Contribution à l'algorithmique et à la programmation efficace des nouvelles architectures parallèles comportant des accélérateurs de calcul dans le domaine de la neutronique et de la radioprotection

Dubois Jérôme

THÈSE

Présentée à

L'UNIVERSITE LILLE 1 SCIENCES ET TECHNOLOGIES

Pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

Par

Jérôme DUBOIS

« Contribution à l'algorithmique et à la programmation efficace des nouvelles architectures parallèles comportant des accélérateurs de calcul dans le domaine de la neutronique et de la radioprotection»

Thèse de l'Ecole Doctorale – Sciences pour l'Ingénieur (EDSPI), effectuée avec le Laboratoire d'Informatique Fondamentale de Lille (LIFL) et le Commissariat à l'Energie Atomique et aux énergies alternatives (CEA).

Thèse soutenue le 13 Octobre 2011, devant la commission d'examen :

Président	:	Clarisse DHAENENS	Professeur à l'Université Lille 1 Sciences et Technologies
Rapporteurs	:	Michel DAYDÉ	Professeur à l'ENSEEIHT
		Marc SNIR	Professeur au Department of Computer Science, University of Illinois
Examinateurs	:	Anthony DRUMMOND	Docteur au Lawrence Berkeley National Laboratory
		Christophe CALVIN	Docteur au Commissariat à l'Énergie Atomique et aux Énergies Alternatives
Directeur de thèse	:	Serge Petiton	Professeur à l'Université Lille 1 Sciences et Technologies

Numéro d'ordre : 40610

Remerciements

En tout premier lieu, je remercie M. Serge PETITON, Professeur à l'Université de Lille 1 Sciences et Technologies. Sous sa direction, ces travaux ont pour moi été très formateurs et m'ont permis une initiation à la Recherche stimulante, ainsi qu'une acquisition de connaissances passionnante. Ses conseils, son expérience et ses idées m'ont permis d'acquérir beaucoup durant ces trois années.

Je remercie particulièrement M. Christophe CALVIN, mon encadrant CEA, pour sa patience et son implication. Ces travaux se placent dans un contexte à la frontière de la Recherche et l'Industrie, et ses remarques pertinentes ainsi que ses conseils m'ont toujours permis de lier mes travaux de recherche à un aspect pratique et de faire avancer ma réflexion.

Je remercie également les rapporteurs, M. Michel DAYDE, Professeur à l'ENSEEIHT et M. Marc SNIR, Professeur à l'Université de l'Illinois, d'avoir accepté d'être rapporteurs de ma thèse et pour les échanges constructifs en ayant résulté ainsi que l'intérêt qu'ils ont porté à mon travail.

Je tiens à remercier Mme Clarisse DHAENENS, Professeur à Polytech'Lille et au LIFL, d'avoir accepté de présider le jury de ma thèse et pour son aide dans l'organisation de la soutenance.

Je remercie M. Tony DRUMMOND du Lawrence Berkeley National Laboratory, d'avoir accepté de faire partie de mon jury et de l'intérêt qu'il a porté à mes travaux de Recherche.

Mes remerciements vont à Anne-Marie BAUDRON, Erell JAMELOT, Jean-Jacques LAUTARD et Stéphane SALMONS, chercheurs au Laboratoire des Logiciels pour la Physique des Réacteurs, mon laboratoire d'accueil au CEA. L'aide liée à leurs spécialités m'aura permis de découvrir et apprendre de nouvelles connaissances en neutronique et génie logiciel et d'enrichir ou améliorer mes travaux.

Enfin, mes plus tendres remerciements vont à Mme Blandine DUBOIS, mon épouse, pour son support continu durant mon initiation à la Recherche, ainsi que sa patience infinie.

Résumé

Dans le domaine des sciences et technologies, la simulation numérique est un élément-clé des processus de recherche ou de validation. Grâce aux moyens informatiques modernes, elle permet des expérimentations numériques rapides et moins coûteuses que des maquettes réelles, sans pour autant les remplacer totalement, mais permettant de réaliser des expérimentations mieux calibrées. Dans le domaine de la physique des réacteurs et plus précisément de la neutronique, le calcul de valeurs propres est la base de la résolution de l'équation du transport des neutrons. La complexité des problèmes à résoudre (dimension spatiale, énergétique, nombre de paramètres, …) est telle qu'une grande puissance de calcul peut être nécessaire.

Les travaux de cette thèse concernent dans un premier temps l'évaluation des nouveaux matériels de calculs tels que les cartes graphiques ou les puces massivement multicoeurs, et leur application aux problèmes de valeurs propres pour la neutronique. Ensuite, dans le but d'utiliser le parallélisme massif des supercalculateurs, nous étudions également l'utilisation de méthodes hybrides asynchrones pour résoudre des problèmes à valeur propre avec ce très haut niveau de parallélisme. Nous expérimentons ensuite le résultat de ces recherches sur plusieurs supercalculateurs nationaux tels que la machine hybride Titane du Centre de Calcul, Recherche et Technologies (CCRT), la machine Curie du Très Grand Centre de Calcul (TGCC) qui est en cours d'installation, et la machine Hopper du Lawrence Berkeley National Laboratory (LBNL), mais également sur des stations de travail locales pour illustrer l'intérêt de ces recherches dans une utilisation quotidienne avec des moyens de calcul locaux.

Abstract

In science, simulation is a key process for research or validation. Modern computer technology allows faster numerical experiments, which are cheaper than real models. In the field of neutron simulation, the calculation of eigenvalues is one of the key challenges. The complexity of these problems is such that a lot of computing power may be necessary.

The work of this thesis is first the evaluation of new computing hardware such as graphics card or massively multicore chips, and their application to eigenvalue problems for neutron simulation. Then, in order to address the massive parallelism of supercomputers national, we also study the use of asynchronous hybrid methods for solving eigenvalue problems with this very high level of parallelism.

Then we experiment the work of this research on several national supercomputers such as the Titane hybrid machine of the Computing Center, Research and Technology (CCRT), the Curie machine of the Very Large Computing Centre (TGCC), currently being installed, and the Hopper machine at the Lawrence Berkeley National Laboratory (LBNL). We also do our experiments on local workstations to illustrate the interest of this research in an everyday use with local computing resources.

Sommaire

Introduction

Dans le domaine de la physique des réacteurs, il existe différentes problématiques qui contribuent toutes à la modélisation du comportement d'un cœur de réacteur nucléaire telles que la thermo-dynamique, la thermo-hydraulique, la chimie ou encore la modélisation des flux de neutrons (neutronique). Ces modélisations sont basées sur différentes équations (Navier-Stokes, Boltzmann, Bateman, …) qui elles-mêmes sont résolues grâce à différents solveurs itératifs qui calculent l'état du cœur de réacteur de manière couplée ou non. Ces solveurs peuvent demander une puissance de calcul très élevée dans le cadre de calculs de cœurs complets très fins ou pour l'évaluation extrêmement rapide de nombreuses solutions en fonction de très nombreux paramètres.

