bien se comporter quand n augmente, c'est-à-dire que, quand n augmente d'un facteur 2, le temps d'exécution augmente d'un facteur constant (autrement dit la courbe s'approche d'un droite).

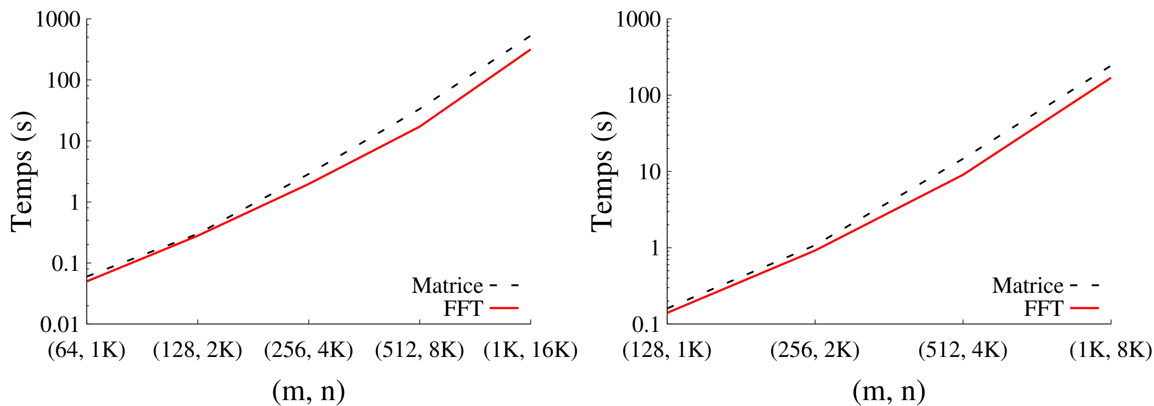

Figure 4.4: Comparaison du temps d'exécution (en secondes) entre matrices de DCT partielles et FFT pour la décomposition réduite avec la règle de Dantzig. Résultats issus de la moyenne de 10 instances pour différents ratios $\frac{m}{n}$ (à gauche: $\frac{m}{n} = \frac{1}{16}$; à droite: $\frac{m}{n} = \frac{1}{8}$). Les résultats ne sont pas présentés quand le temps d'exécution dépasse 1500 secondes.

DCT partielles

La figure 4.4 compare les temps d'exécution de la décomposition réduite avec sa règle la plus rapide - la règle de Dantzig - pour les deux modes de représentation/calcul des DCT partielles : matrices et transformées rapides. Pour les instances les plus petites, les deux représentations donnent les mêmes résultats. Cependant, quand n atteint 4096, la situation change et le bénéfice d'utiliser des FFT à la place de multiplications matrice-vecteur devient tangible. Pour les petites instances, le surcoût lié aux FFT (voir section 4.3.5) n'est pas masqué par leur calcul plus rapide ($O(n \log n)$ à la place de $O(mn)$). De plus, si le ratio $\frac{m}{n}$ n'est pas suffisamment petit, le coût du test de ratio minimum pour obtenir la variable sortante (en $O(m^2)$) peut être significatif, comparé aux calculs de FFT (en $O(n \log n)$). Dans ce cas, l'amélioration globale peut ne pas être aussi élevée que pour les ratios les plus petits.
Nous pouvons noter que l'utilisation de FFT donne de meilleurs résultats pour la règle de Dantzig, car celle-ci est adaptée à l'utilisation de transformées rapides. En effet, l'utilisation de transformées rapides permet de calculer *tous* les coûts réduits en $O(n \log n)$, au lieu de $O(mn)$, mais elles n'ont pas la versatilité de calculer seulement un certain nombre de ceux-ci efficacement. Rappelons que la règle de Bland sélectionne la variable de coût réduit négatif d'indice le plus petit. En pratique, seul un faible nombre de variables est inspecté, et ainsi seulement peu de coûts réduits sont calculés. Implémenter la règle de Bland avec des transformées rapides aurait un coût en temps de calcul identique à celui de la règle de Dantzig (qui requiert, elle, tous les coûts réduits) pour une variable de qualité plus faible, ce qui ne serait pas intéressant.

Au final, la décomposition réduite avec la règle de Dantzig est la méthode la plus rapide considérée ici. Aucune instance considérée ici n'a amené notre méthode de

résolution à cycler. Cependant, dans le cas où cette situation se produirait la règle de Bland pourrait toujours être utilisée, pour assurer un nombre fini d'itérations.

4.5 Conclusion

Dans ce chapitre, nous avons présenté une modification de la décomposition classique de Dantzig-Wolfe spécifique au CS et qui engendre une solution exacte. Notre décomposition réduite repose sur une nouvelle représentation des solutions, qui engendre moins de symétries. Cette représentation permet non seulement de visiter potentiellement beaucoup moins de bases, mais également d'améliorer la complexité de certains calculs, ou de diminuer certaines de leurs constantes. Grâce aux propriétés du CS et de la décomposition, seules des matrices denses sont rencontrées, ce qui permet de mettre en oeuvre facilement un certain nombre d'optimisations au niveau de l'implémentation, telles que la vectorisation. Grâce à tous ces avantages, le nombre d'itérations, ainsi que le temps de calcul par itération est significativement réduit par rapport à la décomposition classique. De plus, notre décomposition réduite permet d'utiliser efficacement toutes les règles classiques du simplexe, ce qui n'est pas le cas de la décomposition standard. En l'occurrence, notre décomposition réduite permet d'obtenir de meilleurs temps de calcul en utilisant la règle de Dantzig et surpasse toutes les autres approches exactes considérées. Enfin, notre décomposition réduite associée à cette règle de pivot dispose de la flexibilité d'utiliser des transformées rapides à la place de calculs matriciels, ce qui est particulièrement adapté à certaines instances du CS. Grâce à ses performances, elle permet donc de construire des bases de comparaison au niveau de la qualité des solutions des méthodes approchées bien plus rapidement qu'avant.

Dans le chapitre suivant, nous nous intéressons à une méthode de résolution approchée pour le CS qui a l'avantage d'être considérablement plus rapide que la méthode exacte présentée dans ce chapitre, tout en permettant d'obtenir une solution de très bonne qualité.

CHAP. 5

Compressive sensing – Approche par programmation convexe

Ce chapitre se situe dans la continuité du précédent. Nous proposons un algorithme de résolution approchée pour le problème du *compressive sensing* (CS). Celle-ci repose sur des outils de *programmation convexe*. En effet, en pratique, une résolution approchée permet d'obtenir très rapidement une solution de très bonne qualité. L'algorithme proposé a la particularité d'avoir été pensé dès sa conception pour tirer parti des architectures matérielles modernes. En l'occurrence, trois différents types de plateformes multi-coeurs ont été ici envisagées : des processeurs multi-coeurs disposant d'unités de calcul vectoriel (CPU), des processeurs graphiques (GPU) et le processeur Cell.

Une variante de cette approche, qui permet d'obtenir une solution exacte, est aussi décrite. Celle-ci n'est cependant efficace que pour un certain type de représentation de matrices.

Rappelons que le problème du CS a précédemment été décrit dans la section 4.1. La section 5.1 donne un aperçu des méthodes de résolution approchée pour le CS. En particulier, cette section cite rapidement les méthodes les plus classiques pour ensuite s'axer autour des travaux proches du type de méthode retenue ici. La section 5.2 décrit l'algorithme, les choix effectués et démontre sa correction. Dans la section 5.3, les différentes implémentations (suivant les architectures ciblées) et les différentes expériences réalisées sont décrites.

Ce travail a donné lieu à un article accepté à la conférence *ISVC* [24] et à un article actuellement soumis à la revue *JSPS*. Une version préliminaire de cet article est disponible en tant que rapport de recherche UCLA [23].

5.1 Etat de l'art des résolutions approchées

Il existe beaucoup d'algorithmes pour résoudre de manière approchée le problème du CS [39,49,58,68,70,83,113–115,125]. Le lecteur intéressé peut se référer à ces différents articles pour une vue d'ensemble de ces méthodes. La plupart des méthodes de résolution reposent sur des seuillages itératifs. Ces approches ont été initialement présentées par [104] et [34], et ont suscité un certain nombre d'améliorations, telles que [15, 27, 39, 45, 54, 55, 57]. La méthode de résolution proposée ici tombe elle aussi dans cette catégorie. Cependant, et contrairement aux approches précitées, nous nous axons principalement autour des considérations pratiques de ce type de méthodes (à l'opposition de considérations purement théoriques). En effet, notre but est de concevoir un algorithme efficace qui tire bénéfice des architectures matérielles modernes.

Nous pouvons noter qu'en général implémenter efficacement ces algorithmes pour des architectures parallèles est assez complexe à mettre en oeuvre, étant donné les types de calculs impliqués et les caractéristiques des plateformes (voir section 2.1). Notre algorithme a été conçu dès le début de telle sorte à prendre en compte toutes ces caractéristiques, dans le but d'être, d'une part, rapide, et d'autre part, facile à mettre en oeuvre.

De manière plus précise, pour notre algorithme, nous nous intéressons à la régularisation de Moreau-Yosida, proposée par [99] et [126]. Celle-ci engendre un algorithme itératif connu sous le nom d'*itérations du point proximal* [39, 52, 78, 91]. La convergence de l'approche est assurée par les propriétés de la régularisation de Moreau-Yosida, telle que décrite dans [78,91]. En particulier, cette méthode requiert l'inversion d'un opérateur non-linéaire. Nous montrerons que, dans notre cas, cette inversion peut être réalisée par un seuillage, si une régularisation de Moreau-Yosida appropriée est utilisée. Nous verrons que cette approche permet une implémentation efficace sur les architectures parallèles ici considérées.

5.2 Un algorithme basé sur le point proximal

Dans cette section, nous décrivons notre algorithme qui permet une implémentation efficace sur les architectures parallèles modernes. Notre approche consiste à utiliser un opérateur proximal sur une version pénalisée du problème du CS original (4.1). De cette manière, le schéma de minimisation est réduit à une série de multiplications matrice-vecteur et de minimisations séparables (c'est-à-dire des optimisations qui travaillent élément par élément). Ce processus est embarqué dans une série de mises à jour de la pénalité qui accélère l'approche.

5.2.1 Régularisation de Moreau-Yosida

Au lieu de résoudre le problème 4.1 original, nous voulons minimiser l'énergie suivante pour $u \in \mathbb{R}^n$:

$$E_\mu(u) = \|u\|_1 + \frac{\mu}{2}\|Au - f\|_2^2 \tag{5.1}$$

où μ est un paramètre non-négatif qui a pour rôle d'assurer la contrainte $Au = f$. Cette énergie correspond à une pénalisation, telle que décrite par [14]. Il est intéressant de noter qu'une solution de (5.1) est aussi solution de (4.1) pourvu que $\mu = \infty$. Cependant, dans le cadre d'une application pratique, un μ fini suffit comme les mesures f sont corrompues par un bruit ou qu'une certaine précision dans la solution suffit. Notons $u^\star$ un minimiseur de E_μ.

Introduisons les notations suivantes : étant donné une certaine valeur non-négative N, le produit scalaire dans $\mathbb{R}^N$ est noté $\langle \cdot, \cdot \rangle$ et sa norme associée $\| \cdot \|_2$. Supposons B un opérateur linéaire symétrique défini positif. Posons $\langle \cdot, \cdot \rangle_B = \langle B \cdot, \cdot \rangle$ et on note pour tout $x \in \mathbb{R}$, $\|x\|_B^2 = \langle x, x \rangle_B^2$. Introduisons maintenant la régularisation de Moreau-Yosida F_μ de E_μ associée à la métrique M comme suit [78, 91] : à tout point $u^{(k)} \in \mathbb{R}^n$,

$$F_\mu(u^{(k)}) = \inf_{u \in \mathbb{R}^n} \left\{ E_\mu(u) + \frac{1}{2} \|u - u^{(k)}\|_M^2 \right\} \tag{5.2}$$

En d'autres termes, cela correspond à inf-convoluer la fonction à minimiser avec la métrique M. Notons que F_μ est strictement convexe en tant que somme d'une fonction convexe et d'une fonction strictement convexe, et donc qu'il n'y a qu'un unique point qui minimise F_μ. De plus, il n'est pas difficile de montrer que l'infimum dans (5.2) est atteint et donc que l'on peut remplacer l'$\inf$ par un $\min$. D'après [99], cet unique minimiseur est aussi appelé point proximal de u par rapport à E_μ et M, et est noté $p_\mu(u^{(k)})$.

Etant donné une fonction $E : \mathbb{R}^n \to \mathbb{R}$, notons ∂E le sous-différentiel de E défini par $w \in \partial E(u) \Leftrightarrow E(v) \geq E(u) + \langle w, v - u \rangle$ pour tout v (voir [78, 78]). Ce point proximal $p_\mu(u)$ est caractérisé par l'équation d'Euler-Lagrange suivante :

$$0 \in \partial \left(E_\mu + \frac{1}{2} \| \cdot - u^{(k)} \|_M^2 \right) (p_\mu(u^{(k)}))$$

Cette dernière est équivalente à :

$$0 \in \partial E_\mu(p_\mu(u^{(k)})) + M(p_\mu(u^{(k)}) - u^{(k)})$$

Ce qui signifie que :

$$M u^{(k)} \in (M + \partial E_\mu) (p_\mu(u^{(k)}))$$

Comme $p_\mu(u^{(k)})$ est défini de manière unique, on obtient :

$$p_\mu \left(u^{(k)} \right) = (M + \partial E_\mu)^{-1} \left(M u^{(k)} \right) \tag{5.3}$$

L'idée de l'approche par régularisation de Moreau-Yosida pour minimiser E_μ est d'itérer la formule de mise à jour (5.3) jusqu'à convergence.

L'approche présentée ci-dessus donne l'algorithme générique suivant pour minimiser E_μ (pour une valeur de μ fixée).

Algorithme 2 Algorithme générique de minimisation du point proximal

1. $k = 0$ et $u^{(0)} = 0$
2. Calculer
$$u^{(k+1)} = p_\mu\left(u^{(k)}\right) = (M + \partial E_\mu)^{-1}\left(Mu^{(k)}\right)$$
3. Si la convergence n'est pas atteinte, $k \leftarrow k+1$ et aller à l'étape 2
Sinon retourner $u^{(k+1)}$

Plusieurs critères de convergence peuvent être retenus suivant l'application. Nous détaillons les choix effectués dans le cadre du CS dans la section 5.2.4.

5.2.2 Preuve de convergence

Nous présentons maintenant une preuve de convergence classique pour les itérations du point proximal. Rappelons que cette approche est standard et qu'une preuve peut être adaptée de [78,91] par exemple.

Tout d'abord, notons que la suite $\{E_\mu\left(p(u^{(k)})\right)\}$ est clairement décroissante et minorée. Ainsi, elle converge vers une valeur notée ici η.

Rappelons un résultat classique d'optimisation convexe. Supposons que $g : \mathbb{R}^N \to \mathbb{R}$ est convexe et différentiable et que $h : \mathbb{R}^N \to \mathbb{R}$ est convexe. Alors, $u^\star$ est un minimiseur global de $(g+h)$ si et seulement si :

$$\forall u \in \mathbb{R}^N \ \langle \nabla g(u^\star), u - u^\star \rangle + h(u) - h(u^\star) \geq 0 \tag{5.4}$$

Dans notre cas, $g(\cdot) = \frac{1}{2}\| \cdot - u^{(k)}\|_M^2$ et $h(\cdot) = E_\mu(\cdot)$, et rappelons que $p_\mu(u^{(k)})$ est le minimum global de l'inf-convolution quand on lui donne $u^{(k)}$:

$$\forall u \in \mathbb{R}^N \ \left\langle p\left(u^{(k)}\right) - u^{(k)}, u - p\left(u^{(k)}\right)\right\rangle_M + E_\mu(u) - E_\mu\left(p\left(\left(u^{(k)}\right)\right)\right) \geq 0 \tag{5.5}$$

Maintenant, considérons cette inégalité pour les deux $\hat{u}$ et $\bar{u}$ avec leurs points proximaux associés respectifs $p(\hat{u})$ et $p(\bar{u})$. Il s'ensuit que :

$$\langle p(\hat{u}) - \hat{u}, p(\bar{u}) - p(\hat{u})\rangle_M + \langle p(\bar{u}) - \bar{u}, p(\hat{u}) - p(\bar{u})\rangle_M \geq 0$$

et on obtient alors :

$$\langle \bar{u} - \hat{u}, p(\bar{u}) - p(\hat{u})\rangle_M \geq \|p(\hat{u}) - p(\bar{u})\|_M^2$$

Cette dernière inégalité est équivalente à :

$$\|\hat{u} - \bar{u}\|_M^2 - \|p(\hat{u}) - \hat{u} - p(\bar{u}) + \bar{u}\|_M^2 \geq \|p(\hat{u}) - p(\bar{u})\|_M^2$$

Maintenant, fixons $\bar{u}$ à un minimiseur global de R_μ, c'est-à-dire $\bar{u} = u^\star$ et $\hat{u} = u^{(k)}$ dans l'inégalité précédente. Notons que $p\left(u^\star\right) = u^\star$ et rappelons que $u^{(k+1)} = p\left(u^{(k)}\right)$. Il vient :

$$\left\|u^{(k)} - u^\star\right\|_M^2 - \left\|u^{(k+1)} - u^{(k)}\right\|_M^2 \geq \left\|u^{(k+1)} - u^\star\right\|_M^2 \tag{5.6}$$

Avec cette inégalité, on peut conclure en utilisant les arguments suivants.

La suite $\{\|u^{(k)} - u^\star\|_M^2\}$ est décroissante et minorée par $E_\mu(u^\star)$, donc elle converge. Ainsi, on déduit :

$$\lim_{k\to\infty} \|u^{(k+1)} - u^\star\|_M^2 - \|u^{(k)} - u^\star\|_M^2 = 0$$

En utilisant ce résultat et l'inégalité (5.6), on obtient $\lim_{k\to\infty} \|u^{(k+1)} - u^{(k)}\|_M^2 = 0$. Notons que du fait de la convexité de l'énergie E_μ, on a :

$$E_\mu\left(u^\star\right) \geq E\left(u^{(k+1)}\right) + \left\langle \partial E_\mu, u^\star - u^{(k+1)} \right\rangle \tag{5.7}$$

Comme $u^{(k+1)}$ est le minimiseur global de $F(u^{(k)})$, on a alors :

$$\left\langle \partial E_\mu, u^\star - u^{(k+1)} \right\rangle + \left\langle u^{(k+1)} - u^{(k)}, u^\star - u^{(k+1)} \right\rangle \geq 0 \tag{5.8}$$

Rappelons que, comme on l'a montré précédemment, $\lim_{k\to+\infty} \|u^{(k+1)} - u^{(k)}\|^2 = 0$. Comme $\|u^{(k+1)} - u^\star\|^2$ est borné, on a $\liminf_{k\to+\infty} \left\langle \partial E_\mu, u^\star - u^{(k+1)} \right\rangle \geq 0$. En injectant cette information dans l'inégalité (5.7) on obtient $\lim_{k\to+\infty} E_\mu\left(u^{(k)}\right) \leq \eta$ et ainsi on peut conclure que $\lim_{k\to+\infty} E_\mu(u^{(k)}) = \eta$.

5.2.3 Choix de l'opérateur proximal

Jusqu'ici, seule la version générique de l'approche a été décrite. En effet, comme on l'a vu précédemment, la régularisation de Moreau-Yosida avec n'importe quelle métrique M engendre un algorithme itératif qui converge vers une solution du problème. La versatilité de l'approche nous donne une certaine liberté dans le choix de la métrique M. Nous souhaitons utiliser une métrique qui donne une bonne vitesse de convergence, c'est-à-dire peu d'itérations [78,91]. Cependant, rappelons que notre but est de concevoir une approche facile à implémenter et efficace sur les architectures multi-coeurs. Ceci va nous guider dans le choix de M.

La partie la plus fastidieuse dans la résolution du problème (5.3) est l'inversion de l'opérateur $(M + \partial E_\mu)$ (pour $Mu^{(k)}$). La condition d'optimalité de la solution $u^{(k+1)}$ s'écrit :

$$s(u^{(k+1)}) + \mu A^t A(u^{(k+1)} - f) + M(u^{(k+1)} - u^{(k)}) = 0$$

où $s(u^{(k+1)})$ est un sous-gradient de $\|\cdot\|_1$ au point $u^{(k+1)}$. De manière plus concise et d'un point de vue d'optimisation, on a :

$$s(u^{(k+1)}) + (\mu A^t A + M)u^{(k+1)} = \mu A^t f + M u^{(k)}$$

Cette opération devient efficace quand elle est séparable, ce qui veut dire que l'optimisation peut être réalisée indépendamment, dimension par dimension. Pour notre problème, la propriété de séparabilité signifie que $(\mu A^t A + M)$ est une matrice diagonale. De plus, cette propriété engendre une implémentation qui satisfait les caractéristiques décrites dans la section 2.3. En effet, comme les variables sont découplées, paralléliser est simple. Les variables sont de plus facilement arrangeables dans un tableau correctement ordonné et aligné pour permettre de vectoriser

les calculs. Cette approche assure aussi une bonne utilisation du cache : les lectures sont séquentielles et permettent facilement de prédire l'emplacement des prochaines données nécessaires.