La simulation numérique joue un rôle prépondérant pour répondre aux défis d'amélioration et de recherche sur la physique des réacteurs. On peut considérer différentes cibles d'utilisation pour le calcul haute performance dans ce domaine:

- calculs paramétrés: c'est la technique de base pour l'optimisation. Le calcul hautes performances est une excellente occasion de prendre en compte plusieurs paramètres et de réduire les temps de mise sur le marché. Ceci permet l'utilisation de techniques d'optimisation, comme les réseaux de neurones, afin de trouver automatiquement et optimiser un ensemble de paramètres.
- Physique Haute résolution: une plus grande capacité de mémoire et une plus grande puissance du CPU sont requis pour des modèles plus raffinés dans chaque discipline. Pour le transport déterministe par exemple, au lieu de l'approche de la diffusion à peu de groupes ;
- Une physique plus réaliste en utilisant systématiquement le modèle physique réel au lieu du modèle simplifié ou des valeurs tabulées. Cela implique non seulement une plus grande puissance de processeur, mais aussi une utilisation robuste et un couplage facile des systèmes
- La simulation en temps réel : cela existe déjà, mais nous pouvons imaginer une amélioration de la modélisation afin d'obtenir des simulateurs plus réalistes et diminuer le nombre d'approximations.

Toutes ces améliorations sont nécessaires afin de pouvoir:

- augmenter la marge de sécurité en réduisant les incertitudes ;
- Optimiser les conceptions ;
- Améliorer la sécurité ;
- Optimiser les conditions de fonctionnement ;
- Accroître les connaissances en physique.

Ces défis peuvent être relevés en partie grâce aux puissances de calcul de plus en plus importantes et à tous les niveaux. En effet, les capacités de calcul sont présentes à petite échelle dans les stations de travail utilisant plusieurs processeurs multi-cœurs et des accélérateurs graphiques, à moyenne échelle dans les clusters ou grappes de calculs présents au niveau du laboratoire ou service de recherche, à plus grande échelle au niveau des mésocentres de calculs régionaux, et à très grande échelle au sein des supercalculateurs nationaux ou des grilles de calcul nationales ou mondiales. De plus, une mutation du calcul hautes performances a introduit de nouveaux matériels de calcul ayant un impact fort sur certains domaines jusque-là hors de la portée d'une station de travail classique : les matériels accélérateurs tels que les cartes graphiques. Il est ainsi possible de

placer plusieurs cartes dans une seule station de travail, et potentiellement obtenir la puissance d'un cluster de taille moyenne (une centaine de cœurs), pour un coût moindre et un encombrement réduit. C'est le concept du « supercalculateur dans une station de travail ». Ces matériels introduisent des concepts nouveaux dans la manière de développer les algorithmes servant à la simulation numérique en général.

Un point clef de la simulation neutronique est la détermination des flux de neutrons dans le cœur du réacteur. Cette détermination passe par la résolution exacte ou approchée de l'équation du transport des neutrons (équation de Boltzmann). Parmi les différentes étapes de la résolution de l'équation, la résolution d'un problème à valeur propre dominante fait partie de celles qui demandent le plus de temps de calcul.

Dans le cadre de cette thèse, nous nous intéressons à l'étude et l'optimisation de méthodes numériques parallèles pour résoudre un problème à valeur propre en utilisant les sous-espaces de Krylov. Ces méthodes sont implémentées pour différentes configurations de calculateurs parallèles (multi-cœurs, avec accélérateurs, …). Pour ces travaux, une maquette sera mise en place avec pour objectif l'intégration au sein du code de physique des réacteurs APOLLO3® du Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA). Le matériel évoluant rapidement, au niveau des accélérateurs notamment, nous cherchons à étudier l'applicabilité de ces matériels pour le code APOLLO3® tant au niveau performances qu'au niveau programmabilité et maintenabilité.

Pour mener cette étude, le plan de la thèse s'organise suivant les chapitres suivants. Le premier chapitre rappellera les évolutions récentes du matériel de calcul ainsi que quelques paradigmes pour adresser ce parallélisme et une analyse des contraintes sur les performances en environnement parallèle. Le chapitre suivant présentera la problématique de la résolution de problèmes à valeur propre ainsi que différentes manières d'effectuer la parallélisation de cette résolution. Des bibliothèques d'algèbre linéaire seront notamment présentées. Le troisième chapitre se concentrera sur l'étude de la précision des accélérateurs Cell et des cartes graphiques Nvidia pour les problématiques d'orthogonalisation dans le cadre de projection dans des sous-espaces de Krylov. Nous montrerons ainsi l'applicabilité de ces matériels pour le calcul de valeurs propres utilisant des sous-espaces de Krylov. Le chapitre 4 décrira la création du framework parallèle adressant les multicoeurs et accélérateurs avec le même algorithme commun, et des expérimentations de la méthode d'Arnoldi explicitement redémarrée (ERAM) sur des stations de travail et des supercalculateurs européens, américains, et français. Nous verrons en particulier la limite de scalabilité intrinsèque à cette méthode lors de l'exploitation de plusieurs milliers de cœurs de calculs. Dans le chapitre 5, nous verrons l'optimisation de la méthode hybride MERAM utilisant plusieurs solveurs ERAM pour obtenir une accélération numérique et augmenter la scalabilité tout en ayant des capacités intrinsèques de tolérance aux pannes. Les expérimentations sur différents supercalculateurs montreront l'accélération numérique de la méthode ainsi que la bonne résistance de MERAM aux pannes matérielles simulées par l'arrêt d'un ou plusieurs solveurs. Nous présenterons également quelques pistes intéressantes qui permettraient à terme une utilisation très efficace de processeurs standards et d'accélérateurs pour la résolution de problèmes à valeurs propres tant au niveau d'une station de travail hybride qu'à celui d'un supercalculateur. Le dernier chapitre présentera l'application des travaux de cette thèse au sein des solveurs de transport SPn MINOS et Sn MINARET du code APOLLO3® du CEA, avec des expérimentations sur le supercalculateur hybride Titane où nous adressons plusieurs dizaines d'accélérateurs.

Chapitre 1. Rappels sur le parallélisme

Introduction

Ce chapitre propose de présenter une vue générale de la zoologie du calcul hautes performances. L'évolution des architectures de calcul et des langages de programmation font partie des motivations profondes de cette thèse, avec une application à la neutronique dans le cadre de codes de calcul développés au Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA). Le paysage a fortement évolué depuis le début des années 2000, et 2010 pourrait être vue comme l'année du début de l'ère des accélérateurs. Nous tenterons rapidement de voir pourquoi et comment cette évolution s'est produite. Parallèlement, nous introduirons les notions clés à comprendre pour appréhender le calcul hautes performances. En particulier, les problématiques telles que le coût de la communication, la portabilité et la maintenance des solveurs numériques sont des points clés autant dans un cadre autant industriel qu'académique.