Pour obtenir un problème d'optimisation séparable, M doit annuler les éléments de $\mu A^t A$ qui ne sont pas sur la diagonale. De plus, pour assurer la convergence de l'approche, M soit être semi-definie positive. Rappelons que les bonnes matrices de CS correspondent aux sous-matrices satisfaisant la propriété d'isométrie restreinte, c'est-à-dire que toutes leurs valeurs propres appartiennent à $[1-\nu, 1+\nu]$ pour $\nu > 0$ et sont proches de 0. Ici, on suppose que $\nu = 0$. Ainsi, on a les valeurs propres de $A^t A$ égales à 0 où à 1 et comme $A^t A$ est toujours une matrice non-négative définie, on peut définir M comme suit :

$$M = (1+\epsilon)\mu Id - \mu A^t A$$

où ϵ est un nombre réel positif proche de 0 pour rendre M définie positive. Avec cette métrique M, on doit résoudre le problème suivant : trouver $u^{(k+1)}$ satisfaisant

$$s(u^{(k+1)}) + (1+\epsilon)\mu u^{(k+1)} = \mu A^t f + M u^{(k)}$$

Ce type de problèmes est bien connu pour se résoudre facilement par seuillage, comme on peut le voir dans [15,27,34,39,45,54,55,57,104]. Nous obtenons alors :

$$u^{(k+1)} = \frac{1}{(1+\epsilon)\mu} \begin{cases} \mu A^t f + M u^{(k)} - sign\left(\mu A^t f + M u^{(k)}\right) & \text{if } \left|\mu A^t f + M u^{(k)}\right| > 1 \\ 0 & \text{sinon} \end{cases} \tag{5.9}$$

où $sign(x) = \frac{x}{|x|}$ si $x \neq 0$ ou 0 sinon. Comme on peut le voir, la mise à jour de la solution requiert principalement des multiplications matrice-vecteur, ainsi que d'autres opérations qui peuvent s'implémenter facilement avec des instructions vectorielles. Nous pouvons aussi noter que l'on souhaite choisir un ϵ aussi petit que possible pour obtenir une convergence plus rapide. Fixer empiriquement ϵ à 0 fonctionne, même si la preuve présentée dans la section précédente ne s'applique pas dans ce cas.

5.2.4 Algorithme proposé

Dans cette section, nous décrivons l'algorithme complet. Jusqu'ici, nous avons considéré l'optimisation de E_μ quand μ est fixé à une certaine valeur positive arbitraire. Rappelons que nous voulons fixer μ à une valeur élevée pour forcer la contrainte $Au = f$ au niveau de l'énergie. Cependant, quand μ est élevé, la procédure génère une suite de signaux $u^{(k+1)}$ qui converge lentement vers la solution. Une manière classique d'accélérer le processus est d'utiliser une approche qui consiste à résoudre le problème (approximativement ou non) pour différentes valeurs croissantes de μ. Ce genre d'approches s'est montré fructueux, par exemple dans [54,101–103].

Nous choisissons le signal nul comme solution de départ. Il se trouve que si μ est trop faible, E_μ ne diminue pas. Il faut donc choisir un μ suffisamment grand pour que E_μ diminue. De plus, rappelons que, pour des raisons de performance,

on souhaite un μ le plus petit possible. Un tel μ initial peut être obtenu par une approche dichotomique. Nous supposons que l'on a une borne inférieure sur μ telle que $\mu > \mu_{min} > 0$. La procédure suivante calcule le plus petit μ, jusqu'à une précision e_{dicho}, qui engendre une décroissance de l'énergie :

Algorithme 3 Recherche dichotomique pour le μ initial

Entrée : f, A, μ_{min}

1. $\mu_{max} = 0$
2. Tant que $\mu_{max} = 0$
   a) Calculer $\hat{u}$ en utilisant la formule (5.9) pour μ_{min} avec le signal nul comme solution précédente
   b) Si $E_{\mu_{min}}(\hat{u}) < E_{\mu_{min}}(0)$ alors $\mu_{max} = \mu_{min}$ et $\mu_{min} = \frac{\mu_{min}}{2}$
   c) Sinon $\mu_{min} \leftarrow 2\mu_{min}$
3. Tant que $|\mu_{max} - \mu_{min}| > e_{dicho}$
   a) Calculer $\hat{u}$ en utilisant la formule (5.9) pour $\frac{\mu_{min} + \mu_{max}}{2}$ avec le signal nul comme solution précédente
   b) Si $E_{\mu_{min}}(\hat{u}) < E_{\mu_{min}}(0)$ alors $\mu_{max} = \frac{\mu_{min} + \mu_{max}}{2}$
   c) Sinon $\mu_{min} = \frac{\mu_{min} + \mu_{max}}{2}$
4. Retourner μ_{max}

Cette procédure recherche d'abord μ_{max}, une borne supérieure pour μ, durant les itérations des étapes $2-(a)$ à $2-(c)$. Ensuite, une recherche dichotomique classique est réalisée au cours des itérations des étapes $3-(a)$ à $3-(c)$.

Une fois qu'un μ initial est trouvé, on embarque les itérations du point proximal dans une procédure de mise à jour dyadique de μ : μ va successivement prendre les valeurs $\mu, 2\mu, \ldots, 2^{l_{max}}\mu$, où $l_{max} \in \mathbb{N}_*^+$. Nous pouvons noter qu'un facteur différent de 2 aurait pu être utilisé, mais les expériences montrent que cette valeur permet d'obtenir de bons résultats.

Intéressons-nous maintenant aux critères d'arrêt de notre approche. Ces critères sont utilisés lors de l'optimisation de E_μ avec μ fixé. Le premier critère concerne la proximité de la solution reconstruite par rapport aux données observées. Le processus d'optimisation est en effet arrêté quand la solution reconstruite $u^{(k+1)}$ est en dessous d'une certaine tolérance e_{tol} spécifiée, qui est calculée de la sorte :

$$\frac{\|Au^{(k+1)} - f\|_2}{\|f\|_2}$$

Comme les itérations du point proximal convergent vers une solution mais pas nécessairement en un nombre fini d'itérations, on peut aussi arrêter le processus

quand la diminution d'énergie entre deux solutions consécutives est plus faible qu'une certaine valeur e_{consec} fixée. Cela correspond au critère d'arrêt suivant :

$$E_\mu(u^{(k)}) - E_\mu(u^{(k+1)}) < e_{consec}$$

Quand l'un de ces deux critères d'arrêt est satisfait, la valeur μ est mise à jour et une nouvelle série de minimisations pour E_μ commence en utilisant la solution actuelle comme initialisation. Il en résulte l'algorithme de minimisation suivant :

Algorithme 4 Algorithme de minimisation

Entrée : f, A, μ_{min}

1. $k = 0$ et $u^{(0)} = 0$
2. Calculer la valeur initiale de μ en utilisant l'approche dichotomique
3. Pour $l = 1$ jusqu'à l_{max} (nombre de mises à jour de μ)
   a) Faire
      i. $k \leftarrow k + 1$
      ii. Calculer $u^{(k+1)}$ en utilisant la formule (5.9)
   b) Tant que $E_\mu(u^{(k)}) - E_\mu(u^{(k+1)}) > e_{consec}$ et $\dfrac{\|Au^{(k+1)} - f\|_2}{\|f\|_2} > e_{tol}$
   c) $\mu \leftarrow 2\mu$
4. Retourner $u^{(k+1)}$

L'algorithme basé sur les itérations du point proximal qui vient d'être décrit peut être utilisé pour obtenir une solution approchée, mais il peut aussi servir à trouver une base pour la solution exacte. En effet, en utilisant le support de la solution trouvée, on peut effectuer la fin d'une itération de simplexe pour obtenir une solution exacte.

En particulier, après que la recherche dichotomique de μ soit finie et avant chaque mise à jour de μ, on peut vérifier si $\|u^{(k+1)}\|_0 \leq m$. Si c'est le cas, comme le support de $u^{(k+1)}$ forme une base, une solution exacte peut être calculée par l'algorithme suivant :

Algorithme 5 Solution exacte à partir de solution approchée

Entrée : f, A, $u^{(k+1)}$

1. Calculer la matrice de base B associée à la solution $u^{(k+1)}$ en choisissant les colonnes de A qui correspondent aux composantes non-nulles de $u^{(k+1)}$
2. Calculer $v = B^{-1} \times f$
3. Retourner $u^{(k+2)}$ dans la base A à partir de v dans la base B en ajoutant les 0 nécessaires

A la fin de l'algorithme, une solution $u^{(k+2)}$ est fournie. Néanmoins, même si le seuillage garantit que les composantes nulles de la solution courante ne changeront plus de valeur, la modification de la valeur de μ peut entraîner un tel changement. Ainsi, l'itération qui fait suite à une mise à jour de μ peut permettre à une composante nulle de devenir non-nulle (auquel cas la solution change de base). Cette situation implique que, même si $\|u^{(k+1)}\|_0 \leq m$, la base associée peut ne pas être une base pour la solution du problème original (4.1), comme le μ initial est très serré. Néanmoins, cette situation peut facilement être vérifiée en utilisant le test suivant :

$$\text{Si } \frac{\|Au^{(k+2)} - f\|_2}{\|f\|_2} > e_{tol} \text{ alors } v \text{ ne permet pas d'obtenir une solution exacte.}$$

Dans ce cas, μ est mis à jour et de nouvelles itérations du point proximal sont nécessaires pour que ce critère soit satisfait et donc qu'une solution exacte puisse être retournée. Il est à noter que cette variante exacte n'est efficace que dans le cadre de matrices stockées explicitement (par opposition aux transformées rapides). En effet, l'étape d'inversion de matrice de base B nécessite au préalable de sélectionner jusqu'à m colonnes de la matrice A. Dans le cadre de transformées rapides, cela voudrait dire appliquer une transformée par colonne à sélectionner (en utilisant en entrée le vecteur avec une unique composante non-nulle à 1 pour sélectionner la colonne voulue). Même sans cela, l'étape d'inversion de la matrice de base sera clairement très pénalisante, étant donné les performances très élevées issues des transformées rapides et les tailles qui en découlent.

5.3 Implémentations et expériences

Dans cette section, nous présentons les choix effectués au niveau de l'implémentation, et en particulier ceux liés aux architectures ciblées. Ensuite, des expériences sont réalisées pour évaluer l'efficacité de la méthode proposée sur les architectures parallèles considérées.

5.3.1 Implémentations

Les opérations les plus consommatrices en temps de cette méthode sont celles qui impliquent la matrice d'échantillonnage A. Les matrices A peuvent soit être stockées

de manière explicite, soit être représentées par des transformées rapides, comme il est fait état dans la section 4.4.1 du chapitre précédent (le lecteur peut s'y référer pour plus de détails, notamment concernant l'utilisation des transformées). Toutes les opérations effectuées dans le cadre de notre méthode respectent les caractéristiques de la section 2.3.

Comme la méthode des itérations du point proximal est robuste, tous les calculs sont ici effectués en nombres flottants simple précision (32 bits), ce qui a l'avantage d'engendrer de meilleures performances, en particulier sur GPU et Cell. La seule exception se situe dans le cas de la variante exacte de la méthode, où l'étape d'inversion de la matrice de base est très sensible à la précision utilisée. Ainsi, cette dernière utilise des nombres flottants double précision (64 bits) pour éviter toute instabilité numérique.

Comme le stockage des matrices requiert une certaine quantité de mémoire, seules les plateformes CPU (et GPU dans une moindre mesure) peuvent exécuter le code qui utilise des matrices Gaussiennes orthogonalisées. Au contraire, les DCT partielles peuvent être utilisées sur toutes les architectures considérées ici (CPU, GPU et Cell). Pour permettre une comparaison juste, seule cette dernière a été implémentée sur les trois plateformes. De plus, les performances en flottants double précision étant très dégradées par rapport à celles en flottants simple précision sur GPU et Cell, seule la version CPU dispose d'une implémentation de la variante exacte.

Nous présentons maintenant les détails d'implémentation spécifiques aux différentes architectures.

CPU

Le code fonctionnant sur CPU est parallélisé en utilisant l'API OpenMP [41]. Notre implémentation est vectorisée et utilise les instructions SSE. De plus, elle tire parti du cache. Rappelons que l'approche par seuillage de la section 5.2.4 a été choisie pour sa propriété de séparabilité (c'est-à-dire que chaque élément peut être traité indépendamment), de telle sorte que sa parallélisation et sa vectorisation permettent d'obtenir une implémentation efficace. Les calculs de FFT sont réalisés avec la bibliothèque FFTW 3.1 [62]. Le code est compilé avec ICC 10.1.

Cell

Notre implémentation pour architecture Cell utilise le SDK Cell 3.0[1] fourni par IBM et la bibliothèque FFTW 3.2 alpha 3. Pour le Cell, il s'agit de la bibliothèque la plus efficace pour les tailles de FFT qui nous intéressent (par exemple, FFTC [6] est plus rapide mais ne peut gérer des entrées complexes plus grandes que 16K). Cette version alpha de FFTW requiert un accès exclusif aux SPE utilisés (c'est-à-dire que les SPE ne peuvent exécuter un autre code si on leur fait calculer des FFT). Comme les calculs de FFT sont les plus consommateurs en temps pour notre méthode, tous les SPE sont affectés aux calculs de FFT. La conversion FFT/DCT, le seuillage et les calculs d'énergie sont donc réalisés par le PPE (qui, rappelons-le, n'a pas été conçu pour les performances). Ces opérations sur PPE utilisent les instructions AltiVec [48]

[1] https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine

pour obtenir des performances plus élevées. Néanmoins, notre implémentation sur Cell pourrait être plus optimisée, comme un certain nombre d'opérations pourraient être réalisées sur SPE, plutôt que sur PPE.

GPU

L'implémentation GPU utilise CUDA 2.0[2] et la bibliothèque CUFFT (qui fait partie des bibliothèques fournies en standard avec CUDA). Le bus PCI Express qui relie le CPU au GPU souffre de limitations fortes au niveau de sa bande passante. Notre implémentation prend donc en compte cette caractéristique et limite au maximum les interactions entre CPU et GPU, ainsi que les transferts entre mémoire principale (RAM) et mémoire embarquée sur la carte graphique (VRAM). Cette limitation est l'une des limitations principales liées au GPGPU. De plus, dans les situations nécessitant des transferts, il faut faire particulièrement attention à la taille des données transférées [94]. Les GPU actuels disposent d'unités de calcul, de caches et de mémoires hautement hiérarchiques. Le nombre de *threads* utilisés doit être maximisé dans le but d'obtenir de meilleures performances (pour qu'aucune unité de calculs ne soit en attente), tout en évitant les conflits de bancs mémoires (c'est-à-dire, plusieurs *threads* accédant à un même banc mémoire en même temps).

5.3.2 Expériences

Dans cette section, nous décrivons tout d'abord les conditions expérimentales puis nous présentons les résultats numériques pour montrer l'efficacité de notre approche.

Conditions expérimentales

Pour montrer l'efficacité de notre approche et comparer les avantages et les inconvénients liés aux différentes architectures, on utilise différentes plateformes :

- Deux processeurs Intel disposant de 4 coeurs. Le premier est un Core 2 Quad Q6600 2,4GHz avec 2×4Mo de cache L2 et 4Go de RAM. Théoriquement, ce processeur peut atteindre 38 GFLOPS en double précision et 76 GFLOPS en simple précision. Le second processeur est un Core i7 920 2,66GHz avec 4×256Ko de cache L2, 8Mo de cache L3 partagé et 6Go de RAM. Ce processeur peut atteindre 43 GFLOPS en double précision et 86 GFLOPS en simple précision. Pour ce processeur, le *Turbo Boost* a été désactivé dans le but d'éviter tout biais résultant d'une augmentation de la fréquence au-delà de sa fréquence nominale.

- Un processeur Cell de Sony Playstation 3. Cette plateforme possède un processeur Cell à 7 SPE qui fonctionne à 3,2Ghz et 256Mo de RAM. Cependant, seulement 6 des 7 SPE et 192Mo de RAM sont utilisables (le dernier SPE et la RAM restante étant réservés au système).

[2] http://developer.nvidia.com/category/zone/cuda-zone

- Deux cartes graphiques produites par NVIDIA : une GEFORCE 8800 GTS et une NVIDIA GEFORCE GTX 275. La première dispose de 96 coeurs à 1,2GHz et embarque 640Mo de VRAM alors que la seconde a 240 coeurs à 1,4GHz et 896Mo de VRAM.

Pour plus de détails sur les architectures et les caractéristiques de ces plateformes, le lecteur peut se référer à la section 2.1.

Les matrices A utilisées sont des matrices Gaussiennes orthogonalisées et des DCT partielles. Pour plus de détails sur ces matrices, se référer à la section 4.4.2.

Nous présentons maintenant les valeurs utilisées pour les différents paramètres de nos expériences. Le ratio $\frac{m}{n}$ est fixé à $\frac{1}{8}$ (rappelons que m est supposé très faible par rapport à n). Le nombre de composantes non-nulles dans le signal creux original est fixé à $k = \frac{m}{10}$. Rappelons que nous utilisons deux critères d'arrêt : la tolérance e_{tol} avec son critère associé $\frac{\|Au^{(k+1)}-f\|_2}{\|f\|_2} < e_{tol}$ et la variation consécutive d'énergie e_{consec} avec son critère $E(u^{(k)}) - E(u^{(k+1)}) < e_{consec}$. Nous fixons $e_{tol} = 10^{-5}$ et $e_{consec} = 10^{-3}$. Pour la recherche dichotomique, on fixe $\mu_{min} = 2$ et $e_{dicho} = 0,5$. Le nombre de mises à jour de μ effectuées après la recherche dichotomique, l_{max}, est fixé à $l_{max} = 10$.

Une expérience préliminaire est maintenant présentée. La figure 5.1 représente la variation de l'erreur en fonction du temps. Deux critères sont ici représentés : la tolérance e_{tol} et la *ground truth* (ou erreur relative à la vérité terrain, issue du signal optimal $u^{(k+1)}$) $relative = \frac{\|u^{(k+1)}-u^*\|_2}{\|u^*\|_2}$. Nous avons choisi un exemple représentatif du comportement général de notre méthode de résolution. En l'occurence, cet exemple utilise une matrice Gaussienne orthogonalisée de taille 2048×16384.

Nous observons en premier lieu l'étape de recherche dichotomique au tout début du processus. Quand un μ initial est trouvé, le processus de mise à jour de μ est lancé. Au niveau de la courbe, chaque mise à jour correspond à une décroissance forte de celle-ci dans les deux critères d'erreurs. Nous pouvons aussi observer que l'erreur liée à la *ground truth* décroît quasi linéairement jusqu'à 400 itérations, puis cette décroissance devient beaucoup plus lente. Au contraire, l'erreur relative ne montre pas cette étape de décroissance rapide. Ce graphique montre que notre méthode pourrait être arrêtée bien plus tôt en utilisant différents paramètres, tout en obtenant une solution de qualité très peu détériorée. Cette caractéristique est très commune de ce genre d'approche d'optimisation, telles que celles présentées dans la section 5.1. L'inconvénient qui en résulte est que la comparaison entre les différentes méthodes en devient beaucoup plus compliquée. Ceci est d'autant plus vrai que les méthodes reposent souvent sur des jeux de paramètres très différents et que ceux-ci n'ont pas forcément d'équivalents dans d'autres méthodes. En particulier, notre méthode a l'avantage de ne pas reposer sur un paramètre dt représentant le pas d'optimisation, contrairement aux méthodes qui se basent sur une linéarisation.

Après cette expérience préliminaire, passons aux expériences plus complètes. Les résultats suivants sont présentés sous forme de tableaux et montrent les valeurs d'erreur de tolérance e_{tol} et d'erreur relative à la *ground truth* $relative$, à la fin de l'optimisation. De plus, les résultats liés aux performances sont aussi présentés : le nombre d'itérations #itér et le temps de calcul total en secondes.

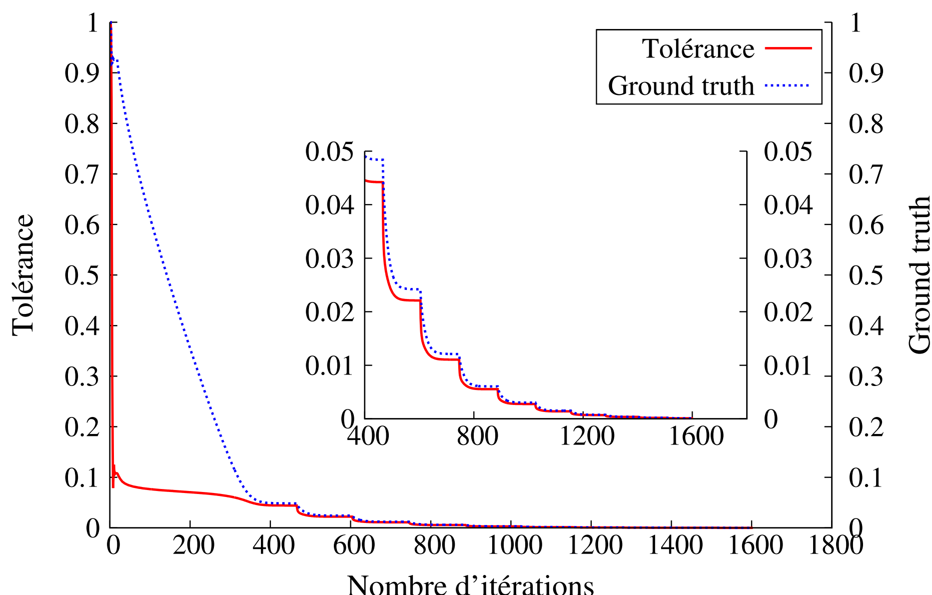

Figure 5.1: Erreurs en fonction du temps pour un exemple représentatif du comportement général de notre méthode utilisant une matrice Gaussienne orthogonalisée de taille 2048×16384.

Matrices Gaussiennes orthogonalisées

Ce jeu d'expériences utilise des matrices Gaussiennes orthogonalisées et notre implémentation CPU avec différents nombres de *threads* sur les deux processeurs Intel envisagés. Le tableau 5.1 présente les résultats obtenus avec un processeur Intel Core 2 Quad Q6600, alors que le tableau 5.2 présente ceux obtenus avec un processeur Intel Core i7 920.

					temps (s)		
m	n	tolérance	relative	#itér	1 *thread*	2 *threads*	4 *threads*
64	512	1,24e-03	1,40e-03	1163,2	0,076	0,044	0,029
128	1024	4,81e-04	5,22e-04	1155,2	0,238	0,128	0,072
256	2048	2,99e-04	3,26e-04	1326,4	1,144	0,562	0,293
512	4096	1,93e-04	2,15e-04	1461,7	5,659	3,584	3,386
1024	8192	1,29e-04	1,43e-04	1505,0	23,224	15,500	15,352
2048	16384	8,66e-05	9,62e-05	1611,1	97,123	64,527	64,778

Tableau 5.1: Résultats avec des matrices Gaussiennes orthogonalisées sur un processeur Intel Core 2 Quad Q6600.

					temps (s)			
m	n	tolérance	relative	#itér	1 *thread*	2 *threads*	4 *threads*	8 *threads*
64	512	1,24e-03	1,39e-03	1162,0	0,042	0,025	0,017	0,252
128	1024	4,81e-04	5,22e-04	1155,5	0,115	0,068	0,046	0,350
256	2048	2,99e-04	3,26e-04	1321,5	0,423	0,227	0,132	0,375
512	4096	1,93e-04	2,15e-04	1465,8	2,362	1,495	1,130	1,359
1024	8192	1,29e-04	1,43e-04	1505,6	9,933	6,035	4,574	4,885
2048	16384	8,66e-05	9,62e-05	1604,9	41,842	25,284	19,576	19,997

Tableau 5.2: Résultats avec des matrices Gaussiennes orthogonalisées sur un processeur Intel Core i7 920.

Le Core 2 Quad passe correctement à l'échelle jusqu'à $n = 4096$. A partir de cette valeur, il n'y a plus de bénéfice à utiliser 4 *threads* au lieu de 2. Ceci est dû à l'interconnexion entre les 2 groupes de 2 coeurs de ce processeur qui passe par le FSB et à sa hiérachie de cache, qui ont de grandes répercussions sur la bande passante. Un tel comportement a déjà été remarqué par exemple par [120]. Nous pouvons noter que ceci n'influe que très peu sur ce qui arrive quand on passe d'1 à 2 coeurs comme les 2 *threads* sont exécutés sur 2 coeurs du même *cluster*.
Pour la même expérience sur le Core i7, rappelons que grâce à l'*Hyper-Threading* (HT) ce processeur permet de gérer jusqu'à 8 *threads* de manière matérielle bien que celui-ci ne dispose que de 4 coeurs physiques. Pour notre expérience, l'utilisation de l'HT implique une dégradation des performances quand on passe de 4 à 8 *threads*, ceci étant dû à une trop forte pression sur le cache et les unités de calcul. Le phénomène qui apparaît à $n = 4096$ avec le Core 2 Quad n'apparaît pas avec le Core i7 grâce à son interconnexion entre coeurs et sa hiérarchie de cache plus efficaces. Au-delà de $n = 4096$ le passage à l'échelle n'est plus aussi bon qu'avant, mais il reste constant et permet tout de même d'obtenir de bons gains en performance.

La figure 5.2 compare les performances de nos résolutions approchées et exactes, toujours avec les mêmes matrices Gaussiennnes orthogonalisées. Seul le Core i7 avec 1 et 4 *threads* est ici utilisé.

Nous voyons que la résolution exacte est significativement plus rapide que la résolution approchée. En effet, pour toutes les tailles de matrices et 1 *thread*, la résolution exacte est toujours plus rapide que la résolution approchée. Nous observons que plus n augmente, plus la proportion de temps pris par l'inversion de matrice augmente. Pour $n = 2048$ et plus, la résolution approchée avec 4 *threads* est même plus lente que la résolution exacte avec 1 *thread*. Nous pouvons finalement ajouter qu'utiliser 4 *threads* à la place d'1 donne toujours de meilleurs résultats pour les deux méthodes et donc que la parallélisation est ici toujours profitable.

Le tableau 5.3 présente des résultats plus complets et plus précis concernant la résolution exacte, toujours avec des matrices Gaussiennes orthogonalisées et sur le Core i7, avec différents nombres de *threads*. #inv correspond au nombre d'inversions effectuées par instance et t_{inv} (resp. t_{total}) est le temps pris par l'inversion de la matrice de base (resp. le temps total) en secondes. Les valeurs prises par nos deux critères d'arrêt sont aussi présentées pour montrer que la solution finale a une erreur

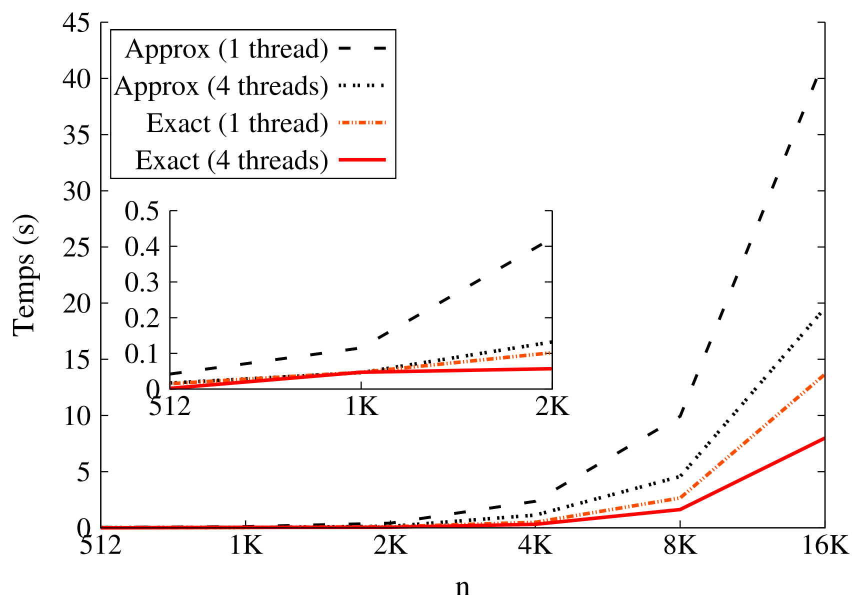

Figure 5.2: Comparaison entre les temps d'exécution de notre méthode approchée et de sa variante exacte avec des matrices Gaussiennes orthogonalisées et un processeur Intel Core i7 920.

						1 thread		*2 threads*		*4 threads*		*8 threads*	
m	n	tolérance	relative	#itér	#inv	t_{inv}	t_{total}	t_{inv}	t_{total}	t_{inv}	t_{total}	t_{inv}	t_{total}
64	512	1,83e-07	1,81e-07	88,9	1,4	0,012	0,015	0,007	0,009	0,000	0,002	0,000	0,180
128	1024	4,06e-07	4,16e-07	149,3	1,0	0,032	0,047	0,018	0,026	0,022	0,047	0,011	0,179
256	2048	6,00e-07	5,91e-07	213,6	1,1	0,027	0,102	0,028	0,075	0,014	0,057	0,007	0,150
512	4096	6,13e-07	6,48e-07	284,0	1,0	0,023	0,493	0,027	0,339	0,034	0,315	0,029	0,386
1024	8192	1,50e-06	1,59e-06	358,5	1,0	0,192	2,658	0,157	1,637	0,175	1,629	0,221	2,267
2048	16384	2,17e-06	2,30e-06	464,5	1,0	1,092	13,679	1,135	8,664	1,023	7,998	1,056	11,365
4096	32768	3,75e-06	3,97e-06	636,6	1,2	8,458	79,598	10,580	53,172	7,500	43,910	6,671	67,691

Tableau 5.3: Résultats pour la variante **exacte** de notre méthode de résolution et une implémentation hybride simple/double précision pour des matrices Gaussiennes orthogonalisées sur un processeur Intel Core i7 920. #inv correspond au nombre moyen d'inversions effectuées par instance. t_{inv} (resp. t_{total}) est le temps pris par l'inversion de la matrice de base (resp. le temps total) en secondes.

qui correspond à ce que l'on peut attendre de nombres flottants simple précision. En effet, la matrice de contraintes est stockée en flottants simple précision donc, même si l'inversion est réalisée avec des flottants double précision, la solution obtenue peut difficilement dépasser la précision des flottants simple précision. Si on voulait une précision plus élevée, il faudrait passer en entrée de l'algorithme une matrice

de contraintes utilisant des flottants double précision. Dans la plupart des cas, le paramètre μ initial est suffisamment bon pour que la base de la solution optimale soit trouvée sans avoir à le mettre à jour. En effet, le nombre d'inversions nécessaires pour obtenir la solution exacte est très proche de 1 (au lieu des 10 fixées pour la résolution approchée). Ainsi, le nombre d'itérations total est drastiquement diminué. Pour les instances les plus petites, nous observons que le temps requis pour les inversions est significatif et correspond quasiment à l'intégralité du temps d'exécution total. Pour les instances plus grandes, le temps d'inversion correspond à peu près à $\frac{1}{10}^{\text{ème}}$ du temps total. Pour les mêmes raisons que précédemment, utiliser 8 *threads* à la place de 4 résulte en une perte de performance. Pour $n = 2048$ et plus, dans tous les cas de 1 à 4 *threads*, utiliser plus de *threads* permet de diminuer le temps d'exécution total, même si l'étape d'inversion peut, elle, être plus coûteuse dans certains cas. Par exemple, pour $n = 32768$, la résolution avec 2 *threads* est 25% plus lente qu'avec 1 seul, mais utiliser 3 ou 4 *threads* permet d'obtenir de meilleurs résultats qu'avec 1 seul *thread*.

Au final, à partir de la figure 5.2 et du tableau 5.3, nous voyons que, grâce à la diminution du nombre d'itérations et malgré les étapes d'inversion de matrices supplémentaires, la variante exacte de notre méthode de résolution est significativement plus rapide que sa contrepartie approchée. Rappelons cependant que cette résolution exacte n'est efficace que dans le cas de matrices stockées explicitement. Ainsi, toutes les expériences suivantes portent sur la résolution approchée du problème du CS.

DCT partielles

Ce nouveau jeu d'expériences utilise des DCT partielles (donc des transformées rapides à la place des matrices stockées explicitement) et l'implémentation CPU avec différents nombres de *threads* sur les deux processeurs Intel envisagés. Le tableau 5.4 montre les résultats obtenus avec un processeur Intel Core 2 Quad Q6600 alors que le tableau 5.5 montre ceux obtenus avec un processeur Intel Core i7 920. Comme des DCT et des DCT inverses sont effectuées à la place des multiplications matrice-vecteur, une grande quantité de mémoire est économisée et les opérations sont beaucoup plus rapides. En particulier, cela signifie que nos expériences peuvent être réalisées sur des tailles de signaux n beaucoup plus élevées.

Nous pouvons voir que les performances passent très mal à l'échelle pour le Core 2 Quad quand n est petit. Le bénéfice d'utiliser plus de coeurs n'apparaît que quand il y a suffisamment de calculs à effectuer sur chaque coeur. Pour $n = 2048$, le meilleur temps d'exécution est observé pour 2 *threads*. A l'intérieur d'un *cluster* de 2 coeurs, le cache de niveau 2 partagé permet de passer à l'échelle pour cette taille de problème. Cependant, l'interconnexion entre coeurs et la hiérarchie de cache pénalise les performances au-delà de 2 coeurs. Dans notre cas, le bénéfice d'utiliser 4 coeurs n'apparaît que quand n atteint 4096. Ensuite, un n plus élevé implique un meilleur passage à l'échelle.

Pour le Core i7, il n'y a aucune pénalité à utiliser 2 *threads* au lieu d'1, ni 4 au lieu de 2, et ce pour toutes les tailles d'instances envisagées. Plus l'instance est grande, meilleur est le passage à l'échelle. En ce qui concerne l'HT, comme pour les matrices

m	n	tolérance	relative	#itér	temps (s) 1 *thread*	2 *threads*	4 *threads*
64	512	1,02e-03	1,11e-03	1029,9	0,034	0,044	0,050
128	1024	4,74e-04	5,08e-04	1075,1	0,072	0,081	0,089
256	2048	2,81e-04	3,09e-04	1219,6	0,180	0,169	0,185
512	4096	1,92e-04	2,15e-04	1415,6	0,501	0,472	0,405
1024	8192	1,27e-04	1,40e-04	1456,6	1,110	0,884	0,818
2048	16384	8,68e-05	9,61e-05	1584,7	2,544	1,840	1,797
4096	32768	6,20e-05	6,91e-05	1780,5	6,366	4,647	4,424
8192	65536	4,33e-05	4,82e-05	2016,0	16,571	11,543	10,797
16384	131072	3,01e-05	3,35e-05	2256,5	46,512	32,627	27,568
32768	262144	2,15e-05	2,40e-05	2672,1	199,254	117,574	94,599
65536	524288	1,52e-05	1,70e-05	3261,0	508,388	298,467	204,657
131072	1048576	1,11e-05	1,25e-05	4067,5	1321,440	825,617	567,315

Tableau 5.4: Résultats pour les DCT partielles sur un processeur Intel Core 2 Quad Q6600.

m	n	tolérance	relative	#itér	temps (s) 1 *thread*	2 *threads*	4 *threads*	8 *threads*
64	512	1,02e-03	1,11e-03	1031,3	0,025	0,023	0,020	0,055
128	1024	4,74e-04	5,08e-04	1076,8	0,055	0,038	0,034	0,068
256	2048	2,81e-04	3,09e-04	1223,2	0,120	0,082	0,070	0,126
512	4096	1,92e-04	2,15e-04	1419,6	0,288	0,197	0,157	0,220
1024	8192	1,25e-04	1,38e-04	1438,5	0,598	0,413	0,316	0,391
2048	16384	8,76e-05	9,70e-05	1597,2	1,452	0,954	0,718	0,911
4096	32768	6,21e-05	6,91e-05	1783,2	3,494	2,112	1,641	1,887
8192	65536	4,33e-05	4,81e-05	2004,8	8,293	5,280	3,780	4,137
16384	131072	3,00e-05	3,35e-05	2253,2	20,026	11,961	8,755	11,017
32768	262144	2,13e-05	2,38e-05	2625,0	87,031	49,508	32,411	34,852
65536	524288	1,52e-05	1,70e-05	3262,0	225,341	129,467	82,432	91,006
131072	1048576	1,12e-05	1,26e-05	4074,6	628,962	337,811	222,477	239,236

Tableau 5.5: Résultats pour les DCT partielles sur un processeur Intel Core i7 920.

Gaussiennes orthogonalisées, nous observons une perte de performance en passant de 4 à 8 *threads*.

Ouvrons une parenthèse pour justifier, à partir des résultats expérimentaux obtenus, le fait que la variante exacte n'est pas efficace dans le cadre des transformées rapides. A partir des résultats du tableau 5.5 et notamment du nombre d'itérations requises, nous voyons que, pour $n = 1048576$, il faut en moyenne $4074,6$ itérations à la version approchée. Comme μ est mis à jour 10 fois dans la version approchée et que la variante exacte ne requiert très souvent aucune mise à jour (voir tableau 5.3), on peut estimer grossièrement que cette dernière demanderait alors $407,46$ itérations (en pratique, entre la version approchée et la version exacte il y a plutôt entre $2,5$ et $3,5$ fois moins d'itérations, mais comme cela dépend de la taille considérée, gardons cette estimation optimiste). Chaque itération demande 2 applications de transformées rapides, ce qui fait $814,92$ transformées rapides avant l'étape d'inversion. Rappelons que les solutions pour nos expériences ont un nombre de composantes non-nulles

égal à $k = \frac{m}{10}$. Ainsi, il faudrait 13107 transformées rapides pour l'étape de la création de matrices de base (toujours dans l'hypothèse où μ n'a pas à être mis à jour et donc que l'on a directement la bonne base, sinon ce nombre pourrait aller jusqu'à atteindre 131072 transformées rapides). Au total, cela donnerait alors $13921,92$ transformées rapides, plus l'étape d'inversion de matrice de base (dont la proportion de temps par rapport au temps total augmente quand n augmente), à comparer aux $8149,2$ transformées rapides de la version approchée. En ce qui concerne la matrice de base, elle aurait une taille 131072×131072, c'est-à-dire 128Go en flottants double précision. Pour $n = 131072$, la matrice de base prendrait plus raisonnablement 2Go et pourrait ainsi être stockée en RAM, mais même pour cette taille le temps nécessaire à son inversion serait considérable. A en juger par les valeurs pour $n = 32768$, $3,5$ secondes de résolution en approché donnerait $0,35$ secondes en exact pour la partie itérative (toujours avec notre estimation optimiste du nombre d'itérations), à comparer aux $8,5$ secondes requises pour l'inversion de la matrice de base correspondante dans l'expérience avec les matrices Gaussiennes orthogonalisées (tableau 5.3). Etant données les performances issues de l'utilisation de transformées rapides à la place de matrices explicites et les nouvelles tailles considérées, l'inversion de matrice est clairement l'étape la plus consommatrice en temps, et justifie à elle seule le fait que la variante exacte ne soit pas adaptée à l'utilisation de transformées rapides. De plus, rappelons que nous nous sommes ici placés dans le cas optimiste (mais néanmoins assez réaliste) où μ n'a pas à être mis à jour, c'est-à-dire qu'une seule matrice de base est générée puis inversée, et que, dans le cas contraire la création de la matrice de base pourrait être jusqu'à 10 fois plus lente (à cause de la valeur prise par k). De plus, le nombre d'itérations requises a aussi été estimé de manière optimiste et les valeurs choisies dans le cadre de nos expériences pour le ratio $\frac{m}{n}$ et k ne désavantagent clairement pas la résolution exacte avec des transformées rapides.

Les expériences suivantes (tableaux 5.6, 5.7 et 5.8) concernent les implémentations sur Cell et GPU. Les résultats présentés ici reposent sur l'utilisation de DCT partielles (donc de transformées rapides) étant données les limitations sur la mémoire disponible pour ces deux plateformes. Rappelons que ces implémentations ne sont pas aussi avancées que l'implémentation CPU et qu'elles montrent surtout les performances que l'on peut attendre à partir d'un effort minime de développement (et qui est surtout dû aux contraires logicielles et de développement de ces plateformes). Néanmoins, ces implémentations prennent bien évidemment en compte les caractéristiques décrites dans la section 2.3 et tirent avantage de la conception de l'algorithme.

Contentons-nous ici de remarquer que les erreurs sont particulièrement proches sur les différentes plateformes. Les GPU sont actuellement limités par la dimension des données sur lesquelles ils peuvent travailler, c'est pourquoi ils ne permettent pas facilement de travailler au-delà de $n = 131072$. Les performances liées à ces plateformes et à nos implémentations sont discutées ci-après.