1.1. État de l'art du calcul haute performance

La notion de calcul hautes performances possède plusieurs définitions, et celle de « science des superordinateurs » est plutôt intéressante. Elle reflète de l'utilisation de machines appelées superordinateurs, avec des connaissances nécessaires pour leur utilisation effective. Ces connaissances sont une sous-partie du domaine du calcul scientifique, une discipline très variée (1). Ce que désigne le terme *superordinateur* varie avec le temps car les ordinateurs les plus puissants du monde à un moment donné tendent à être égalés puis dépassés par des machines d'utilisation courante des générations suivantes, comme nous le verrons par la suite.

Les superordinateurs sont utilisés pour toutes les tâches qui nécessitent une très forte puissance de calcul comme les prévisions météorologiques, l'étude du climat, la modélisation moléculaire (calcul des structures et propriétés de composés chimiques…), les simulations physiques (simulations aérodynamiques, calculs de résistance des matériaux, simulation d'explosion d'arme nucléaire, étude de la fission/fusion nucléaire…), la cryptanalyse, etc.

Ces machines tirent leur supériorité grâce bien souvent à un matériel dédié optimisé pour le calcul, tel que des réseaux spécifiques réglés pour le système de calcul, des systèmes de stockage répartis et un système d'exploitation optimisé pour la machine.

1.1.1. Rappels historiques

Les premiers superordinateurs étaient constitués de processeur à un seul cœur, aux capacités de calculs relativement limitées par rapport à ce que nous permet la technologie actuelle. Par la suite, dans les années 1970, l'architecture vectorielle a été adoptée massivement. Cette dernière proposait d'effectuer une seule instruction sur plusieurs opérandes. L'un des objectifs étaient de recouvrir les accès mémoires en effectuant un grand nombre d'opérations identiques et régulières. Vers la fin des années 1980, la plupart des superordinateurs ont progressivement migré vers une architecture massivement parallèle à base de processeurs scalaires ou superscalaires, que l'on retrouve encore aujourd'hui : l'utilisation de processeurs dits de commodité à grande échelle. Les économies réalisées sont importantes, et permettent d'atteindre une performance globale satisfaisante. Concernant les échelles, le tableau suivant résume les performances crêtes atteignables avec les superordinateurs d'époque différentes. Les unités employées sont successivement le Flops, qui représente le nombre d'opérations flottantes par secondes, puis les Kilo, Mega, Giga, Tera et PetaFlops, soit respectivement 10^3, 10^6, 10^9, 10^{12} et 10^{15} Flops.

Nom	Z1	ENIAC	SAGE	CDC 7600	Cyber-205	CM-5	Earth Simulator	K computer
Année	1938	1946	1958	1969	1981	1993	2002	2011
performance	1	50k	400k	36M	400M	59.7G	35.86T	8.16P
Facteur génération précédente	-	50000x	8x	90x	11x	150x	601x	228x

Cette vue générale serait à compléter avec l'apparition de matériels spécifiques tels que les Connection Machine, où des travaux s'approchant de ceux du chapitre 4 ont été réalisés (2).

1.1.2. Évolution récente du matériel

Si l'on s'intéresse à l'évolution du matériel depuis la moitié des années 2000, on voit une augmentation significative du nombre de cœurs dans les machines de bureaux, alors que la fréquence d'horloge tend à se stabiliser voir même se réduit. La raison directe derrière cette tendance est un phénomène appelé « Power Wall », soit la limite de consommation en énergie et indirectement de dissipation de chaleur. La figure 1 illustre l'évolution de la dissipation thermique d'un processeur, qui au centimètre carré s'approche de celle de la surface du soleil.

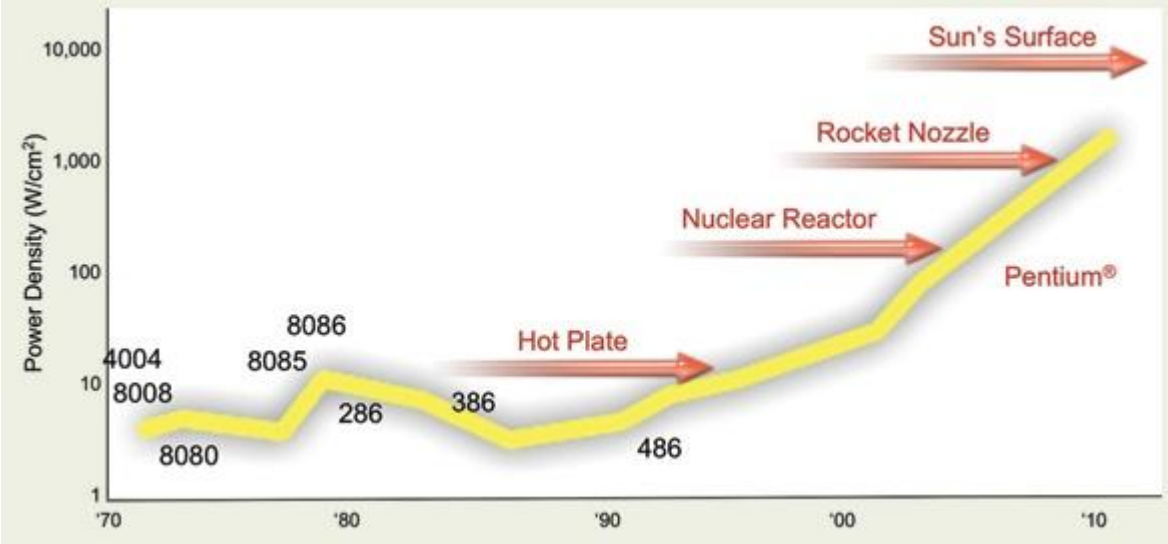


Figure 1. Évolution de la dissipation thermique au sein des processeurs Intel.

Cette consommation est en partie liée à l'augmentation de la fréquence, qui induit souvent une augmentation du voltage, et également du nombre de transistors. Ces paramètres encourageant la consommation sont contrebalancés par la finesse de gravure, qui permet de diminuer les pertes d'énergie par rapport à la finesse de gravure précédente. En 1965, Gordon Moore établissait que le nombre de transistors d'un microprocesseur devrait doubler tous les 12 mois. En 1975, ce cycle a été corrigé à 24 mois. De nos jours, cette augmentation tend à se produire tous les 18 mois. La figure 2 illustre plusieurs tendances des processeurs Intel.

La courbe verte confirme la loi de Moore, avec une augmentation régulière du nombre de transistor. Parallèlement, les autres courbes, liées directement ou indirectement à la fréquence d'horloge du processeur, augmentent presque aussi régulièrement jusqu'aux environs de l'année 2003. Par la suite, elles se stabilisent, quand bien même le nombre de transistors continue d'augmenter… nous sommes entrés à cette date dans l'ère du multicoeur. A titre d'exemple, le représentant Intel de l'ère monocoeur le plus rapide était un Pentium4 671 fonctionnant à une fréquence de base de 3.8 GHz, fréquence de base qui depuis n'a pas été dépassée depuis par les processeurs grand public. Dans les évolutions récentes des processeurs multicoeurs, certains proposent des technologies de surcadençage (overclocking), qui permettent de désactiver certains cœurs, et de fortement augmenter la fréquence d'autres. Actuellement, le record grand public de ce type de technologie est détenu par Intel, avec une augmentation de 3.33 GHz à 3.9GHz sur certains corei7. Cette technologie utilise l'enveloppe thermique maximum supportée par le processeur, de l'ordre de 130 Watts. Encore une fois, le « Power Wall » limite l'augmentation en fréquence.