La figure 5.3 montre une comparaison entre les trois implémentations sur les cinq plateformes matérielles envisagées avec des DCT partielles (FFT). Notre implémentation basique pour GPU est aussi rapide que notre implémentation optimisée pour CPU. Plus précisément, le GPU de la NVIDIA 8800 GTS fait face au Core 2 Quad Q6600

m	n	tolérance	relative	#itér	temps (s)
64	512	1,02e-03	1,12e-03	1033,3	0,071
128	1024	4,74e-04	5,08e-04	1077,7	0,153
256	2048	2,81e-04	3,09e-04	1231,1	0,359
512	4096	1,92e-04	2,15e-04	1420,6	0,913
1024	8192	1,27e-04	1,39e-04	1458,8	1,935
2048	16384	8,67e-05	9,61e-05	1581,1	4,340
4096	32768	6,21e-05	6,91e-05	1772,4	10,445
8192	65536	4,33e-05	4,82e-05	2006,8	29,523
16384	131072	3,00e-05	3,34e-05	2232,5	60,178
32768	262144	2,14e-05	2,38e-05	2663,0	149,679
65536	524288	1,51e-05	1,68e-05	3236,5	367,822
131072	1048576	1,09e-05	1,23e-05	4066,7	934,656

Tableau 5.6: Résultats pour les DCT partielles sur le Cell de la Playstation PS3 de Sony.

m	n	tolérance	relative	#itér	temps (s)
64	512	1,02e-03	1,12e-03	1014,6	0,309
128	1024	4,72e-04	5,03e-04	1034,1	0,344
256	2048	2,83e-04	3,12e-04	1213,1	0,461
512	4096	1,92e-04	2,14e-04	1411,3	0,623
1024	8192	1,26e-04	1,39e-04	1437,8	0,978
2048	16384	8,71e-05	9,64e-05	1573,0	1,985
4096	32768	6,25e-05	6,97e-05	1760,5	4,532
8192	65536	4,37e-05	4,88e-05	2017,2	11,036
16384	131072	3,09e-05	3,46e-05	2259,5	28,337

Tableau 5.7: Résultats pour les DCT partielles sur une carte graphique NVIDIA 8800 GTS.

m	n	tolérance	relative	#itér	temps (s)
64	512	1,02e-03	1,12e-03	1015,6	0,177
128	1024	4,72e-04	5,03e-04	1033,8	0,222
256	2048	2,83e-04	3,12e-04	1215,6	0,298
512	4096	1,92e-04	2,14e-04	1408,5	0,404
1024	8192	1,26e-04	1,39e-04	1428,1	0,500
2048	16384	8,71e-05	9,65e-05	1563,1	0,957
4096	32768	6,25e-05	6,97e-05	1757,2	2,011
8192	65536	4,37e-05	4,88e-05	2001,5	4,586
16384	131072	3,09e-05	3,46e-05	2251,2	11,274

Tableau 5.8: Résultats pour les DCT partielles sur une carte graphique NVIDIA GTX 275.

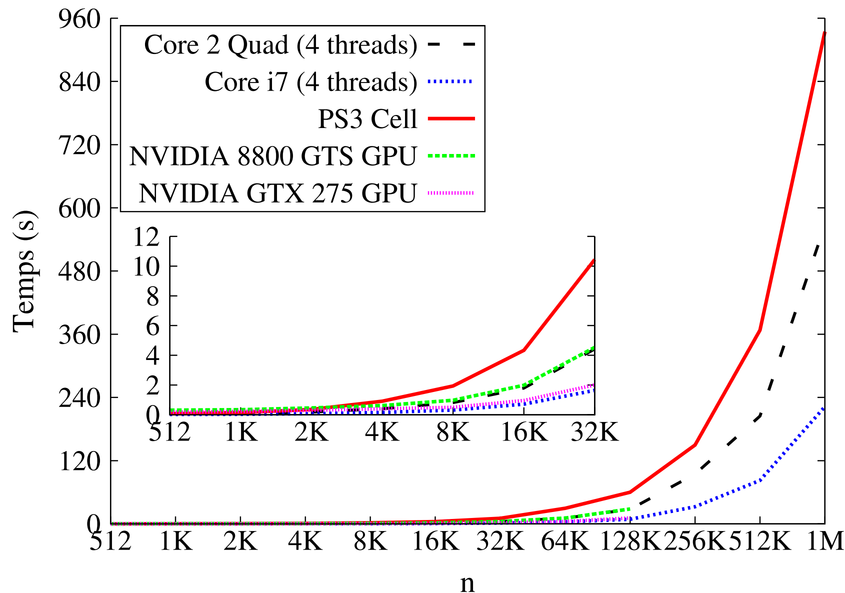

Figure 5.3: Résultats pour les DCT partielles sur les cinq plateformes considérées.

alors que celui de la NVIDIA GTX 275 fait face au Core i7 920. Dans les deux cas, le CPU est seulement très légèrement plus rapide que le GPU. Notre implémentation sur Cell est plus lente que notre implémentation sur CPU. Cela est principalement dû au fait que beaucoup de calculs sont réalisés sur le PPE (même si des instructions AltiVec sont utilisées pour améliorer les performances), alors qu'il faudrait les réaliser sur les SPE pour en tirer toute la quintescence. Rappelons que ceci n'est pas possible avec la version de FFTW utilisée pour notre implémentation et nos expériences. De plus, notons que le Cell est désormais une puce qui accuse son âge et que la version utilisée ici ne dispose que de 6 SPE. Nos implémentations sur GPU et sur Cell donnent une borne inférieure sur les performances que l'on peut attendre sur ces plateformes, avec un effort de développement minimal.

Dans le but de se comparer à une méthode classique de résolution du CS, la figure 5.4 montre une comparaison sur GPU entre notre méthode de résolution basée sur les itérations du point proximal et l'implémentation du *matching pursuit* issue de l'article [4]. Pour effectuer une comparaison juste, les deux implémentations utilisent la bibliothèque NVIDIA CUBLAS pour les opérations matricielles et l'implémentation du *matching pursuit* a été modifée pour utiliser des nombres flottants simple précision (comme notre implémentation). Pour des raisons techniques et pour obtenir une comparaison fiable entre les deux méthodes, les matrices utilisées sont des matrices de DCT partielles (représentées explicitement) et ici $\frac{m}{n} = \frac{1}{4}$. Cette expérience a été conduite de telle sorte que l'erreur de tolérance finale soit la même pour les deux méthodes. Rappelons que notre implémentation sur GPU est particulièrement simple

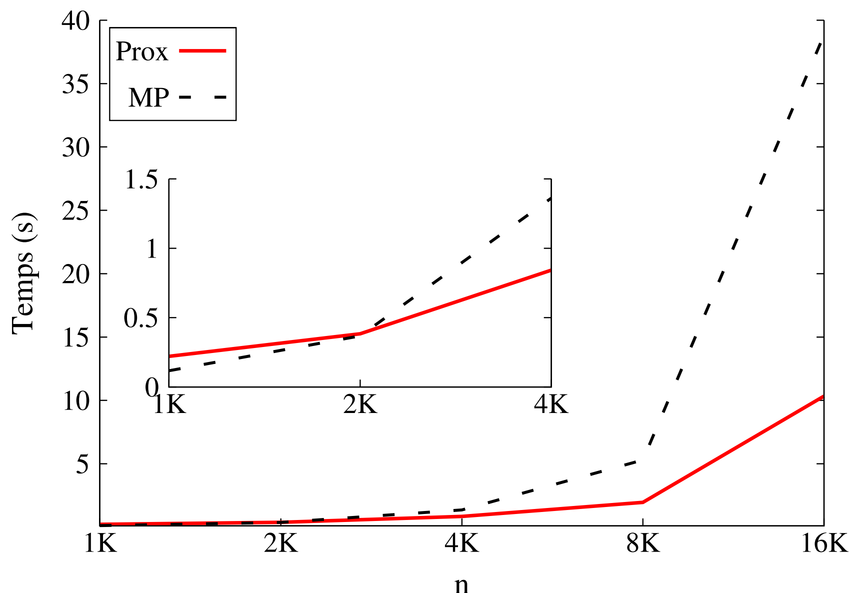

Figure 5.4: Comparaison GPU entre notre méthode (Prox) et l'algorithme *matching pursuit* (MP) sur un GPU NVIDIA GTX 275 avec des matrices de DCT partielles (représentées explicitement). $\frac{m}{n} = \frac{1}{4}$ et paramètres amenant à la même erreur de tolérance pour les deux méthodes.

et qu'elle n'est pas aussi optimisée que notre implémentation sur CPU.

Nous observons ici que, pour des matrices de taille 256×1024, le parallélisme n'est pas suffisant pour masquer les opérations plus complexes de notre méthode. Pour une taille de 512×2048, les deux méthodes obtiennent des performances similaires et au-delà de celle-ci notre méthode devient plus rapide. En effet, notre méthode devient presque 4 fois plus rapide pour une taille de matrices de 4096×16384. De plus, on peut aussi observer qu'à partir de 512×2048 notre méthode passe mieux à l'échelle que le *matching pursuit*. Ainsi, on peut s'attendre à ce que plus l'instance soit grande plus le gain engendré par l'utilisation de notre méthode soit élevé, ce qui est d'autant plus un bon point pour notre méthode que notre implémentation GPU est loin d'être optimale.

5.4 Conclusion

Dans ce chapitre, nous avons présenté un algorithme approché et rapide pour résoudre le problème du CS. Cet algorithme repose sur la méthode classique des itérations du point proximal et a spécialement été conçu pour tirer parti des architectures parallèles. Ainsi, il permet d'obtenir de très bons temps de calculs, tout en étant facile à implémenter sur ces différentes architectures. Notre algorithme et son

implémentation peuvent travailler soit avec des matrices représentées explicitement soit avec des transformées rapides. Une variante exacte de notre algorithme a aussi été présentée. Cette dernière est valable pour tout type de représentation de matrices, mais est uniquement efficace pour une représentation explicite. Pour valider notre approche, nous avons proposé des implémentations sur différentes plateformes parallèles (CPU, GPU et Cell) et effectué un certain nombre d'expériences dans lesquelles nous montrons les avantages et inconvénients liés aux différentes implémentations et aux différentes architectures.

CHAP. 6

VÉRIFICATION APPROCHÉE SUR ARCHITECTURE PARALLÈLE

Ce chapitre s'inscrit dans le contexte de la vérification de modèles sur des architectures parallèles. En l'occurrence, nous nous intéressons à la vérification approchée de modèles basée sur l'échantillonnage d'exécutions du système étudié. La vérification de propriétés temporelles est réalisée sur ces chemins d'exécution. Nous présentons une implémentation parallèle d'APMC (*Approximate Probabilistic Verification for Markov chains*) qui utilise le modèle *Bulk Synchronous Parallelism* (BSP) et qui permet d'obtenir une implémentation efficace sur trois architectures d'ordinateur parallèles : *cluster*, SMP (*Symmetric MultiProcessing*) et *cluster* de SMP (hybride). Une seule et unique implémentation avec la bibliothèque de haut niveau BSP++ [72] permet d'arriver à ce résultat, tout en laissant le choix d'utiliser MPI [60], OpenMP [41] ou MPI+OpenMP, ce qui est un avantage particulièrement important. Des expériences sont effectuées pour montrer les bénéfices de cette approche et comparer les différentes plateformes, ainsi que les différents moyens de parallélisation. Nous verrons que le cas du Cell est problématique dans ce contexte, c'est pourquoi une section complète est dédiée à une implémentation plus appropriée pour ce processeur.

La section 6.1 introduit le problème, expose un état de l'art rapide, ainsi que la méthode utilisée tout au long de ce chapitre : APMC. L'architecture logicielle d'APMC 3.0 est présentée dans la section 6.2. La section 6.3 porte sur l'utilisation du modèle BSP et de la bibliothèque BSP++ pour paralléliser APMC. Finalement, la section 6.4 traite le cas particulier de l'architecture Cell et propose une nouvelle implémentation d'APMC : APMC-CA.

Ce travail a donné lieu à deux articles acceptés aux conférences *PDMC* [71] et *QEST* [25].

6.1 Contexte

Tout d'abord, nous introduisons le contexte dans lequel nous allons travailler, en particulier le problème traité, un état de l'art rapide et la méthode APMC.

6.1.1 Problème

La vérification de modèles est un domaine d'étude qui s'intéresse aux systèmes à ensemble fini d'états. Le problème consiste à vérifier si un modèle satisfait une spécification ou non. Ce problème trouve des applications dans les circuits électroniques, certains protocoles de communication, la programmation, etc. Cependant, dans le cadre de la vérification classique de tels systèmes, un phénomène d'explosion combinatoire de l'espace des états est occasionné. Cela consiste en l'augmentation considérable de la taille des données entre la description implicite dans un langage ad hoc et la structure de travail effectivement en mémoire. Pour éviter cette explosion combinatoire, des méthodes probabilistes de résolution approchée peuvent être utilisées, et en particulier des méthodes approchées basées sur l'échantillonnage d'exécutions du système étudié.

6.1.2 Etat de l'art

Plusieurs méthodes permettent de vérifier des formules de logique temporelle sur des systèmes probabilistes (voire même concurrents probabilistes).

Dans le cadre de la vérification de propriétés qualitatives, qui consiste à vérifier qu'une formule est satisfaite avec probabilité 0 ou 1, nous pouvons citer [118]. Ce dernier utilise une technique à base d'automates. Nous pouvons aussi citer l'approche par échantillonnage d'exécutions de [127], implémentée dans Ymer.

Nous nous intéressons ici plus particulièrement à la vérification de propriétés quantitatives. Il s'agit de calculer la probabilité qu'un système probabiliste modélisé par une chaîne de Markov satisfait une formule de logique temporelle linéaire (LTL).

L'un des premiers algorithmes pour la vérification quantitative a été donné par Courcoubetis et Yannakakis [40]. L'algorithme transforme pas à pas la chaîne de Markov et la formule, en éliminant un par un les connecteurs temporels, tout en préservant la probabilité de satisfaction de la formule. Cette élimination est réalisée en résolvant un système d'équations linéaires de la taille de la chaîne de Markov. Cet algorithme souffre très clairement de problèmes de complexité en espace. Cette méthode est celle utilisée pour la vérification de propriétés quantitatives dans l'outil de vérification PRISM [3].

6.1.3 Méthode de résolution APMC

La vérification probabiliste de chaînes de Markov peut être efficacement approchée [75, 90]. Ces travaux ont mené à la méthode APMC (pour *Approximate Probabilistic Verification for Markov chains*). Dans [90] il est montré que la probabilité de satisfaction de propriétés LTL monotones ou anti-monotones peut être approchée par un schéma d'approximation randomisé totalement polynomial (FPRAS). Deux types de chaînes

de Markov peuvent être utilisées : des chaînes de Markov à temps discret (DTMC pour *Discrete Time Markov Chain*) ou à temps continu (CTMC pour *Continuous Time Markov Chain*). Soit $Prob[\psi]$ la mesure de probabilité de l'ensemble des chemins d'exécution satisfaisant la propriété ψ et $Prob_k[\psi]$ la mesure de probabilité associée à l'espace probabiliste de chemins d'exécution de taille finie k. Etant donné une DTMC ou une CTMC $\mathcal{M}$ et une propriété monotone ψ, on peut approcher $Prob[\psi]$ par un algorithme de point fixe obtenu en itérant un schéma d'approximation randomisé pour $Prob_k[\psi]$.

La notion de schéma d'approximation randomisée pour les problèmes de comptage, issue de Karp et Luby [81], a été adaptée pour obtenir l'algorithme d'échantillonnage $\mathcal{AGA}$ suivant, qui est un schéma d'approximation totalement polynomial. Il utilise un générateur aléatoire G pour $\mathcal{M}$ pour calculer une bonne approximation de $Prob_k[\psi]$.

Algorithme 6 Algorithme générique d'approximation (AGA)

Entrée : $G, k, \psi, \varepsilon, \delta$
Sortie : ε-approximation de $Prob_k[\psi]$

1. $N := \ln(\frac{2}{\delta})/2\varepsilon^2$
2. $A := 0$
3. Pour $i = 1$ à N faire
   a) Générer un chemin aléatoire σ de longueur k
   b) Si ψ est vraie sur σ alors $A := A + 1$
4. Retourner $Y = A/N$

Un générateur aléatoire est utilisé pour générer des chemins d'exécution et calculer la variable aléatoire Y qui approche $p = Prob_k[\psi]$. L'approximation obtenue sera correcte, c'est-à-dire $|Y - p| < \varepsilon$ (erreur additive), avec confiance $(1 - \delta)$, après un nombre polynomial d'échantillonages en $\frac{1}{\epsilon}, \log \frac{1}{\delta}$.

6.2 Architecture logicielle d'APMC 3.0

APMC 3.0 se présente sous la forme d'un compilateur qui prend en entrée un fichier pour le modèle dans une variante du format *Reactive Module* (utilisé par PRISM [77]) et un fichier pour la ou les formules LTL à vérifier. Le compilateur est écrit en Java (comme l'analyseur syntaxique de PRISM a été réutilisé et que celui-ci est écrit en Java) et génère un fichier C qui contient une implémentation de l'algorithme décrit dans la section 6.1.3 et une représentation succincte du modèle et des propriétés (linéaires dans la taille de leur fichier respectif). Une fois compilé, le programme (ou déployeur) pourra être lancé pour effectuer la vérification. Lors de l'appel de ce programme, plusieurs paramètres sont à spécifier : le nombre de chemins à générer et leur taille.

Au moment de la compilation du fichier C pour obtenir le déployeur, l'utilisateur peut choisir entre deux fichiers endcode.h fournis avec APMC 3.0. Le premier correspond à une implémentation séquentielle alors que le second correspond à une implémentation distribuée utilisant MPI. Si un nouveau schéma de parallélisation doit être mis en place, cela peut être très facilement fait en remplaçant le fichier endcode.h par une nouvelle implémentation. La figure 6.1 montre une vue d'ensemble de la compilation du déployeur.

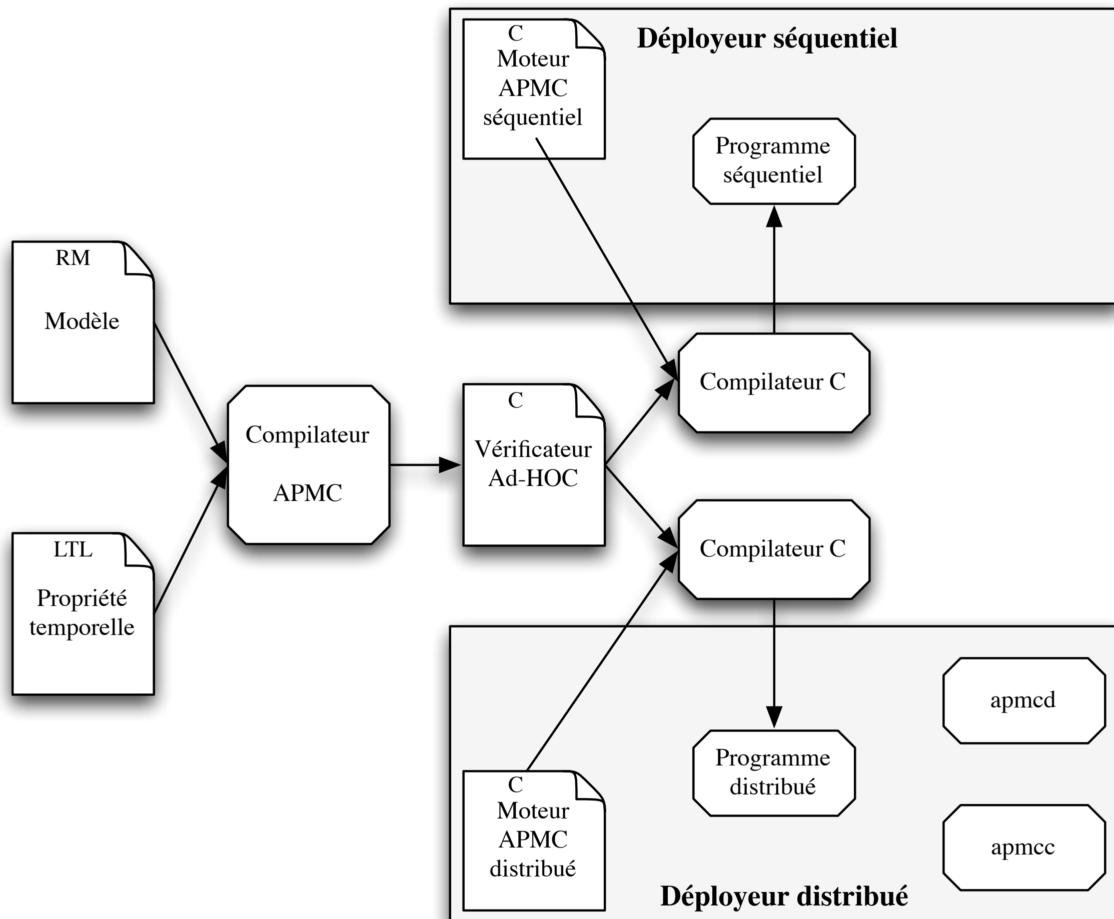

Figure 6.1: APMC 3.0 – Compilation du déployeur.

Le déployeur d'APMC 3.0 génère aléatoirement des chemins dans l'espace probabiliste sous-jacent au système et vérifie au fur et à mesure si les propriétés sont satisfaites sur ces chemins. A partir de ces résultats, pour chaque propriété, la probabilité qu'elle soit satisfaite est alors calculée. Pour la parallélisation, elle consiste simplement à générer des chemins et les vérifier séparément, comme ceux-ci sont indépendants les uns des autres, et ensuite à réunir les résultats (c'est-à-dire, le nombre de chemins satisfaisant la propriété et le nombre total de chemins testés). Ce schéma de parallélisation a aussi été utilisé dans le cadre du vérificateur de modèles PRISM [77].

En pratique, la représentation succincte du système sur laquelle repose APMC est primordiale, car elle permet de n'utiliser que très peu de mémoire, ce qui avantage grandement APMC par rapport aux méthodes classiques de vérification de modèles, qui sont sujettes à une explosion combinatoire de l'espace des états. Cela permet, entre autres, à APMC 3.0 de travailler sur des tailles de problèmes beaucoup plus grandes.

Pour plus de détails sur l'architecture logicielle d'APMC, le lecteur intéressé peut se référer à [111].

6.3 Utilisation du modèle BSP pour parallélisation semi-automatique

Dans cette section, nous présentons une implémentation parallèle d'APMC qui utilise le modèle *Bulk Synchronous Parallelism* (BSP) dans le but d'obtenir de très bonnes performances sur trois architectures d'ordinateur parallèles : *clusters*, SMP et *clusters* de SMP (hybride). Précédemment (voir section 6.2), APMC a été parallélisé à la main, mais l'utilisation de la bibliothèque BSP++ [72] permet d'obtenir de meilleures performances, est plus simple à utiliser (pas besoin d'un expert en programmation parallèle), et permet de gérer plusieurs architectures différentes et plusieurs moyens de parallélisation différents, avec une seule implémentation.

Des expériences sont réalisées pour montrer l'intérêt du *framework*, mais aussi quels types d'implémentation et d'architecture sont les plus adaptés pour utiliser APMC (et probablement la plupart des méthodes de vérification de modèles reposant sur de l'échantillonnage).

Les plateformes à base de *clusters* de SMP sont les solutions les plus rentables pour les applications à grande échelle. Dans la liste Top500[1] des superordinateurs les plus rapides, la plupart reposent sur des *clusters* de SMP. Des travaux tels que [85,119] ont montré l'amélioration des performances dans le cadre de l'utilisation du modèle SPMD (*Single Process Multiple Data*) sur une architecture à mémoire partagée avec OpenMP par rapport à MPI. Ceci est dû au fait que les communications et les synchronisations par mémoire partagée sont généralement plus rapides que dans un cadre distribué. Une combinaison de MPI et d'OpenMP est considérée comme un modèle approprié pour de telles plateformes : utiliser MPI pour communiquer entre les noeuds et OpenMP pour la parallélisation à l'intérieur d'un noeur SMP [33,37,51,66,80,96]. Cependant, écrire de tels programmes hybrides est une tâche complexe, comme elle repose sur des langages de programmation de bas niveau. La littérature abonde de propositions pour des langages de haut niveau et des modèles pour simplifier le développement d'applications sur machines parallèles [110]. Ces modèles sont souvent basés sur des motifs contraints, qui cachent la "glue" requise pour réaliser les calculs de manière parallèle et permettent aux utilisateurs de se concentrer sur la partie calculatoire du programme, qui est spécifique à l'application. De tels modèles incluent les patrons de conception parallèles [109], les

[1]Voir http://www.top500.org pour la liste des 500 superordinateurs les plus rapides.

squelettes algorithmiques [38] et le modèle *Bulk Synchronous Parallelism* (BSP) [116].

6.3.1 Modèle BSP

Le modèle *Bulk Synchronous Parallel* (BSP) a été introduit par Leslie G. Valiant comme un pont entre le matériel et le logiciel pour simplifier le développement d'algorithmes parallèles. De manière générale, le modèle BSP est défini par trois éléments :

- **Un modèle de machine**, qui décrit une machine parallèle comme un ensemble de processeurs liés à travers un moyen de communication supportant les communications point-à-point et les synchronisations. De telles machines sont alors décrites par un ensemble de paramètres expérimentaux [16]: le nombre de processeurs P, la vitesse de processeurs r en FLOPS, la vitesse de communication g et la durée de synchronisation L.

- **Un modèle de programmation**, qui décrit comment un programme parallèle est structuré (voir figure 6.2). Un programme BSP consiste en une séquence de super-étapes (ou phases), dans lesquelles chaque processus effectue des calculs locaux suivis par des communications. Quand tous les processus atteignent la barrière de synchronisation, la prochaine super-étape commence.

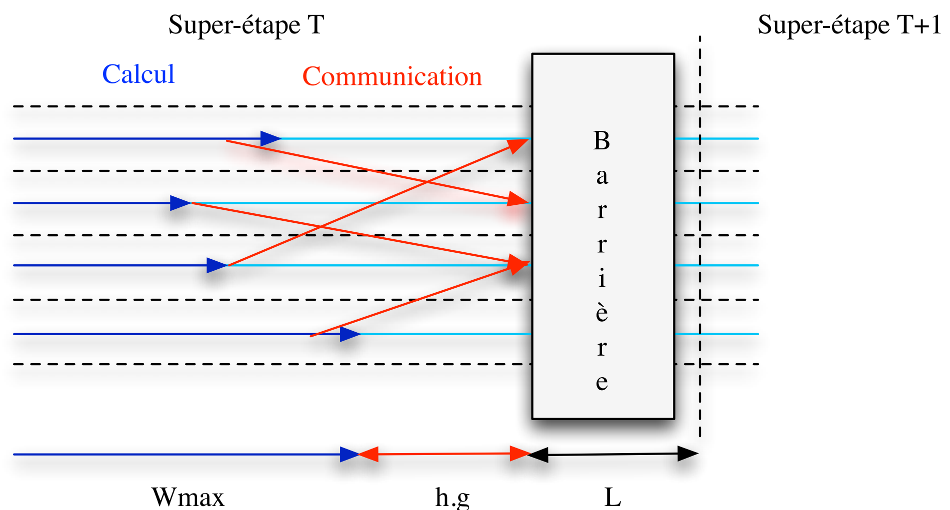

Figure 6.2: Vue d'ensemble du modèle de programmation BSP.

- **Un modèle de coût**, qui estime la durée δ d'une super-étape comme une simple fonction combinant les paramètres de la machine (L and g), la quantité maximale de données qui est envoyée ou reçue par un processeur (h) et la charge de travail maximale pour tous les processeurs (W_{max}). Pour toute super-étape donnée, la durée δ est donnée par :

$$\delta = W_{max} + h.g + L$$

Les implémentations classiques du modèle BSP incluent : l'*Oxford BSP Toolset* [76], le *Paderborn University BSP* (PUB) [20] et le langage *Bulk Synchronous Parallel ML* (BSML) [65], qui introduit BSP comme une extension parallèle de ML. Un grand nombre d'extensions du modèle BSP classique ont été proposées comme Oblivious-BSP [69] ou H-BSP [122], qui essayent d'améliorer la justesse du modèle de coût ou de supporter des machines hétérogènes.

6.3.2 Bibliothèque BSP++

L'interface de BSP++ [72] est une implémentation orientée objet de la bibliothèque fonctionnelle BSML [64]. Pour localiser la source d'un potentiel parallélisme, elle fournit une interface structurée au modèle BSP fondée sur la notion de *vecteur de données parallèles*. Dans ce modèle, l'utilisateur stocke ses données distribuées dans une classe générique spécialisée, nommée *par*, et qui supporte des motifs de communications de style BSP. L'interface de BSP++ se résume essentiellement en les éléments suivants :

- `par<T>`, qui encapsule le concept de vecteur parallèle. Cette classe peut être construite à partir d'une large sélection de constructions C++, allant du tableau C, des conteneurs standards et des fonctions C++, aux lambda fonctions. L'accès local à un vecteur de données parallèles est effectué au travers de l'opérateur de déréférencement (listing 6.1, ligne 13);
- `pid_` est un vecteur parallèle global qui contient les PID;
- `sync`, qui effectue une synchronisation explicite (via `MPI_Barrier` ou le pragma `barrier` d'OpenMP) et achève la super-étape courante;
- `proj`, qui, étant donné un vecteur parallèle, retourne un objet fonction qui associe le PID d'un processeur à la valeur contenue dans un vecteur parallèle par ce processeur, et achève la super-étape courante. Ceci est réalisé via une étape de communication qui utilise un appel à `MPI_Allgather` pour l'implémentation MPI et une copie asynchrone vers un segment de mémoire partagée pour OpenMP. Après un appel à `proj`, tous les processeurs d'une machine BSP ont une copie locale du vecteur distribué;
- `put`, qui permet à toute valeur locale d'être transférée à n'importe quel autre processeur et achève la super-étape courante. Le paramètre de `put` est un vecteur parallèle qui, à chaque processeur, contient un objet fonction de type `T(int)`, qui retourne les données à être envoyées au processeur i quand on y applique i. `put` construit une matrice distribuée $\mathcal{P}$ de telle sorte que $\mathcal{P}_{ij}$ contienne les valeurs que le processeur i envoie au processeur j et calcule la transposée de $\mathcal{P}$. Ce transfert est effectué soit par `MPI_Alltoall` soit en construisant la matrice $\mathcal{P}$ dans un segment mémoire partagé. L'appel de `put` retourne alors un vecteur parallèle d'objets fonctions de type `T(int)` qui retournent les données reçues par le processeur i quand on y applique i. Contrairement à `proj`, cette primitive permet tout type de schéma de communication.

Nous pouvons remarquer que la sémantique entière de BSP est encapsulée à l'intérieur de ces éléments. Le fait que ceux-ci soient réduits à un très petit nombre limite l'impact sur le code utilisateur. L'interface avec des constructions C++ ou des fonctions C permet une intégration simple de code *legacy* dans des sections BSP.

Le listing 6.1 montre comment une fonction de produit intérieur peut être écrite en utilisant la bibliothèque BSP++ :

Listing 6.1: BSP++ inner product

```
1  #include <bsppp/bsppp.hpp>
2
3  int main(int argc, char** argv)
4  {
5    bsp::init(argc,argv);
6
7    BSP_SECTION()
8    {
9      bsp::par< vector<double> > v;
10     bsp::par< double >         r;
11
12     // super-étape (1) : réalise le calcul local...
13     *r = std::inner_product(v->begin(), v->end(),
14                             v->begin(), 0.);
15
16     // ... et effecture un échange global
17     bsp::result_of::proj<double> exch = proj(r);
18
19     // super-étape (2) : accummule le résultat partiel
20     *r = std::accumulate(exch.begin(), exch.end());
21     bsp::sync();
22   }
23
24   bsp::finalize();
25 }
```

Après l'initialisation de l'environnement lié à l'utilisation de la bibliothèque BSP++ à la ligne 5, une section BSP commence avec la macro `BSP_SECTION()`. Alors, une instance de vecteur parallèle est créée pour stocker les données et le résultat (lignes 9 et 10). L'algorithme est alors divisé en deux super-étapes :

- La première réalise un calcul local de produit intérieur de `v` en utilisant l'algorithme fourni par la bibliothèque standard du C++ (ligne 13), la STL. Une fois les résultats calculés, un échange global de ces résultats intermédiaires est effectué par la primitive `proj` (ligne 16). Cette dernière récupère les données locales de tous les processeurs de la machine BSP et les envoie à tous les autres processeurs, en retournant un objet qui contient localement les valeurs

des données distribuées. `proj` synchronise la machine et achève la super-étape;

- La seconde super-étape utilise cet objet comme un conteneur standard C++ pour appeler génériquement `std::accumulate` pour effectuer la réduction finale (ligne 19). Une synchronisation explicite est alors réalisée pour achever la super-étape (ligne 20).

Le programme BSP se termine après l'appel de la fonction `finalize` de la bibliothèque BSP++ (ligne 23).

Le fait que le code ne fasse aucune référence à MPI ou OpenMP est une caractéristique particulièrement importante. Le choix de ce support est effectué par un symbole de préprocesseur passé à la compilation : `-DBSP_OMP_TARGET` pour OpenMP, `-DBSP_MPI_TARGET` pour MPI ou `-DBSP_CELL_TARGET` pour le Cell. Une autre caractéristique importante réside dans le fait que tous les objets retournés par les différentes primitives de communication s'interfacent naturellement avec la plupart des idiomes et des bibliothèques C++ (telles que la STL ou Boost[2]), ce qui permet leur utilisation directe.

6.3.3 Support pour programmation hybride

L'une des objections communes à l'utilisation du modèle BSP est le coût des synchronisations globales qui peuvent devenir dominantes dans le cadre de machines parallèles de forte taille. Pour réduire ce coût, plusieurs propositions ont été faites : par exemple, le modèle dBSP [11] peut se partitionner dynamiquement en de plus petites parties, se comportant elles-mêmes comme des ordinateurs BSP avec des nombres de processeurs et des temps de synchronisation plus faibles; d'autres comme [95] proposent d'utiliser des primitives de haute performance, mais qui s'éloignent du modèle BSP pour assurer de meilleures performances au niveau des synchronisations.

Dans le cadre de *clusters* de SMP, nous proposons de tirer avantage des communications efficaces entre les coeurs en SMP en utilisant OpenMP pour minimiser le coût de synchronisation et effectuer des communications rapides, et de limiter le surcoût de l'ordonnancement d'OpenMP en utilisant un modèle SPMD [85].

	MPI			OpenMP		
P	4	8	16	4	8	16
g	$0,087$	$0,22$	$1,69$	$0,025$	$0,069$	$0,68$
L	$4,46$	$20,8$	$108,0$	$2,94$	$8,13$	$13,1$

Tableau 6.1: Variation de L (en ms) et g (en secondes par Mo) sur un *cluster* de 4 processeurs avec chacun 4 coeurs.

[2] http://www.boost.org

Le tableau 6.1 montre l'évaluation des paramètres BSP g et L par rapport au moyen de communication utilisé (MPI ou OpenMP) et au nombre de coeurs. La synchronisation via OpenMP sur un coeur donné est entre $1,5$ et 8 fois plus rapide que la même étape de synchronisation avec MPI. De manière similaire, la communication via la mémoire partagée est $1,5$ à 8 fois plus rapide.

La bibliothèque BSP++ supporte l'utilisation d'OpenMP et de MPI de telle sorte que les utilisateurs puissent composer des super-étapes MPI avec des super-étapes OpenMP. Le support de code hybride est alors assuré quand une super-étape OpenMP est embarquée dans une super-étape MPI, comme montré dans la figure 6.3. La figure 6.3.a montre l'agencement classique d'une super-étape MPI. Pour permettre des calculs hybrides (figure 6.3.b), la phase de communication de la super-étape MPI est remplacée par un appel à une fonction BSP OpenMP. En général, rien de plus n'est nécessaire, mais, dans certains cas, on peut noter qu'une copie des données MPI dans chaque *thread* privé OpenMP peut être requise.

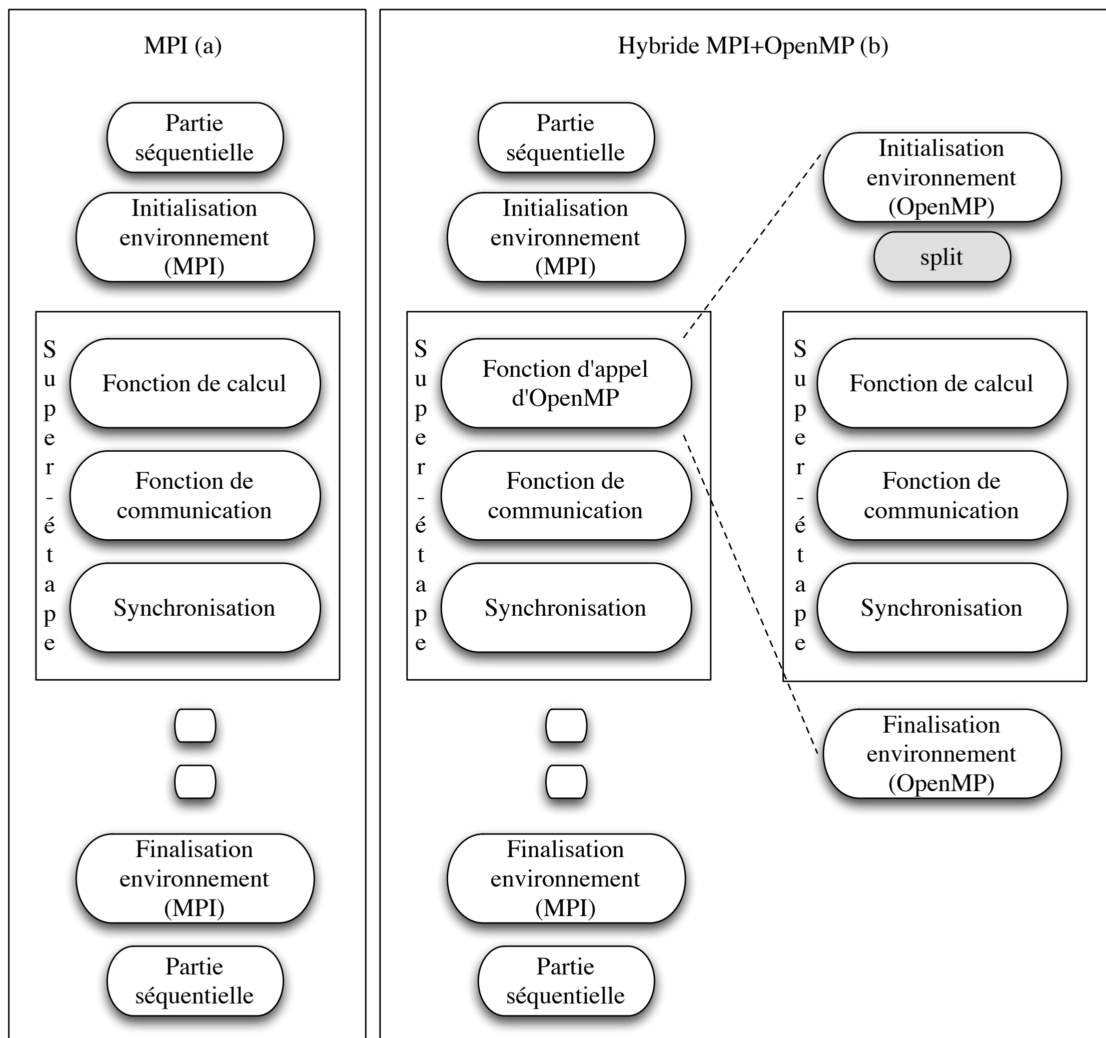

Figure 6.3: Parallélisation avec le modèle hybride.

6.3.4 Expériences

Conditions expérimentales

Les expériences présentées ici ont été conduites sur deux plateformes différentes :

- La première machine, la machine **AMD**, dispose de 4 processeurs ayant chacun 4 coeurs. Chaque processeur est un AMD Opteron 8354 2GHz avec 2Mo de cache de niveau 3 partagé. La machine dispose de 16Go de RAM et fonctionne sous Linux (noyau 2.6.26). Le compilateur C++ utilisé est g++ 4.3, qui supporte OpenMP 2.0, et en ce qui concerne MPI, c'est openMPI qui a été utilisé. Dans toutes les expériences, l'association tâche/coeur a été fixée en utilisant l'appel système *sched_setaffinity* pour éviter la migration de *threads* entre coeurs et obtenir des performances stables.

- La seconde machine, la machine **CLUSTER**, est un *cluster* de la plateforme GRID5000 [32]. Nous avons utilisé entre 2 et 64 noeuds connectés par un double réseau gigabit ethernet BCM5704. Chaque noeud dispose de 2 processeurs à 2 coeurs et de 4Go de RAM. Les processeurs sont des AMD Opteron 2218 à 2,6GHz, avec chacun 2 $\times$ 2Mo de cache de niveau 2. Pour MPI, la bibliothèque MPICH2.1.0.6 est utilisée.

L'implémentation repose sur celle d'APMC 3.0. En effet, la seule modification réside dans le fichier endcode.h qui a été modifié pour utiliser la bibliothèque BSP++ (voir section 6.2).