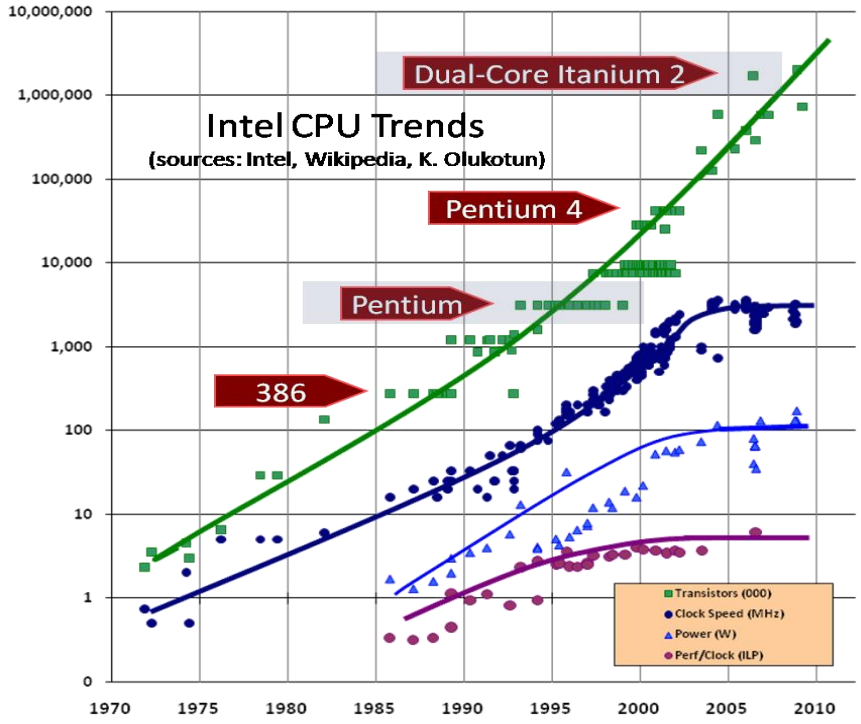


Figure 2. Tendances matérielles des CPUs Intel, depuis 1970.

D'où l'intérêt de profiter de l'amélioration des processus de gravure pour ajouter des cœurs d'exécutions fonctionnant à une vitesse acceptable, tout en délivrant une performance suivant la loi de Moore. La figure 3 montre les statistiques de 1970 à 2011, puis plusieurs projections de l'évolution du nombre de cœurs au sein de nos machines. De 1978 à 2003, les processeurs grand public étaient monocoeur. En 2004, les premiers processeurs dual-core sont apparus, avec des performances parallèles réduites dues à l'utilisation d'un bus mémoire externe aux deux cœurs pour les faires communiquer (bus mémoire de la carte mère). En 2005, les premiers vrais processeurs dual-cœur avec bus mémoire interne ont permis une performance parallèle acceptable, et par la suite le nombre de cœurs a régulièrement augmenté jusqu'en 2011, où sont attendus les premiers processeurs à 16 cœurs. De manière intéressante, la fréquence d'horloge et donc la performance ont augmenté d'un facteur 1.37 en moyenne chaque année, de 1 MHz en 1978 à 3.8 GHz en 2003. Ce facteur a été conservé par la suite, mais dans le parallélisme des cœurs, avec une augmentation de 1 cœur en 2003 à 16 cœurs en 2011, pour un facteur annuel de 1.38x. Les courbes de tendance utilisées sur cette figure nous montrent un nombre de cœurs projeté dans 10 ans variant entre 70 à plus de 200, au sein d'un seul processeur. Si l'on utilise naïvement la règle du facteur 1.38x annuel soutenu pour l'augmentation du nombre de cœurs à partir de 2011, alors la projection nous amène à $16 * 1.38^9 = 290$ cœurs. Parallèlement, la fréquence est stable ou en légère baisse comme nous avons pu le voir avec l'exemple des récents corei7, fer de lance des processeurs multicoeurs grand public : 3.33 GHz en fréquence de base. La baisse de fréquence n'implique pas nécessairement une baisse de performances, car l'efficacité de l'architecture joue également, et va en s'améliorant en général.