Dans le but d'obtenir des figures plus claires, les résultats sont présentés en utilisant le *ralentissement* comme métrique. Le ralentissement pour un calcul spécifique sur une machine donnée est défini comme le temps d'exécution pour la machine n-coeurs multiplié par le nombre de coeurs n. Avec cette métrique, si le résultat pour une machine n-coeurs reste constant, l'accélération est linéaire. Si l'efficacité parallèle diminue quand le nombre de coeurs augmente, le ralentissement augmente. Dans le cas d'une accélération superlinéaire, le ralentissement diminue quand le nombre de coeurs augmente.

Nos expériences consistent en la vérification de propriétés temporelles sur deux modèles différents. Le premier est le dîner des philosophes [75, 108], pour lequel on vérifie une propriété d'accessibilité (c'est-à-dire une propriété de la forme $true\ U\ \phi$ avec ϕ une formule de la logique du premier ordre). Comme ce modèle est particulièrement connu, il permet de s'assurer qu'aucun comportement étrange n'apparaît durant le processus de vérification. Le paramètre de ce modèle est le nombre de philosophes qui interagissent entre eux. Le second modèle correspond à des réseaux de capteurs (voir [1, 46]). Son intérêt vient du fait qu'il est composé d'un grand nombre de modules de petite taille interagissant entre eux. Pour ce second modèle, on vérifie aussi une propriété d'accessibilité, qui correspond ici à ce qu'un capteur sur le bord de la grille ait bien l'information qu'un feu ait été détecté au milieu de celle-ci. Le paramètre de ce modèle est la taille de la grille de communication, c'est-à-dire que le modèle SNX contient X capteurs. Les deux modèles utilisés ici ont une représentation explicite, qui est de taille exponentielle par rapport aux paramètres.

La taille de chemin passé en paramètre pour le premier modèle est de 900, alors qu'elle est de 8000 pour le second.

MPI

Les figures 6.4 et 6.5 montrent le ralentissement pour les versions MPI d'APMC avec le dîner des philosophes et le réseau de capteurs sur les machines **AMD** et **CLUSTER**, respectivement. De manière plus précise, ces figures montrent le temps (en secondes, moyenne d'approximativement 120000 chemins) nécessaire à la vérification de la formule sur un chemin, par rapport au nombre de coeurs utilisés.

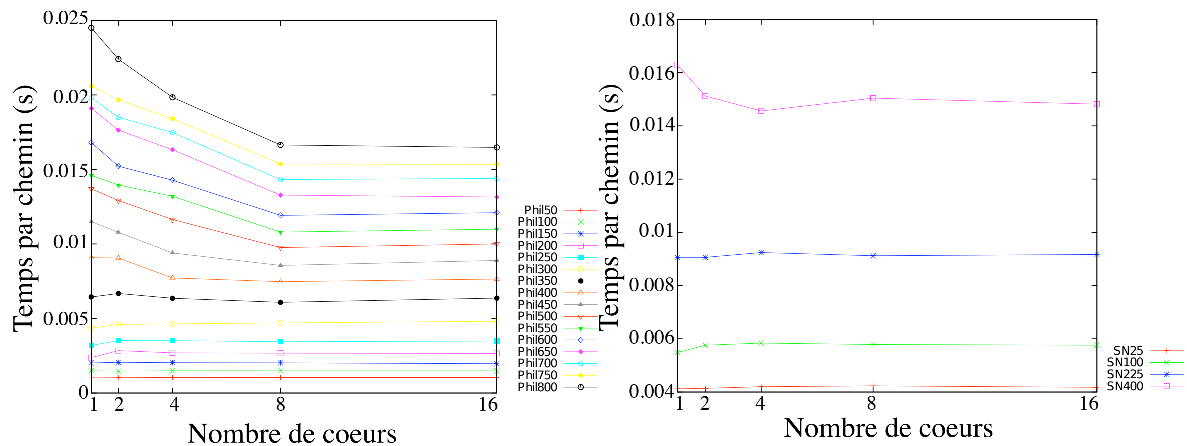

Figure 6.4: Résultats obtenus avec la version MPI pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur la **machine AMD**.

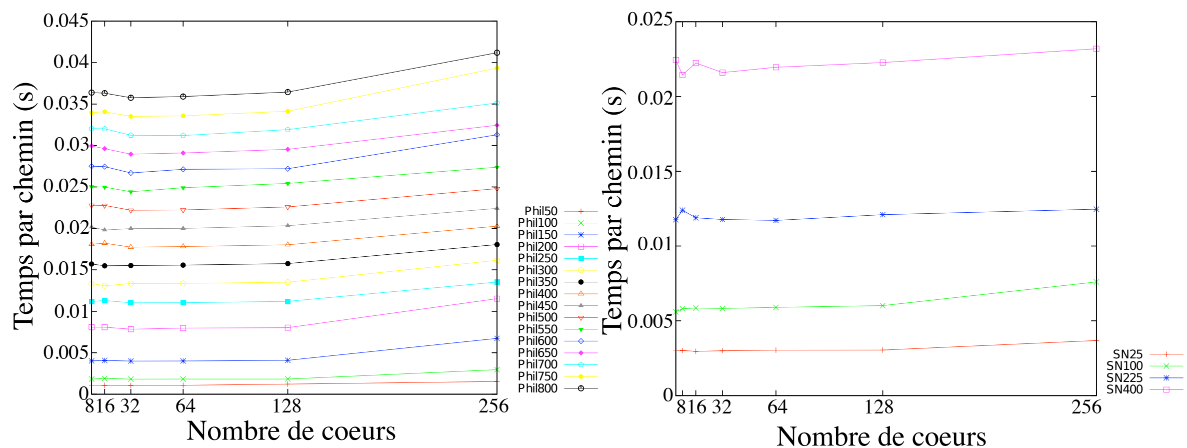

Figure 6.5: Résultats obtenus avec la version MPI pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur la **machine CLUSTER**.

Sur la machine **AMD**, on obtient une accélération superlinéaire de 1 à 8 coeurs qui devient linéaire avec 16 coeurs. Cela signifie que la parallélisation passe parfaitement à l'échelle : il n'y a pas de surcoût à utiliser BSP++. Sur la machine **CLUSTER**, les figures montrent une accélération linéaire jusqu'à 128 coeurs, et un léger ralentissement de 128 à 256 coeurs. Ce ralentissement est dû à la synchronisation, dont le temps augmente avec le nombre de coeurs.

OpenMP

La version OpenMP d'APMC donne des ralentissements légèrement différents pour les deux modèles. Ces résultats sont montrés par la figure 6.6. Encore une fois, ces figures montrent le temps (en secondes, moyenne d'approximativement 120000 chemins) nécessaire à la vérification de la formule sur un chemin par rapport au nombre de coeurs utilisés. Nous obtenons une accélération superlinéaire pour le dîner des philosophes, et une accélération linéaire pour le réseau de capteurs. Comparés aux résultats avec MPI, il n'y a pas de différence de temps d'exécution entre 1 et 8 coeurs. Avec 16 coeurs, la version OpenMP a un léger avantage, grâce au temps de synchronisation, qui est plus court avec OpenMP qu'avec MPI.

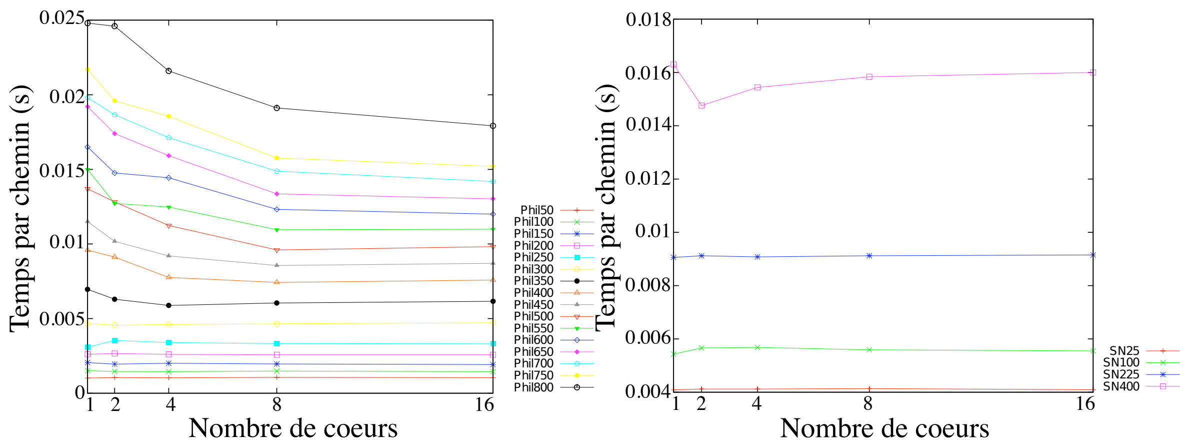

Figure 6.6: Résultats obtenus avec la version OpenMP pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur la **machine AMD**.

MPI+OpenMP

Comme les noeuds de la machine **CLUSTER** disposent de 2 processeurs à 2 coeurs, chaque noeud a son propre *thread* MPI qui lance 4 *threads* OpenMP. Par exemple pour 256 coeurs, on a 64 *threads* MPI $\times$ 4 *threads* OpenMP. Comparée à la version MPI, la version hybride MPI/OpenMP obtient de meilleures performances. Le temps de synchronisation est en effet réduit, grâce à l'utilisation d'une synchronisation à deux niveaux (OpenMP entre coeurs d'un même noeud et MPI entre noeuds).

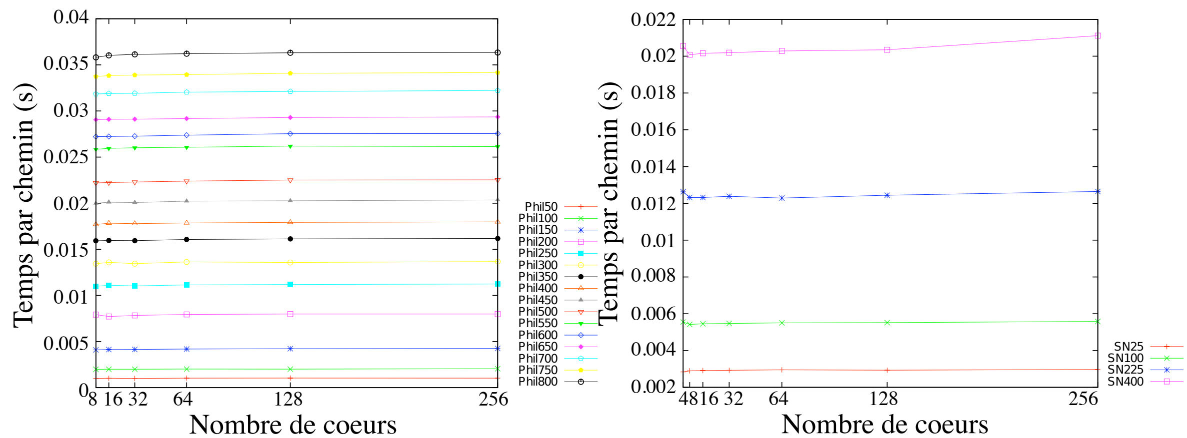

Figure 6.7: Résultats obtenus avec la version hybride MPI+OpenMP pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur la **machine CLUSTER**.

La figure 6.7 présente les résultats obtenus. Pour le dîner des philosophes, l'accélération liée à la version hybride va de 50% à 10% par rapport à la version MPI (respectivement pour 50 et 800 philosophes). En effet, plus la taille augmente, plus l'avantage de la version hybride se réduit. L'explication est la suivante : quand le nombre de philosophes est faible, le ratio temps de communication sur temps de calcul est élevé (comme le nombre de modules est exactement le nombre de philosophes), et donc la version hybride est avantagée. Quand le nombre de philosophes augmente, ce ratio s'affaiblit, et le temps de calcul devient prépondérant, ainsi l'avantage de la version hybride s'affaiblit et c'est en fait la version MPI qui devient de plus en plus efficace.

Les résultats de ralentissements de la figure 6.7 montrent que la version hybride MPI+OpenMP permet d'obtenir une accélération superlinéaire jusqu'à 256 coeurs.

Ces résultats indiquent clairement que la meilleure stratégie pour obtenir des performances dans le cadre de la vérification de modèles fondée sur l'échantillonnage, et en particulier APMC 3.0, est d'utiliser une architecture hybride (*cluster* de SMP) avec une version hybride du code (MPI+OpenMP).

Cell

Finalement, nous nous intéressons aux performances d'APMC sur l'architecture Cell. En l'occurrence, nous avons utilisé un processeur Cell à 3,2GHz disposant de 8 SPE. Pour plus de détails sur l'architecture du Cell, voir section 2.1.

La taille du code du déployeur et de la mémoire qu'il utilise augmente avec la taille des modèles et chaque SPE ne dispose que de 256Ko de mémoire locale. Ainsi, seuls des modèles de petite taille ont pu être ici considérés. Etant données les tailles de problèmes sur lesquelles on peut travailler, les tailles de chemin ont été revues à 28 pour le dîner des philosophes et à 69 pour le réseau de capteurs.

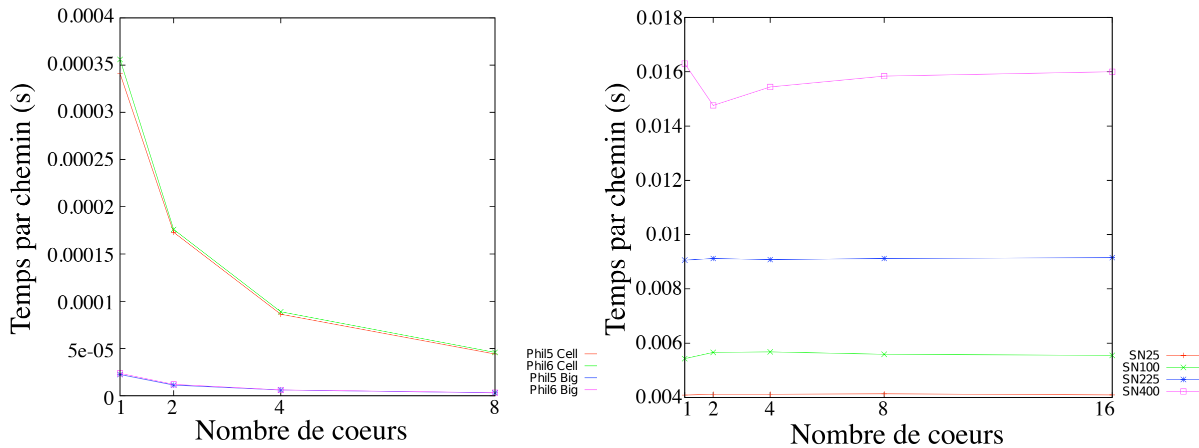

Figure 6.8: Comparaison entre le temps d'exécution pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur processeurs Cell et multi-coeur.

La figure 6.8 représente une comparaison entre les résultats obtenus sur le Cell et sur la machine **AMD** pour les deux modèles. Nous observons que la parallélisation sur Cell passe parfaitement à l'échelle, mais que les performances sont faibles comparées à celles d'un processeur multi-coeur classique. En effet, le Cell est jusqu'à 17 fois plus lent. Les résultats au niveau du passage à l'échelle montrent qu'il n'y a pas de surcoût à utiliser BSP++ et ceux en termes de performance montrent que l'architecture du Cell n'est pas adaptée à APMC. En effet, les opérations sur les gardes et les actions ne peuvent pas se vectoriser efficacement, alors que le Cell repose principalement sur cette caractéristique. De plus, le code abuse de branchements conditionnels, qui sont particulièrement coûteux sur Cell. Nous pouvons ajouter que, pour les mêmes raisons, la situation serait encore pire dans le cadre de calculs sur GPU.

Ces résultats montrent que l'architecture du Cell n'est pas adaptée à la vérification de modèle basée sur l'échantillonnage, telle qu'elle est implémentée dans APMC 3.0.

6.4 Adaptation d'APMC à l'architecture parallèle Cell

Comme les résultats sur Cell sont décevants, nous nous sommes intéressés au développement d'une version spécifique d'APMC pour tirer parti des spécificités de ce processeur (voir section 2.1). La nouvelle version, APMC-CA (pour *Cell Assisted*), va donc être conçue pour non seulement pouvoir travailler sur des modèles plus grands mais aussi obtenir de meilleures performances, dans l'optique d'une exécution sur Cell.

6.4.1 Nouvelle implémentation : APMC-CA

L'algorithme d'APMC et la méthode générale restent inchangés. Nous travaillons ici sur les représentations de données utilisées en interne par le déployeur et sur la

manière d'effectuer les différentes opérations pour les adapter à l'architecture du Cell.

APMC-CA dispose de la même architecture générale qu'APMC 3.0 (un compilateur et un déployeur, voir section 6.2). Mais comme APMC 3.0 génère un déployeur qui repose principalement sur du code scalaire (par opposition à vectoriel) et utilise abondamment la mémoire, il n'est pas adapté à l'architecture du Cell. APMC-CA est le résultat d'une re-conception d'APMC 3.0 pour minimiser les branchements conditionnels et l'utilisation mémoire. Le nouveau déployeur suit un schéma d'exécution simple : les SPE réalisent les calculs (génération de chemins et vérification) et le PPE récupère les résultats. Ainsi, comme les chemins générés par un SPE sont entièrement traités par celui-ci, il n'a donc pas à les communiquer (ni à un autre SPE, ni au PPE). La seule communication correspond donc aux résultats des SPE, c'est-à-dire le nombre de chemins satisfaisant la propriété et le nombre total de chemins testés, qui sont envoyés au PPE.

Nous décrivons maintenant l'implémentation d'APMC-CA, en comparant avec celle d'APMC 3.0.

Dans APMC 3.0, chaque module dispose de ses propres fonctions de garde et d'action. Ainsi, si un module avec n gardes et actions est renommé m fois, alors il y aura $n \times m$ fonctions de gardes générées et autant de fonctions d'actions. Avec APMC-CA, chaque *type* de module a ses propres fonctions de garde et d'action. A chaque module renommé correspond une instance de ce type de module et une *fonction d'association* spécifique est appelée à chaque fois que l'on travaille sur un nouveau module pour mettre à jour des pointeurs qui associent les variables du type de module aux bonnes données. Ainsi, la taille du programme est drastiquement réduite.

La génération de chemins d'APMC 3.0 est réalisée comme suit : une garde est choisie aléatoirement, puis évaluée, et si elle est satisfaite l'action associée est réalisée, sinon le processus est relancé. Dans le cadre d'APMC-CA, c'est désormais un type de module qui est choisi aléatoirement, puis une instance de ce type. Toutes les gardes du module sont évaluées, l'une des gardes évaluées à *vrai* est choisie aléatoirement et l'action correspondante est effectuée. De cette manière, APMC-CA n'a pas à stocker ni travailler sur les gardes qui sont évaluées à *faux*. Au lieu de tester une garde parmi tous les modules, on teste donc ici toutes les gardes d'un module particulier, tout en préservant le côté aléatoire du processus. Comme le nombre de gardes à *vrai* est petit, cela permet d'économiser un grand nombre de branchements conditionnels, tout en utilisant la puissance de calcul du Cell à meilleur escient.

Dans le cadre de certains modèles (tels que le réseau de capteurs, voir section 6.4.2), certains modules peuvent disposer de variables, mais ne pas avoir de gardes et d'actions. Ceci facilite en effet l'écriture de certains modèles et permet en particulier de gérer les cas particuliers (par exemple, les bords de la grille pour le réseau de capteurs) sans avoir à écrire des modules avec des gardes ou des actions spéciales. Dans ce cas, en général, ces modules disposent d'un nombre de variables très réduit, le plus souvent 1. Avec des modules vides de gardes et d'actions, on peut ainsi gérer par renommage le cas général *et* les cas particuliers, alors qu'avec des modules devant gérer les cas particuliers de manière interne, cela voudrait dire ne pas tirer profit

du renommage pour ces cas particuliers. Ainsi, la clarté et la concision du modèle s'en retrouveraient affectées. En contrepartie, gérer ainsi les cas particuliers signifie ajouter des modules pour les cas particuliers, et donc utiliser plus de mémoire, mais aussi pouvoir tirer un tel module lors de la phase de vérification. Cependant, comme ils sont vides de gardes et d'actions l'implémentation tire parti de cette caractéristique et ne les prend pas en compte lors du tirage aléatoire. De plus, comme ces modules n'ont ni garde ni action, aucune fonction d'association n'est générée. Comme en plus ils ne disposent bien souvent que d'une seule variable (ou de très peu), l'utilisation mémoire est très réduite.

De plus, l'opérateur *Until* utilisé pour la phase de vérification est implémenté de manière récursive dans APMC 3.0. Il a été réimplémenté ici itérativement pour éviter tout débordement de pile et minimiser le coût lié aux branchements conditionnels.

6.4.2 Expériences

Dans cette section, nous regroupons les différentes expériences réalisées pour montrer l'efficacité d'APMC-CA par rapport à APMC 3.0 pour le processeur Cell.

Conditions expérimentales

Pour montrer l'efficacité d'APMC-CA, on utilise deux plateformes : l'une basée sur un processeur Cell et la seconde sur un processeur classique. Le processeur Cell utilisé est celui d'une Sony Playstation 3, qui permet d'utiliser 6 SPE à 3,2GHz et 192Mo de RAM. Pour plus de détails sur l'architecture du Cell, voir section 2.1. La plateforme de référence est composée d'un processeur Intel Core 2 Duo 2,13GHz avec 2Mo de cache de niveau 2 et 2Go de RAM.

Nos expériences consistent en la vérification de propriétés temporelles sur deux modèles différents. Le premier est le dîner des philosophes [75,108], pour lequel on vérifie une propriété d'accessibilité (c'est-à-dire une propriété de la forme $true\ U\ \phi$ pour un ϕ de premier ordre). Comme ce modèle est particulièrement connu, il permet de s'assurer qu'aucun comportement étrange n'apparaît durant le processus de vérification. Le paramètre de ce modèle est le nombre de philosophes qui interagissent entre eux. Le second modèle correspond à des réseaux de capteurs (voir [1,46]). Son intérêt vient du fait qu'il est composé d'un grand nombre de modules de petite taille interagissant entre eux. Pour ce second modèle, on vérifie aussi une propriété d'accessibilité, qui correspond ici à ce qu'un capteur sur le bord de la grille ait bien l'information qu'un feu ait été détecté au milieu de celle-ci. Le paramètre de ce modèle est la taille de la grille de communication. La taille de chemin utilisée correspond à la taille minimale pour laquelle la propriété est satisfaite.

APMC 3.0

La première expérience (figure 6.9) consiste à vérifer qu'APMC 3.0 n'est pas adapté à l'architecture du processeur Cell. Deux critères sont pris en compte : le temps d'exécution (en secondes) et la quantité de mémoire utilisée (en Ko). Nous avons donc utilisé APMC 3.0 sur les deux plateformes considérées. Dans le cas du Cell, ici,

seul le PPE est utilisé. Nous observons très clairement que le Cell est bien plus lent que le Core 2 Duo, ce qui justifie le travail d'implémentation spécifique réalisé avec APMC-CA. En ce qui concerne la mémoire utilisée, les deux plateformes présentent des résultats identiques, ce qui était attendu, mais surtout on remarque qu'on dépasse très rapidement les 256Ko disponibles sur un SPE, et donc que l'économie en mémoire est un point crucial pour l'implémentation sur les SPE du Cell. Ce point était aussi visible lors de l'expérience de la section 6.3.4.

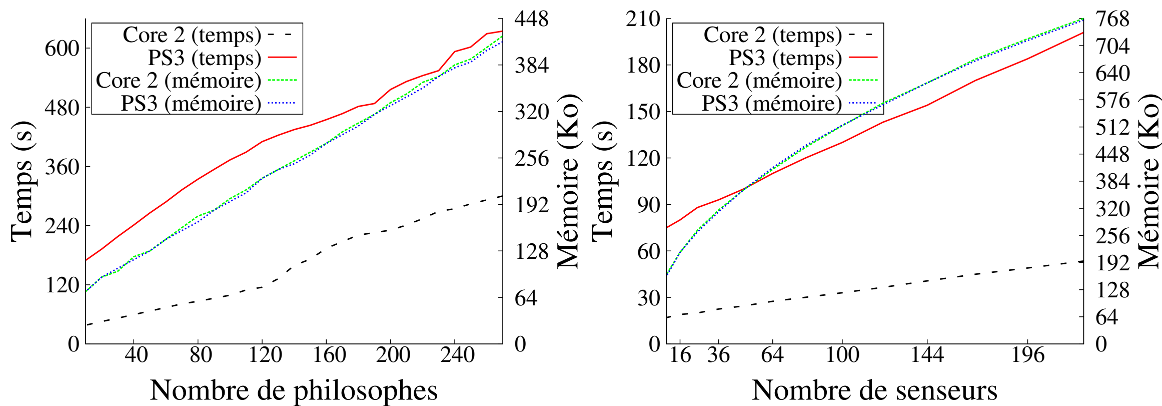

Figure 6.9: Résultats obtenus avec APMC 3.0 pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur processeurs Cell (PS3) et multi-coeur (Core 2).

APMC-CA

La deuxième expérience (figure 6.10) est une comparaison entre les performances d'APMC-CA sur les deux plateformes. Le Cell est ici plus rapide que le Core 2 Duo. Cependant, le Cell est loin d'atteindre les performances que l'on peut retrouver dans le cadre d'autres applications. En effet, le code reste ici intrinsèquement scalaire, alors que l'architecture des SPE repose sur des unités vectorielles. Nous pouvons aussi remarquer que les courbes des deux plateformes ont tendance à se rapprocher quand la taille du modèle augmente (c'est-à-dire le nombre de philosophes ou de capteurs). Ce phénomène vient en partie du fait que la phase de vérification devient plus coûteuse quand la taille des chemins augmente (qui, pour nos expériences, est liée à la taille des modèles). En effet, celle-ci repose principalement sur des branchements conditionnels, qui sont très coûteux sur Cell. Nous pouvons aussi remarquer qu'APMC-CA est plus rapide sur Core 2 Duo qu'APMC 3.0, comme ce processeur tire aussi parti des optimisations réalisées pour le Cell.

Comparaison entre APMC 3.0 et APMC-CA

La troisième expérience (figure 6.11) est une comparaison entre APMC 3.0 sur le Core 2 Duo et APMC-CA sur le Cell. Les performances montrent clairement l'avantage d'utiliser la nouvelle implémentation d'APMC sur la nouvelle plateforme considérée.

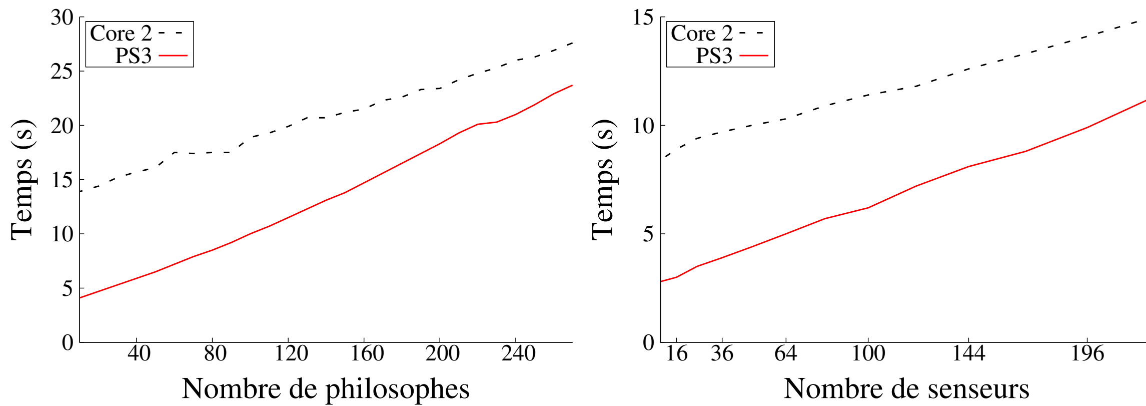

Figure 6.10: Résultats obtenus avec APMC-CA pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite) sur processeurs Cell (PS3) et multi-coeur (Core 2).

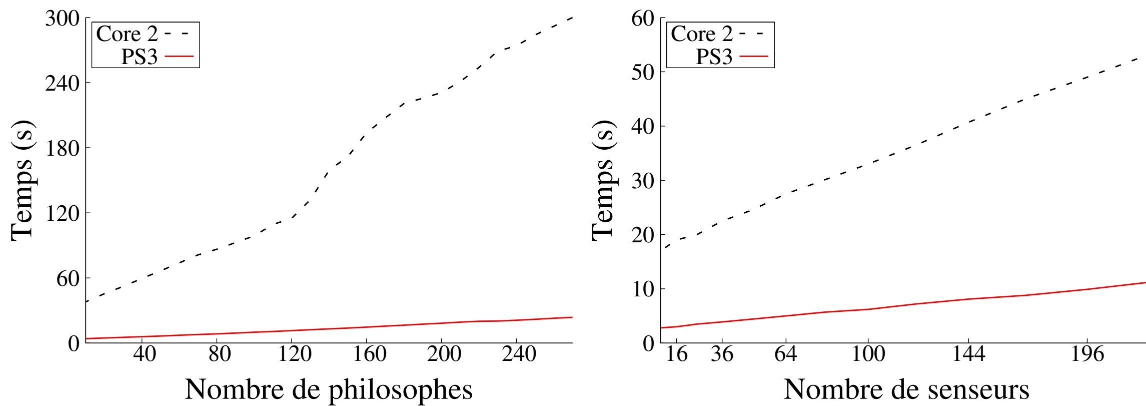

Figure 6.11: Comparaison entre APMC 3.0 sur processeur Core 2 et APMC-CA sur processeur Cell pour le dîner des philosophes (à gauche) et le réseau de capteurs (à droite).

6.5 Conclusion

Dans ce chapitre, nous avons présenté une implémentation parallèle d'APMC qui utilise le modèle *Bulk Synchronous Parallelism* (BSP) et qui permet d'obtenir de très bonnes performances sur trois architectures d'ordinateur parallèles : *clusters*, SMP et *clusters* de SMP (hybride). En particulier, une seule implémentation avec la bibliothèque de haut niveau BSP++ permet au code d'être utilisable efficacement sur ces trois plateformes, ainsi que d'utiliser MPI, OpenMP ou MPI+OpenMP, ce qui est un avantage particulièrement important. Les expériences ont montré les bénéfices de cette approche et valident ainsi l'utilisation d'outils de haut niveau dans le cas d'algorithmes non-triviaux. Finalement, nous avons noté que la version hybride MPI+OpenMP sur architecture hybride (*cluster* de SMP) donne de meilleurs résultats pour APMC. Dans le cadre du Cell, nous avons vu que la nouvelle implémentation avec BSP++ ne permettait que de travailler sur des modèles de très petite taille et que les performances étaient très faibles. C'est pourquoi une nouvelle version d'APMC

spécifique aux caractéristiques du Cell, APMC-CA, a été écrite pour résoudre ces problèmes. L'algorithme en lui-même n'a pas été modifié, mais la représentation des données et la manière d'effectuer les opérations ont été adaptées à cette architecture. Dans un cadre général et hormis le cas très particulier du Cell, l'utilisation d'outils de haut niveau est d'ores et déjà très clairement avantageuse d'un point de vue du développement logiciel, ainsi que des performances.

CHAP. 7

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à l'adaptation de l'algorithmique aux architectures parallèles modernes. Plusieurs approches ont été utilisées, suivant les problèmes et les plateformes considérées.

7.1 Bilan

Tout d'abord, dans le chapitre 3, nous avons effectué la reformulation d'un problème de sorte à obtenir une formulation adaptée aux outils choisis pour la résolution. En l'occurrence, nous avons travaillé sur le problème de la factorisation en nombres premiers, que nous avons reformulé sous forme de programmes linéaires en nombres entiers. Suivant la formulation effectivement utilisée, nous avons pu observer les différents avantages et inconvénients dans le cadre de la résolution avec plusieurs solveurs classiques. Notamment, nous avons vu que la modélisation la plus simple engendre des problèmes de stabilité numérique et ne permet pas de travailler sur des tailles d'instance suffisamment grande. La seconde modélisation corrige ces problèmes, mais est plus lente à résoudre.

Les chapitres 4 et 5 sont consacrés à la résolution du problème du *compressive sensing* (CS). Dans le chapitre 4, nous nous sommes intéressés plus particulièrement à concevoir un algorithme exact et rapide, ce qui permet de comparer la qualité des différentes solutions apportées par les algorithmes approchés. Cette méthode repose sur la programmation linéaire et plus particulièrement sur une modification de la décomposition de Dantzig-Wolfe pour le CS. Cette variante rend possible l'utilisation d'outils développés pour le simplexe, qui ne sont pas performants dans le cadre de la décomposition classique de Dantzig-Wolfe. Dans le chapitre 5, nous avons proposé un algorithme approché pensé dès sa conception pour être parallèle et efficace sur les architectures de processeurs modernes (processeurs multi-coeurs, Cell et GPU). Cette méthode approchée repose sur la programmation convexe et plus particulièrement

sur la régularisation de Moreau-Yosida. Elle a l'avantage d'être très rapide, tout en permettant d'obtenir une solution de très bonne qualité. De plus, une variante exacte de cette méthode, utilisable pour certains types d'entrées, a aussi été présentée et permet d'augmenter les performances, en réduisant le nombre d'itérations.

Enfin, dans le chapitre 6, nous avons utilisé une bibliothèque de haut niveau permettant de générer des versions parallèles reposant sur différents outils standards (OpenMP, MPI ou MPI+OpenMP) et pour différents types de plateformes (*cluster*, SMP et *cluster* de SMP) à partir d'une seule et unique implémentation. Pour cela, l'implémentation, qui est à la charge du développeur, repose sur le modèle BSP. Ici, nous nous sommes appuyés sur une méthode de résolution probabiliste et approchée, nommée APMC, pour un problème de vérification de modèles. Nous avons aussi vu que ce type d'outil et de modèle de haut niveau ne sont pas adaptés aux plateformes atypiques, telles que le Cell et qu'un travail particulier est nécessaire pour obtenir de bonnes performances sur celles-ci. En l'occurrence, nous avons développé APMC-CA, une implémentation d'APMC spécifique pour le processeur Cell. L'algorithme reste inchangé, mais cette implémentation utilise des représentations et effectue des opérations plus adaptées à cette architecture, ce qui permet d'augmenter significativement les performances.

Ainsi, nous avons travaillé dans un cadre haut niveau sur la modélisation d'un problème pour l'adapter aux solveurs, sur la conception de méthodes de résolution exacte et approchée conçues pour être efficaces sur processeurs multi-coeurs, Cell et GPU et finalement sur l'utilisation d'outils logiciels de haut niveau pour la parallélisation automatique sur des architectures de plus grande échelle et plus hiérarchisées, telles que les *clusters* de SMP.

7.2 Perspectives

Parmi les perspectives, nous pouvons citer le travail sur des architectures plus complexes, comme par exemple une plateforme munie de plusieurs GPU. En particulier, il est possible d'adapter l'implémentation effectuée dans le chapitre 5 pour tirer parti d'une telle architecture d'ordinateur. Cependant, les limitations qui existent dans les communications entre les GPU, ainsi qu'entre les GPU et le CPU, et leurs performances, rendent ce travail complexe, étant donné les types de calculs nécessaires à notre méthode, notamment dans le cadre de l'utilisation de transformées rapides.

Il est aussi intéressant d'aborder des plateformes à base de *cluster* d'ordinateurs multi-processeurs et multi-GPU (ou multi-Cell). Nous avons vu que nous pouvons adapter des méthodes préexistantes et concevoir des algorithmes pour obtenir des performances élevées sur les architectures parallèles actuelles. Cependant, bien souvent la nature des outils algorithmiques utilisés permet soit un parallélisme de bas niveau (par exemple la vectorisation et d'autres caractéristiques du Cell et des GPU) soit un parallélisme de haut niveau (par exemple du calcul distribué sur un *cluster*). En effet, il existe très souvent une antinomie entre les méthodes de résolution adaptées au parallélisme de bas niveau, qui demandent une grande bande passante et utilisent des primitives vectorielles, et celles adaptées au parallélisme de haut niveau, qui se contentent très souvent d'une faible bande passante et effectuent

des calculs de nature bien plus disparate. Ainsi, travailler sur des problèmes qui ne sont pas naturellement adaptés au parallélisme de bas *et* de haut niveau mais qui pourraient tout de même tirer parti des deux, en ayant recours aux approches utilisées dans cette thèse serait un très bon sujet d'étude.

De manière plus générale, les plateformes hybrides très hétérogènes proposent de grands défis. En effet, l'attribution des tâches aux différents types de processeurs suivant leurs performances brutes et les performances des communications est particulièrement compliqué. Ceci est d'autant plus compliqué que les quantités de mémoire disponibles au niveau des accélérateurs (GPU, Cell, etc.) sont souvent faibles par rapport à la quantité de mémoire totale. Enfin, les superordinateurs disposent souvent d'architectures très hiérarchisées, ce qui implique des schémas de communication complexes et très fins pour obtenir les meilleurs performances. Il existe une véritable barrière technologique, qui empêche d'utiliser efficacement la puissance des machines à très grande échelle et/ou hétérogènes.

Dans le cadre de cette thèse, nous nous sommes intéressés à l'adéquation de l'algorithmique aux architectures, c'est-à-dire que nous avons utilisé des algorithmes dans un modèle de calcul connu et nous avons cherché à les adapter pour devenir plus efficaces en pratique. Cependant, il n'y a pas de mesure de complexité connue qui reflète les performances pratiques. Trouver un modèle de calcul qui correspond aux architectures parallèles et hétérogènes modernes permettrait de prédire les performances de l'implémentation et constitue un réel challenge.

Annexe

A.1 Résultats complémentaires pour le chapitre 3

Cette section présente des résultats complémentaires pour les expériences de la section 3.4.2.

Le tableau A.1 montre une comparaison en termes de temps d'exécution entre les formulations (F_1') et (F_2') avec CBC et GLPK pour des instances de taille au plus 38 bits, et correspond à la figure 3.2. Rappelons que GLPK ne permet pas de gérer des instances de taille plus grande que 34 bits pour la formulation (F_1').

#bits	8	10	12	14	16	18	20	22
(F_1') CBC time (s)	0,02	0,03	0,04	0,06	0,08	0,16	0,26	0,49
(F_2') CBC time (s)	0,03	0,04	0,30	0,64	2,75	5,86	9,81	32,49
(F_1') GLPK time (s)	0,00	0,01	0,01	0,01	0,02	0,04	0,04	0,09
(F_2') GLPK time (s)	0,01	0,03	0,08	0,15	0,23	0,92	2,90	3,32
#bits	24	26	28	30	32	34	36	38
(F_1') CBC time (s)	0,65	1,60	2,78	3,48	10,84	23,43	37,68	85,82
(F_2') CBC time (s)	41,79	142,68	200,17	248,51	214,81	183,67	80,92	228,25
(F_1') GLPK time (s)	0,14	0,44	0,82	1,12	0,92	0,35		
(F_2') GLPK time (s)	23,63	14,70	30,37	60,62	65,83	50,26	115,39	96,01

Tableau A.1: Comparaison des temps d'exécution de (F_1') et (F_2') avec CBC séquentiel et GLPK sur un Core i7 920. Résultats issus de la moyenne de 10 instances.

Le tableau A.2 montre les temps d'exécution sur les instances de (F_2') avec CBC parallèle et Gurobi parallèle avec 8 *threads*, et correspond à la figure 3.3.

#bits	8	10	12	14	16	18	20	22
//CBC time (s)	0,03	0,04	0,22	0,30	1,04	2,06	4,41	8,21
//Gurobi time (s)	0,01	0,06	0,06	0,09	0,10	0,22	0,44	0,49
#bits	24	26	28	30	32	34	36	38
//CBC time (s)	26,14	88,73	53,80	95,07	90,39	108,285	35,16	40,52
//Gurobi time (s)	1,33	1,87	1,28	2,22	3,14	3,01	3,45	2,54
#bits	40	42	44	46	48	50	52	54
//CBC time (s)	63,49	28,12	61,04	37,97	53,67	92,46	91,96	151,37
//Gurobi time (s)	6,64	6,33	3,78	4,05	3,70	4,03	3,76	3,52
#bits	56	58	60	62	64			
//CBC time (s)	103,85	53,12	87,46	107,82	58,32			
//Gurobi time (s)	3,56	5,37	3,23	4,57	5,77			

Tableau A.2: Performance de (F_2') avec CBC parallèle et Gurobi parallèle (8 *threads*), sur un Core i7 920. Résultats issus de la moyenne de 10 instances.

A.2 Résultats complémentaires pour le chapitre 4

Cette annexe présente des résultats complémentaires pour les expériences de la section 4.4.2.

Le tableau A.3 donne l'ensemble des résultats obtenus pour des matrices de contraintes Gaussiennes orthogonalisées et correspond aux expériences de la section 4.4.2, c'est-à-dire les expériences 4.2 et 4.3.

Le tableau A.4 compare les résultats pour des DCT partielles sous forme de matrices et de transformées rapides.