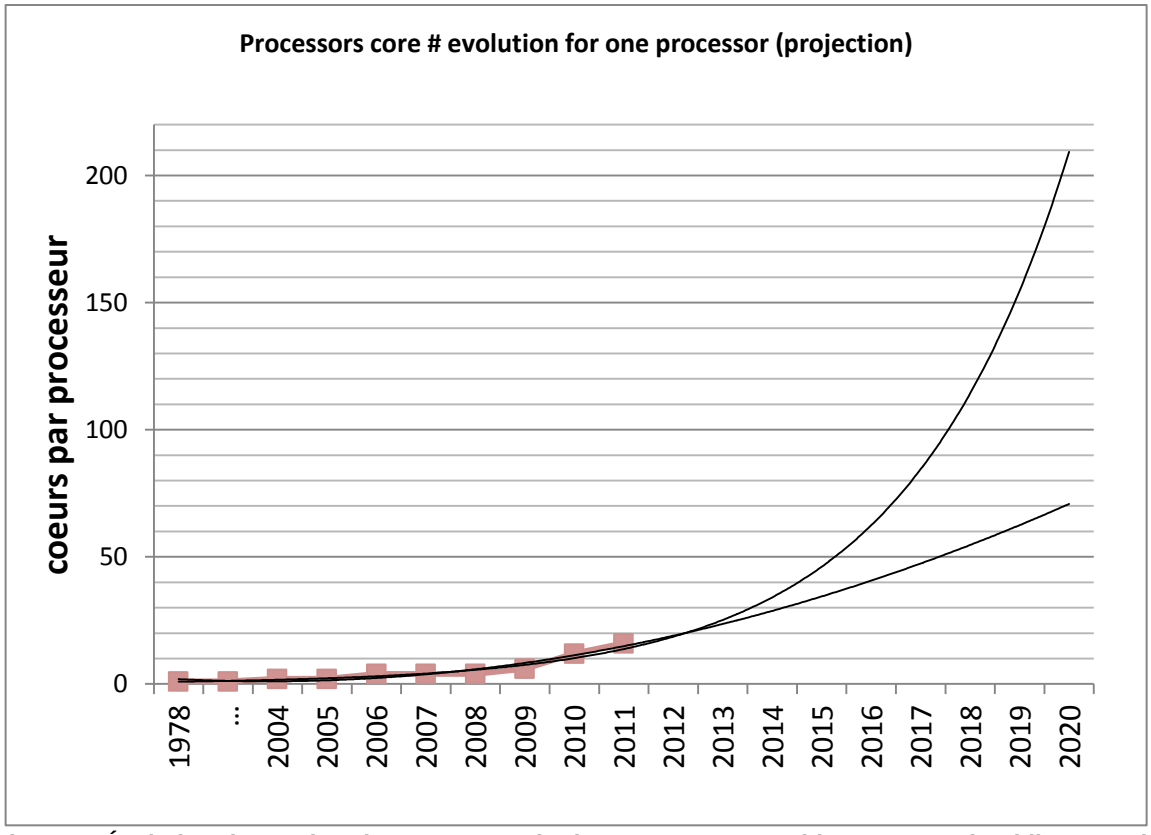


Figure 3. Évolution du nombre de cœurs au sein des processeurs multicoeurs grand public. A partir de 2012, il s'agit de deux projections utilisant une loi exponentielle (optimiste) ou logarithmique (pessimiste).

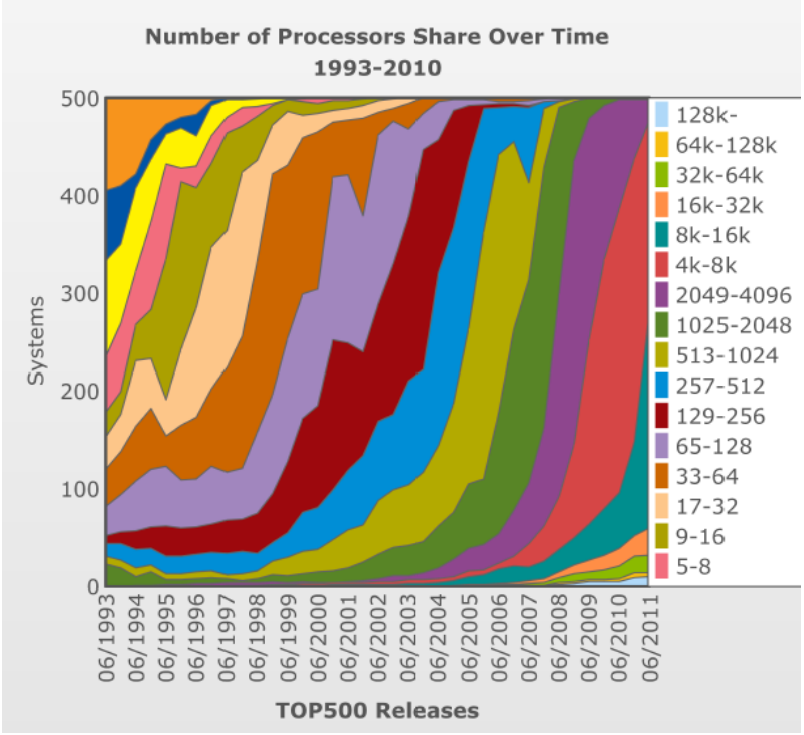


Figure 4. Répartition du nombre de processeurs/cœurs par système du top500, de 1993 à 2011.

Cette augmentation de la performance par le parallélisme se retrouve dans la figure 4 représentant la répartition du nombre de cœurs par paliers, au sein des machines du top500, de 1993 à aujourd'hui. Le top500 est un classement des 500 machines de calculs les plus rapides au monde. A chaque nouveau top500, on a assisté à un glissement du nombre de processeurs/cœurs par système le dominant, de 65-256 pour plus de 50% des systèmes en 2001, à 4000-16000 pour plus de 60% des systèmes en 2011 par exemple.

1.1.2.1. Processeurs multicoeurs

Dans la section précédente, nous avons vu quelles étaient les motivations à produire des processeurs multicoeurs. Leur implémentation peut varier en fonction du fabriquant, mais le principe général restait le même jusqu'à 2010 : plusieurs cœurs identiques sont placés dans le même processeur. La présence de plusieurs cœurs introduit de nouvelles problématiques pour la gestion de la répartition de la mémoire située sur le processeur : les caches.

Ces caches permettent de garder dans une mémoire rapide, proche du processeur, les données les plus utilisées lors de calculs, afin de cacher la grande latence de la mémoire centrale comme décrit en détails dans (3). On trouve de 1 à 3 niveaux de caches appelés L1, L2 et L3, avec des caractéristiques différentes : plus le niveau est petit, plus il est rapide mais également cher à produire. Pour ces raisons, la taille de L1 est de l'ordre de 100 Ko, celle de L2 de l'ordre de 1 Mo, et celle de L3 de l'ordre de 10 Mo. L'emplacement de ce cache a varié au fur et à mesure du temps, d'une puce mémoire sur la carte mère, hors du processeur, à des solutions où tout est intégré sur la surface du processeur. L'introduction du multicoeur a eu une influence sur la répartition des caches : pour des raisons de cohérences des données dans le cadre de calculs parallèles, des caches L1 et L2 sont en général implantés sur chaque cœur d'exécution, et le cache L3 est partagé par tous les cœurs.

Au-delà de la répartition des caches, une autre avancée apparue au début des années 2000 est la modification de l'emplacement du contrôleur mémoire permettant au processeur d'envoyer et recevoir des données de la mémoire principale. Ce dernier, originellement présent sur la carte mère, limitait la bande passante mémoire atteignable à cause du bus mémoire le reliant au processeur, qui était synchronisé sur la fréquence de la carte mère. Il a ensuite été déplacé au sein du processeur même, permettant un découplage du bus mémoire lié à la mémoire principale, et une augmentation très forte de la bande passante mémoire, qui passait alors d'un seul canal mémoire à plusieurs. Actuellement, le maximum est de 4. Cette augmentation de la bande passante mémoire fut cruciale pour permettre une meilleure scalabilité des algorithmes en général, et en particulier ceux limités par la bande passante mémoire. La figure 5 montre la présence de contrôleurs mémoires pour la mémoire principale et les entrées/sorties, pour le l'architecture Sandy Bridge d'Intel.

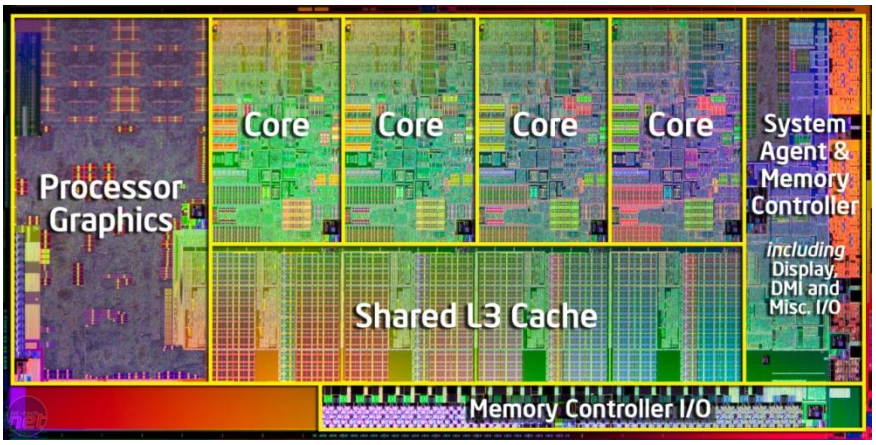


Figure 5. Image de la surface d'un processeur Sandy Bridge d'Intel. On note la présence de cœurs de calcul, de contrôleurs mémoires et d'un processeur graphique.