		Décomposition								Simplex Dual					
		Réduite S		Réduite D		Réduite B		classique DW		Gurobi		CLP		GLPK	
m	n	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time	#iter	time
64	1024	32,7	0,06	116,0	0,04	6832,0	0,17	226089,5	11,72	1881,8	0,38	1356,0	0,72	1336,7	0,79
128	2048	163,9	0,46	432,5	0,23	66361,5	5,39			4086,3	2,99	3049,8	6,66	3006,7	9,46
256	4096	653,3	10,69	1512,9	2,16	544767,7	168,06			9037,0	26,60	7686,8	79,03	7274,4	175,91
512	8192	2330,6	165,79	5226,0	24,69					20357,3	238,99	17619,8	1010,35		
1024	16384			17829,1	439,46										
128	1024	34,7	0,12	178,5	0,11	28102,0	2,06	8047106,0	984,71	2026,2	0,73	1458,1	1,81	1467,4	1,64
256	2048	360,8	3,30	813,7	0,94	361603,5	101,94			4442,8	6,09	3788,9	24,28	3548,8	24,56
512	4096	1499,4	58,05	3214,9	12,06					9975,3	55,35	8887,0	289,00	8493,9	463,18
1024	8192	5387,8	879,57	10797,9	217,35					22493,3	528,13				

Tableau A.3: Temps d’exécution séquentiel et nombre d’itérations avec des matrices Gaussienne orthogonalisées. Résultats issus de la moyenne de 10 instances. Temps en secondes. Les résultats ne sont pas présentés quand le temps d’exécution dépasse 1500 secondes.

m	n	matrice	FFT
64	1024	0,06	0,05
128	2048	0,30	0,28
256	4096	2,88	1,97
512	8192	33,43	17,15
1024	16384	530,16	316,06
128	1024	0,16	0,14
256	2048	1,08	0,92
512	4096	14,65	9,04
1024	8192	243,66	169,12

Tableau A.4: Comparaison du temps d’exécution (en secondes) entre matrices de DCT partielles et FFT pour la décomposition réduite avec la règle de Dantzig. Résultats issus de la moyenne de 10 instances pour différents ratios $\frac{m}{n}$. Les résultats ne sont pas présentés quand le temps d’exécution dépasse 1500 secondes.

Références bibliographiques

[1] S. Peyronnet A. Demaille and B. Sigoure. Modeling of sensor networks using XRM. *Isola*, 0:271–276, 2006. 83, 89

[2] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th international symposium on High performance distributed computing (HPDC)*, pages 165–174, 2008. 14

[3] L. De Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In *Proc. 6th Int. Conf. Tools and Algorithms for Construction and Analysis of Systems*, pages 395–410, 2000. 74

[4] M. Andrecut. Fast GPU implementation of sparse signal recovery from random projections, 2009. 70

[5] A. O. L. Atkin and D. J. Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246):1023–1030, 2004. 23

[6] D. A. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In R. Badrinath S. Aluru, M. Parashar and V. K. Prasanna, editors, *HiPC*, volume 4873 of *Lecture Notes in Computer Science*, pages 172–184. Springer, 2007. 60

[7] W. U. Bajwa, J. D. Haupt, G. M. Raz, S. J. Wright, and R. D. Nowak. Toeplitz-structured compressed sensing matrices. In *Proceedings of the 14th IEEE/SP Workshop on Statistical Signal Processing (SSP)*, pages 294–298, Aug. 2007. 30

[8] C. Barnhart, C. A. Hane, and P. H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Operations Research*, 48(2):318–326, 2000. 31

[9] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998. 31

[10] J. E. Beasley, editor. *Advances in linear and integer programming*. Oxford University Press, Inc., New York, NY, USA, 1996. 19, 22

[11] M. Beran. Decomposable bulk synchronous parallel computers. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pages 349–359, 1999. 81

[12] D. J. Bernstein. The tangent FFT. In Serdar Boztas and Hsiao feng Lu, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 4851 of *Lecture Notes in Computer Science*, pages 291–300. Springer, 2007. 44

[13] Daniel J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. Ecm on graphics cards. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, pages 483–501, Berlin, Heidelberg, 2009. Springer-Verlag. 17

[14] D. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Athena Scientific, 1996. 53

[15] J. Bioucas-Dias and M. Figueiredo. A new TwIST: two-step iterative shrinkage/thresholding algorithms for image restoration. *IEEE Trans. on Image Processing*, 16(12):2992–3004, 2007. 52, 56

[16] R. H. Bisseling and W. F. Mccoll. Scientific computing on bulk synchronous parallel architectures. *Proc. 13th IFIP World Computer Congress*, page 31, 1994. 78

[17] R. E. Bixby. Solving real-world linear programs: A decade and more of progress. *Operations Research*, 50:3–15, 2002. 44

[18] R. G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2:103–107, 1977. 42

[19] P. Bloomfield and W. Steiger. *Least Absolute Deviations: Theory, Applications, and Algorithms*. Birkhauser, 1983. 34

[20] O. Bonorden, B. H. H. Juurlin, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29:187–207, 2003. 79

[21] A. Borghi, J. Darbon, and S. Peyronnet. L1-Compressive Sensing: exact optimization a la Dantzig-Wolfe. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS'10)*, pages 7–12, Oct. 2010. 4, 29

[22] A. Borghi, J. Darbon, and S. Peyronnet. Exact algorithm for the l1-Compressive Sensing problem using a modified Dantzig-Wolfe method. *Theoretical Computer Science*, 412(15):1325–1337, 2011. 4, 29

[23] A. Borghi, J. Darbon, S. Peyronnet, T.F. Chan, and S. Osher. A simple Compressive Sensing algorithm for parallel many-core architectures. Aug. 2009. 51

[24] A. Borghi, J. Darbon, S. Peyronnet, T.F. Chan, and S. Osher. A Compressive Sensing algorithm for many-core architectures. In *Proceedings of the International Symposium on Visual Computing (ISVC 2010)*, pages 678–686, Nov. 2010. 3, 51

[25] A. Borghi, T. Herault, R. Lassaigne, and S. Peyronnet. Cell assisted apmc. In *Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST)*, pages 75–76, Sep. 2008. 4, 73

[26] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. 37

[27] K. Bredies and D. A. Lorenz. Iterated hard shrinkage for minimization problems with sparsity constraints. *SIAM Journal on Scientific Computing*, 30(2):657–683, 2008. 52, 56

[28] Richard P. Brent. Some parallel algorithms for integer factorisation. In *Proc. Third Australian Supercomputer Conference*, 1999. 17

[29] E. Candès and J. Romberg. Quantitative robust uncertainty principles and optimally sparse decompositions. *Foundations of Computational Mathematics*, 6:227–254, 2006. 30

[30] E. Candès, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete frequency information. *IEEE Trans. on Information Theory*, 52(2):489–509, 2006. 30

[31] E. Candès and T. Tao. Near optimal signal recovery from random projections: Universal encoding strategies? *IEEE Trans. on Information Theory*, 52(12):5406–5426, 2006. 30

[32] F. Cappello, F. Desprez, and D. Margery. Grid5000. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home, January 2010. 83

[33] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, 2000. 77

[34] A. Chambolle, R. A. DeVore, N.-Y. Lee, and B. J. Lucier. Nonlinear wavelet image processing: variational problems, compression, and noise removal through wavelet shrinkage. *IEEE Trans. on Image Processing*, 7:319–335, 1998. 52, 56

[35] B. Chapman, B. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 7

[36] S. S. Chen, D. L. Donoho, and M. A. Saunders. Atomic decomposition by basis pursuit. *SIAM Review*, 43(1):129–159, 2001. 29

[37] E. Chow and D. Hysom. Assessing performance of hybrid MPI/OpenMP programs on SMP clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, May 2001. http://www.llnl.gov/CASC/people/chow/chow_pubs.html. 77

[38] M. Cole. *Research Directions in Parallel Functional Programming*, chapter 13, Algorithmic skeletons. Springer, 1999. 78

[39] P. Combettes and J.-C. Pesquet. Proximal thresholding algorithm for minimization over orthonormal bases. *SIAM Journal on Optimization*, 18(4):1351–1376, 2007. 52, 56

[40] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM (JACM)*, 42(4):857–907, 1995. 74

[41] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan-Mar 1998. 7, 60, 73

[42] G. B. Dantzig. Programming in a linear structure. USAF, Washington D.C., 1948. 42

[43] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, 1963. 30, 34, 37, 41

[44] G. B. Dantzig and P. Wolfe. The decomposition algorithm for linear programming. *Econometrica*, 9(4), 1961. 30, 31, 32, 33

[45] I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications in Pure and Applied Mathematics*, 57(11):1413–1457, 2004. 52, 56

[46] A. Demaille, S. Peyronnet, and B. Sigoure. Modeling of sensor networks using XRM. In *ISoLA*, pages 271–276. IEEE, 2006. 83, 89

[47] R. A. DeVore. Deterministic constructions of compressed sensing matrices. *Journal of Complexity*, 4–6(23):918–925, 2007. 30

[48] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scale. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, March/April 2000. 6, 60

[49] D. Donoho, Y. Tsaig, I. Drori, and J.-L. Starck. Sparse solution of underdetermined linear equations by stagewise orthogonal matching pursuit. Technical report, 2006. 52

[50] D. L. Donoho. Compressed sensing. *IEEE Trans. on Information Theory*, 52(4):1289–1306, 2006. 30

[51] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. *Parallel and Distributed Processing Symposium, International*, 1:15a, 2004. 77

[52] F.-X. Dupe, J. Fadili, and J.-L. Starck. A proximal iteration for deconvolving Poisson noisy images using sparse representations. *IEEE Trans. on Image Processing*, 16(12):2992–3004, 2008. 52

[53] B. P. Dzielinski and R. E. Gomory. Optimal programming of lot sizes, inventory and labor allocations. *Management Science*, 11(9):874–890, 1965. 31

[54] W. Yin E. T. Hale and Y. Zhang. A fixed-point continuation method for l1-regularized minimization with applications to compressed sensing. Technical report, Rice University, 2007. 52, 56

[55] M. Elad. Why simple shrinkage is still relevant for redundant representation? *IEEE Trans. on Information Theory*, 52:5559–5569, 2006. 52, 56

[56] G. De Fabritiis. Performance of the Cell processor for biomolecular simulations. *Computer Physics Communications*, 176(11–12):660–664, 2007. 9

[57] M. Figueiredo and R. Nowak. An EM algorithm for wavelet-based image restoration. *IEEE Trans. on Image Processing*, 12(8):906–916, 2003. 52, 56

[58] M. Figueiredo, R. Nowak, and S. Wright. Gradient projection for sparse reconstruction: application to compressed sensing and other inverse problems. *IEEE Journal of Selected Topics in Signal Processing*, 1(3):586–598, 2007. 52

[59] J. J. H. Forrest and D. Goldfarb. Steepest-edge simplex algorithms for linear programming. *Mathematical Programming*, 57:341–374, 1992. 42

[60] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Application*, 8:165–416, 1994. 7, 73

[61] M. P. Friedlander and M. A. Saunders. Discussion: the Dantzig selector: statistical estimation when p is much larger than n. *Annals of Statistics*, 35(6):2385–2391, December 2007. 45

[62] M. Frigo and S.G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, May 1998. 45, 60

[63] S. I. Gass and S. Vinjamuri. Cycling in linear programming problems. *Computers and Operations Research*, 31(2):303–311, 2004. 37

[64] L. Gesbert and F. Gava. New syntax of a high-level BSP language with application to parallel pattern-matching and exception handling. Technical Report TR-LACL-2009-06, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12), 2009. 79

[65] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. Bulk synchronous parallel ML with exceptions. *Future Generation Computer Systems*, 26:486–490, 2010. 79

[66] L. Giraud. Combining shared and distributed memory programming models on clusters of symmetric multiprocessors: Some basic promising experiments. *International Journal of High Performance Computing Applications*, 16:425–430, 2002. 77

[67] D. Goldfarb and J. Reid. A practicable steepest edge simplex algorithm. *Mathematical Programming*, 12:361–371, 1977. 42

[68] T. Goldstein and S. Osher. The split Bregman method for l1 regularized problems. Technical Report CAM 08-29, UCLA, 2008. 52

[69] J. A Gonzalez, C. Leon, F. Piccoli, M. Printista, J. L. Roda, C. Rodríguez, and F. de Sande. Oblivious BSP. In *Euro-Par'00 : Proceedings of the 6th International EURO-PAR Conference*, volume 1900, pages 682–685, 2000. 79

[70] R. Griesse and D. A. Lorenz. A semismooth Newton method for Tikhonov functionals with sparsity constraints. *Inverse Problems*, 24(3), 2008. 52

[71] K. Hamidouche, A. Borghi, P. Esterie, J. Falcou, and S. Peyronnet. Three high performance architectures in the parallel approximate probabilistic model checking boat. In *Proceedings of the 9th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC)*, Sep. 2010. 4, 73

[72] K. Hamidouche, J. Falcou, and D. Etiemble. Hybrid bulk synchronous parallelism library for clustered smp architectures. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 55–62, New York, NY, USA, 2010. ACM. 73, 77, 79

[73] P. M. J. Harris. Pivot selection methods of the Devex LP code. *Mathematical Programming*, 5(1):1–28, 1973. 42

[74] J. Hastad. On using RSA with low exponent in a public key network. In *Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 403–408, New York, NY, USA, 1986. Springer-Verlag New York, Inc. 16

[75] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proceedings of the 5th Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 73–84, 2004. 74, 83, 89

[76] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24:1947–1980, 1998. 79

[77] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006. 75, 76

[78] J.-B. Hiriart-Urruty and C. Lemaréchal. *Convex Analysis and Minimization Algorithms*. Springer Verlag, Heidelberg, 1996. Two volumes - 2nd printing. 52, 53, 54, 55

[79] S. Horie and O. Watanabe. Hard instance generation for SAT. In *ISAAC '97: Proceedings of the 8th International Symposium on Algorithms and Computation*, pages 22–31, London, UK, 1997. Springer-Verlag. 17

[80] C. J. Noble I. J. Bush and R. J. Allan. Mixed OpenMP and for MPI parallel fortran applications. *In European workshop on OpenMP (EWOMP 2000)*, Edinburgh, UK, 2000. 77

[81] R. M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64. IEEE, 1983. 75

[82] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master's thesis, Department of Mathematics, University of Chicago, Chicago, IL, USA, 1939. 37

[83] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky. An interior-point method for large-scale l_1-regularized least squares. *IEEE Journal of Selected Topics in Signal Processing*, 1(4):606–617, 2007. 52

[84] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. te Riel, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit rsa modulus. Cryptology ePrint Archive, Report 2010/006, 2010. 16

[85] G. Krawezik and F. Cappello. Performance comparaison of MPI and three OpenMP programming styles on shared memory multiprocessors. *ACM Symposium on Parallel Algorithms*, pages 118–127, 2003. 77, 81

[86] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. Berkeley, California, 1951. 37

[87] A. Kumar, G. Senthilkumar, M. Krishna, N. Jayam, P. K. Baruah, R. Sharma, A. Srinivasan, and S. Kapoor. A buffered-mode MPI implementation for the cell betm processor. In *Proceedings of the 7th international conference on Computational Science, Part I: ICCS 2007*, ICCS'07, pages 603–610, Berlin, Heidelberg, 2007. Springer-Verlag. 9

[88] J. Kurzak and J. Dongarra. Implementation of mixed-precision in solving systems of linear equations on the CELL processor. *Concurrency and Computation: Practice and Experience*, 19(10):1371–1385, 2007. 8

[89] L. S. Lasdon. *Optimization Theory for Large Systems*. The Macmillan Company, New York, 1970. 30, 31, 37

[90] R. Lassaigne and S. Peyronnet. Probabilistic verification and approximation. *Annals of Pure and Applied Logic*, 152(1-3):122–131, 2008. 4, 74

[91] C. Lemaréchal and C. Sagastizábal. Practical aspects of the Moreau-Yosida regularization: Theoretical preliminaries. *SIAM Journal on Optimization*, 7(2):367–385, 1997. 52, 53, 54, 55

[92] A. K. Lenstra and H. W. Lenstra Jr., editors. *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1993. 16

[93] H. W. Lenstra. Factoring integers with elliptic curves. *The Annals of Mathematics*, 126(3):649–673, November 1987. 16

[94] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1111–1120, 2008. 61

[95] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam, G. Dick van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (ICCS 2005), Part II*, volume 3515 of *LNCS*, pages 1046–1054. Springer-Verlag, 2005. 81

[96] G. Mahinthaumar and F. Said. A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. *International Journal of High Performance Computing Applications*, 16(4):371–393, 2002. 77

[97] G. McCormick. Computability of global solutions to factorable nonconvex programs: Part I - Convex underestimating problems. 10:146– 175, 1976. 18, 20

[98] J Milan. Factoring Small Integers: An Experimental Comparison. 16

[99] J.J. Moreau. Proximité et dualité dans un espace hilbertien. *Bulletin de la S.M.F.*, 93:273–299, 1965. 52, 53

[100] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. 9

[101] M. Nikolova. Markovian reconstruction using a GNC approach. *IEEE Trans. on Image Processing*, 8(9):pp. 1204–1220, 1999. 56

[102] M. Nikolova, J. Idier, and A. Mohammad-Djafari. Inversion of large-support ill-posed linear operators using a piecewise Gaussian MRF. *IEEE Trans. on Image Processing*, 8(4):571–585, 1998. 56

[103] M. Nikolova, M. K. Ng, S. Q. Zhang, and W. K. Ching. Efficient reconstruction of piecewise constant images using nonsmooth nonconvex minimization. *SIAM Journal on Imaging Sciences*, 1(1):2–25, 2008. 56

[104] R. Nowak and M. Figueiredo. Fast wavelet-based image deconvolution using the em algorithm. In *Proceedings of the 35th Asilomar Conference on Signals, Systems, and Computers*, pages 371–275, 2001. 52, 56

[105] K. O'Brien, K. M. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008. 9

[106] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. 9

[107] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *Proceedings of the Solid-State Circuits Conference*, pages 184–185, 2005. 7

[108] A. Pnueli and L. D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986. 83, 89

[109] D. Goswami S. Siu, M. De Simone and A. Singh. Design patterns for parallel programming. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 230–240, 1996. 77

[110] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998. 77

[111] S. Peyronnet T. Hérault, R. Lassaigne. Apmc 3.0: Approximate verification of discrete and continuous time markov chains. In *QEST'06*, pages 129–130, 2006. 77

[112] T.-H. Tcheng. *Scheduling of a Large Forestry-Cutting Problem by Linear Programming Decomposition*. PhD thesis, University of Iowa, 1966. 31

[113] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B*, 58:267–288, 2006. 52

[114] J. A. Tropp. Greed is good: algorithmic results for sparse approximation. *IEEE Trans. on Information Theory*, 50(10):2231–2242, 2004. 52

[115] J. A. Tropp. Just relax: Convex programming methods for identifying sparse signals. *IEEE Trans. on Information Theory*, 51(3):1030–1051, 2006. 30, 52

[116] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990. 78

[117] F. Vanderbeck. On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, 48(1):111–128, 2000. 31

[118] M. Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 327–338, Washington, DC, USA, 1985. IEEE Computer Society. 74

[119] A. J. Wallcraft. SPMD OpenMP versus MPI for ocean models. *concurrency: Practice and Experience*, 12:1155–1164, 2000. 77

[120] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. *IEEE International Parallel and Distributed Processing Symposium*, pages 1–14, 2008. 64

[121] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proceedings of the 3rd conference on Computing frontiers (CF)*, pages 9–20, New York, NY, USA, 2006. ACM. 9

[122] T. L. Williams and R. J. Parsons. The heterogeneous bulk synchronous parallel model. *Workshop on Advances in Parallel and Distributed Computational Models*, 1:102–108, 2000. 79

[123] L. T. Yang, L. Xu, and M. Lin. Integer factorization by a parallel gnfs algorithm for public key cryptosystems. In L. T. Yang, X. Zhou, W. Zhao, Z. Wu, Y. Zhu, and M. Lin, editors, *ICESS*, volume 3820 of *Lecture Notes in Computer Science*, pages 683–695. Springer, 2005. 16

[124] L. T. Yang, L. Xu, S.-S. Yeo, and S. Hussain. An integrated parallel GNFS algorithm for integer factorization based on Linbox Montgomery block Lanczos method over GF(2). *Comput. Math. Appl.*, 60:338–346, July 2010. 16

[125] W. Yin, S. Osher, D. Goldfarb, and J. Darbon. Bregman iterative algorithms for l1-minimization with applications to compressed sensing. *SIAM Journal on Imaging Sciences*, 1(1):143–168, 2008. 52

[126] K. Yosida. *Functional Analysis*. Springer, 1965. 52

[127] H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Proceedings of CAV*, volume 2404 of *LNCS*, pages 223–235. Springer, 2002. 74