L'autre avancée nécessaire à la scalabilité potentielle des applications au sein d'un multicoeur est l'interconnexion entre les cœurs. Comme énoncé précédemment, les premiers dual-cœur grand public utilisaient le bus mémoire de la carte mère pour faire communiquer leurs cœurs. Le débit entre ces derniers était de l'ordre de la centaine de Mo/s, ce qui était faible au vu de la vitesse de la mémoire principale et des caches processeurs. De plus, la latence était relativement élevée, ce qui impliquait une synchronisation des deux cœurs très coûteuse. Par la suite, des liens spécifiques ont été développés entre les cœurs, leur permettant des échanges plus rapides que la mémoire principale, et avec une latence plus faible. Les technologies grand public utilisées sont historiquement l'HyperTransport d'AMD et le QuickPath Interconnect (QPI) d'Intel.

1.1.2.1.1. Intégration du GPU sur la surface des multicoeurs

L'autre apport récent des fabricants de processeurs multicoeurs grand public est l'ajout d'un processeur graphique au sein de leur processeur. Cet ajout, certainement motivé par l'utilisation de plus en plus généralisée des cartes graphiques pour le calcul, est réalisé de manière différente parmi les fabricants de processeurs grand public. Intel a ainsi tout d'abord ajouté un processeur graphique sur la surface du multicoeur, sans que ces deux éléments ne partagent d'information avec la génération Bloomfield. Par la suite, ce sont les transistors du GPU qui ont été intégrés sur la surface du multicoeur, avec partage du cache L3, pour la génération Sandy Bridge. Un autre ajout technologique au sein des processeurs Sandy Bridge est la technologie QuickSync, qui fait appel au GPU intégré et à quelques circuits dédiés pour accélérer le transcodage de vidéos. Le but est de concurrencer les GPUs sur un terrain où ils excellent habituellement. Ainsi, pour l'opération d'encodage ou décodage vidéo, un GPU pouvait fonctionner 2 ou 3 fois plus vite qu'un processeur multicoeur. Avec QuickSync, le multicoeur effectue à son tour les opérations de transcodage 2 à 3 fois plus vite… que les GPUs. On assiste ainsi à la spécialisation de certains composants d'un processeur généraliste, qui pourrait être une solution pour atteindre à l'avenir une plus grande efficacité de calcul.

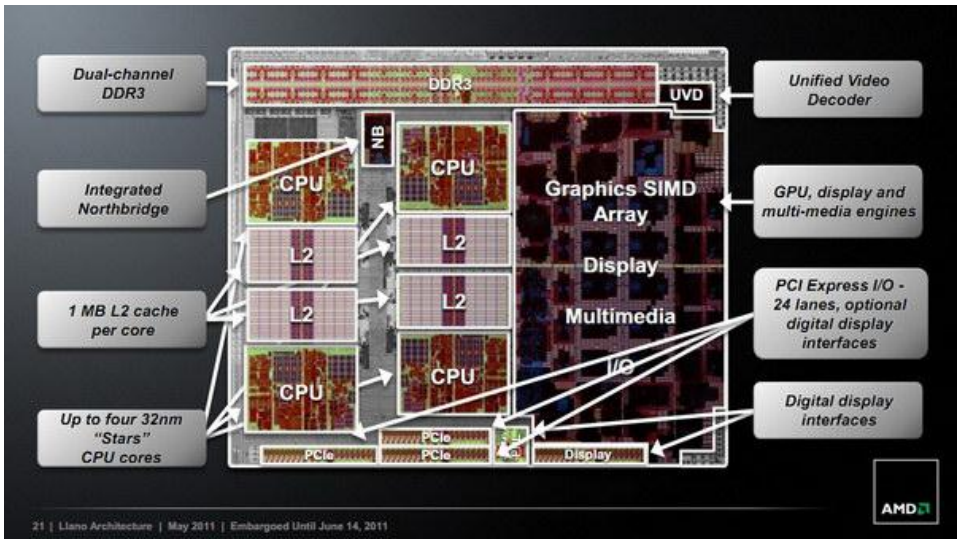


Figure 6. Surface d'un processeur AMD Fusion intégrant puce graphique et cœurs x86 standards.

La solution d'intégration proposée par un autre fabricant de processeurs, AMD est directement l'intégration des transistors du GPU sur la surface du processeur. La figure 6 montre la surface d'un de ces processeurs appelé « Fusion ». Les caches ne sont ici pas partagés, l'architecture GPU AMD possédant déjà ses propres registres et caches. La différence par rapport à la solution Intel Sandy Bridge est une technologie GPU supérieure, et notamment la présence de plusieurs centaines d'unités de calcul, alors que le GPU des Sandy Bridge n'en possède qu'une dizaine. De plus, le GPU des processeurs Fusion est programmable, alors que la solution Intel ne l'est pas. Nous verrons les paradigmes de programmation plus tard dans ce chapitre.

1.1.2.1.2. Cœurs virtuels et partage de ressources

Une autre solution pour contourner le Power Wall est une meilleure utilisation des cœurs de calcul. En effet, dans un programme les opérations de calcul, très rapides, sont souvent en attente de données d'entrée ou de l'autorisation d'écrire en mémoire. Ainsi l'utilisation des cœurs n'est pas efficace, et une solution appelée Simultaneous Multi Threading (SMT) permet de palier à ces défauts. Le principe revient à simuler au niveau du système d'exploitation l'utilisation de plusieurs cœurs virtuels avec un seul cœur physique. En pratique, il n'est nécessaire de dupliquer qu'une petite partie d'un cœur pour autoriser la gestion de deux ou plusieurs fils d'exécutions simultanés sur un seul cœur. La partie calcul n'est pas nécessairement dupliquée par contre. Ainsi, selon Intel, augmenter la surface du processeur de seulement 5% introduit un gain moyen de 15 à 30% sur les temps d'exécutions. La figure 7 montre le principe de fonctionnement de l'hyperthreading, implémentation SMT d'Intel. En pratique, les gains annoncés sont complètement dépendants de l'application, et peuvent même se révéler négatifs.

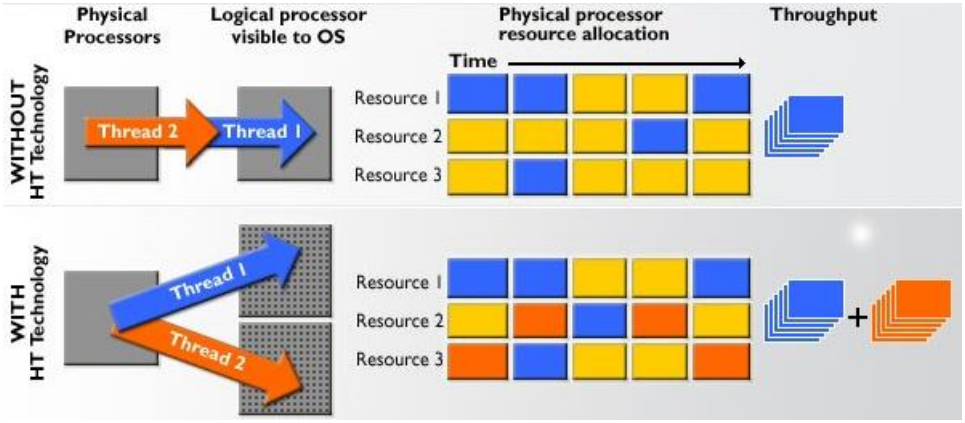


Figure 7. Fonctionnement de l'Hyperthreading sur processeur Intel.

Une autre approche est la création d'unités de calculs appropriées à l'utilisation finale du processeur, qui rentre en quelque sorte dans le principe du codesign : créer un processeur avec les utilisateurs pour qu'il satisfasse leurs besoins. Le fabricant de processeur AMD propose cette idée dans l'implémentation de son architecture Bulldozer, présentée dans la figure 8. Le nom de la technologie SMT d'AMD est Clustered Multi Threading. Le constat d'AMD est que statistiquement, dans un usage général, les unités double précision ne sont pas utilisée à 100%, et les unités arithmétiques entières sont beaucoup plus sollicitées tout comme d'autres parties d'un cœur de processeur telles que le décodage, prefetching, etc… De ce fait, il est possible d'allouer deux cœurs traitant les entiers indépendamment et partageant la même unité flottante et d'autres parties du cœur. Le nom de cet ensemble est alors appelé module, qui contient deux cœurs. Ainsi, un processeur à 8 cœurs possédera 4 modules, et « seulement » 4 unités flottantes 256 bits, ou 8 unités 128 bits. Selon AMD, les performances pour un module à deux cœurs se répartiraient ainsi, comparées à celle de l'hyperthreading Intel pour un seul cœur :

- 2 threads sur un module à 2 cœurs « Bulldozer » : 160 à 180% d'un thread
- 2 threads sur un cœur Intel Nehalem hyperthreadé : 85 à 130% d'un thread.

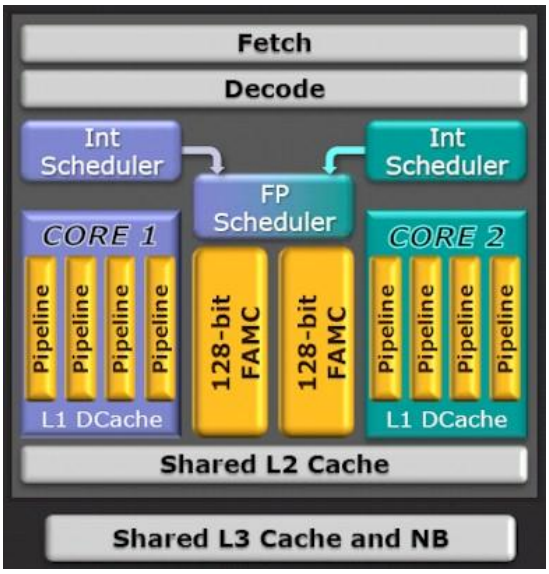


Figure 8. Architecture du processeur Bulldozer d'AMD.

Tout comme le SMT, le but est d'augmenter la performance en offrant un coût raisonnable et également une enveloppe thermique acceptable. On voit donc que le choix de l'implémentation des processeurs est modulé par plusieurs paramètres parmi lesquels la

performance, la consommation énergétique et le coût de fabrication. Selon le domaine d'application, plusieurs architectures spécialisées permettent de privilégier l'un de ces paramètres.

1.1.2.2. Architectures hétérogènes

Pour répondre au problème du Power Wall, la solution de spécialisation des FPGAs est une option difficilement utilisable en pratique pour le calcul scientifique à cause des spécificités de programmation. De leur côté, les architectures ARM sont intéressantes car plus généralistes, mais avec quelques limitations concernant leur puissance de calcul brute, pour le moment. Les multicoeurs ont un intérêt également, mais leur conception pour un usage général fait qu'une grande partie de la surface du multicoeur est dédiée au contrôle du flux de données et non au calcul. A cet effet, certaines architectures ont été imaginées pour offrir une performance optimale suivant certaines contraintes de programmation. L'efficacité énergétique s'en trouve renforcée, de par un ratio performance par Watt extrêmement optimisé. Ainsi, le premier super-ordinateur à avoir dépassé la performance d'1 Pflops (10^{16} Flops) est le Roadrunner, une machine exploitant des processeurs multicoeurs généraliste AMD Opterons, et des accélérateurs IBM Cell. Parmi les machines qui ont par la suite rejoint le Roadrunner dans l'ère Petascale, la majeure partie dispose d'accélérateurs graphiques, des cartes graphiques Nvidia. De manière intéressante, l'ajout des accélérateurs dans les super-calculateurs est similaire à l'adoption massive de multicoeurs grand public durant les années 1990 : l'économie d'échelle par l'adoption de matériel massivement consommé par le grand public : les cartes vidéos accélératrices pour le marché des « gamers ».

1.1.2.2.1. Cell

Le processeur Cell, fruit de l'union du groupement STI (Sony, Toshiba, IBM), fut à l'origine produit pour équiper une console de jeux-vidéos, la Playstation 3 produite par Sony. Le but de ce processeur était de fournir une performance par Watt élevée pour du calcul relativement général appliqué au domaine du jeu-vidéo. La simple précision (32 bits) domine ainsi les calculs, avec une perte de précision non dérangeante dans le cadre de calcul d'image ou de collisions approximées.

Ce processeur possède une architecture dite hétérogène à base d'un cœur IBM PowerPC fonctionnant à 3.2 GHz offrant une performance relativement faible à cause de son architecture. Ce processeur, appelé PowerPC Element (PPE), a en réalité le rôle de « chef d'orchestre », dont les instruments à diriger sont les 8 unités de calculs vectoriels 128 bits, visibles sur la figure 9. Ces unités sont appelées les Synergistic Processing Elements (SPE). Chacun de ces SPEs développe une puissance de 25.6 GFlops en simple précision, pour un total de 205 GFlops. La première version du Cell offrait une performance de seulement 20 GFlops en double précision avec les 8 SPEs. Par la suite, cette différence avec la simple précision a été ramenée à un facteur 2 plus habituel, pour 100 GFlops avec 8 SPEs. La particularité de ces SPEs est d'être in-order, ce qui implique que l'optimisation de l'utilisation du pipeline d'instructions est laissée au développeur. C'est un inconvénient certain, mais l'avantage de ce choix est une efficacité très grande, proche de 100%. Dans le chapitre 3, nous étudierons plus en détails les problèmes de précision arithmétique flottante liés à l'implémentation IEEE du Cell.

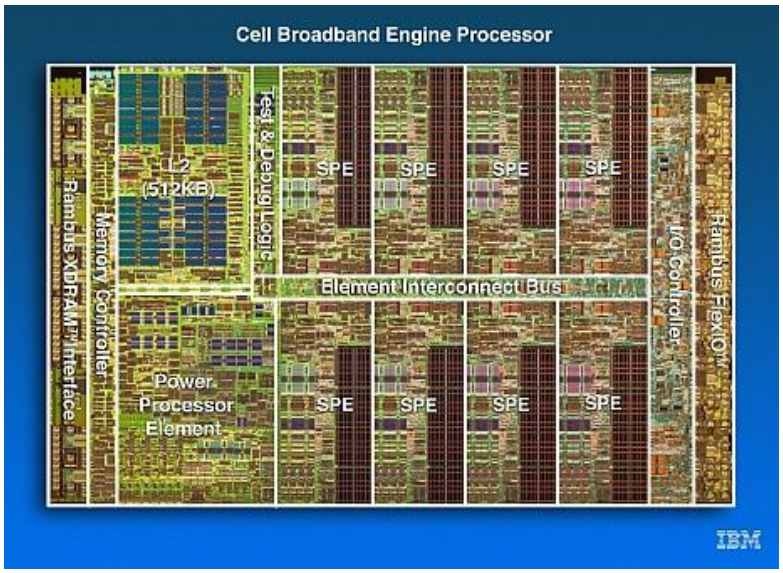


Figure 9. Surface du processeur IBM Cell.

Les 8 SPEs possèdent une mémoire locale de 256 Ko appelée Local Storage (LS). Cette mémoire sert à la fois au stockage des données pour les calculs, mais aussi pour le programme lui-même. Cela implique que le traitement d'un très long vecteur ne peut s'effectuer complètement dans le LS. Il faut que le PPE transfère un morceau de ce vecteur jusqu'au LS du SPE, puis que ce dernier effectue le calcul, et ainsi de suite jusqu'à complétion de l'opération. Il est possible d'effectuer des transferts asynchrones, et donc de calculer pendant que l'on transfère des données du PPE vers un SPE.

Les données transitent entre PPE et SPEs mais également entre les SPEs au travers d'un bus mémoire appelé Element Interconnect Bus. Ce dernier est constitué de quatre anneaux unidirectionnels reliant les 9 éléments de calculs. Deux anneaux sont orientés dans un sens de parcours, et les deux autres dans l'autre sens. Ce bus offre une bande passante totale de 200Go/s, ce qui est extrêmement rapide au regard des technologies concurrentes d'interconnexion entre cœurs de calculs, de l'ordre de la dizaine de Go/s.

La mémoire principale était de la XDR dans les premières versions commercialisées du Cell, offrant une bande passante de 25,6 Go/s vers le PPE. Cette mémoire, fournie par la société Rambus, proposait une bande passante excellente à un prix relativement élevé. Par la suite, la mémoire a été remplacée par de la DDR2 sur quatre canaux, moins onéreuse pour la même performance.

Avec son architecture, le Cell offre un processeur très équilibré, qui permet aux applications denses en calcul d'obtenir une performance très proche de la puissance crête. Les obstacles principaux dans l'utilisation de cette architecture sont la gestion explicite du LS et les transferts mémoires à partir de la mémoire principale du PPE, l'obligation de vectoriser les calculs sur chaque SPE, et finalement la nécessité d'éviter tout branchement imprévisible sous peine d'impacter fortement les performances. Ce dernier défaut est dû à l'architecture in order du Cell. Ce défaut se retrouve d'ailleurs également dans les cartes graphiques, que nous présenterons à la section suivante.

1.1.2.2.2. GPUs

L'utilisation des GPUs pour le calcul scientifique n'est pas nouvelle mais l'introduction du langage CUDA en 2007, tout comme le soutien fort du fabricant de processeurs Nvidia pour le calcul scientifique, ont permis un essor important de l'intérêt et l'expérimentation du calcul sur cartes graphiques. Cependant, cette adoption massive des accélérateurs graphiques

n'est pas motivée uniquement par le soutien d'un fabricant où un langage tel que CUDA. Les raisons principales sont présentées dans la figure 10 et la figure 11 : une performance crête flottante et une bande passante mémoire supérieures.

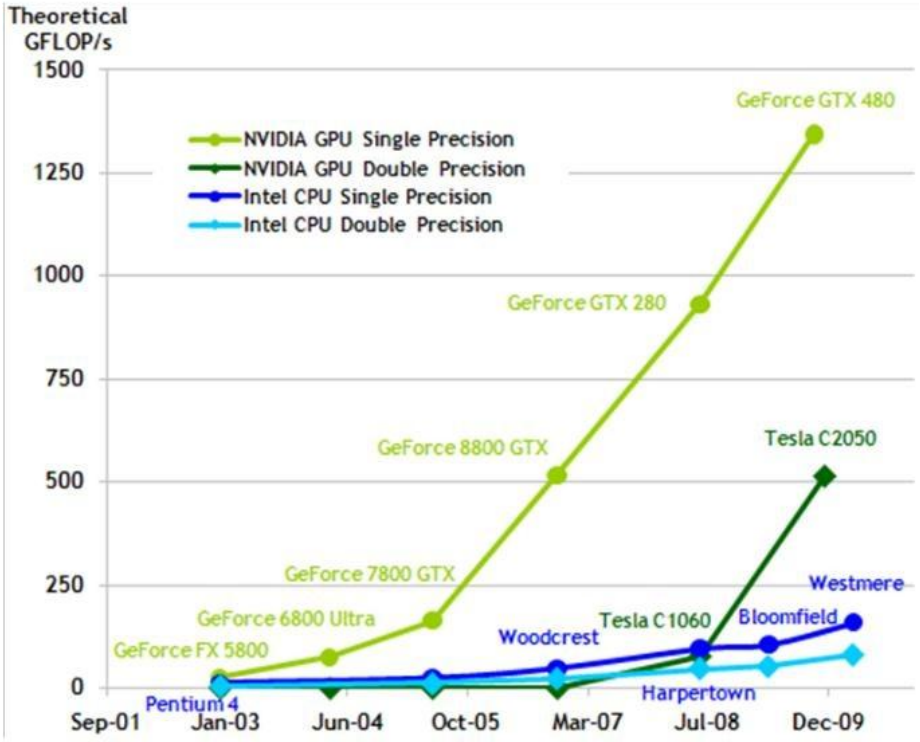


Figure 10. Performance crête simple et double précision des processeurs Intel et les cartes graphiques NVidia.

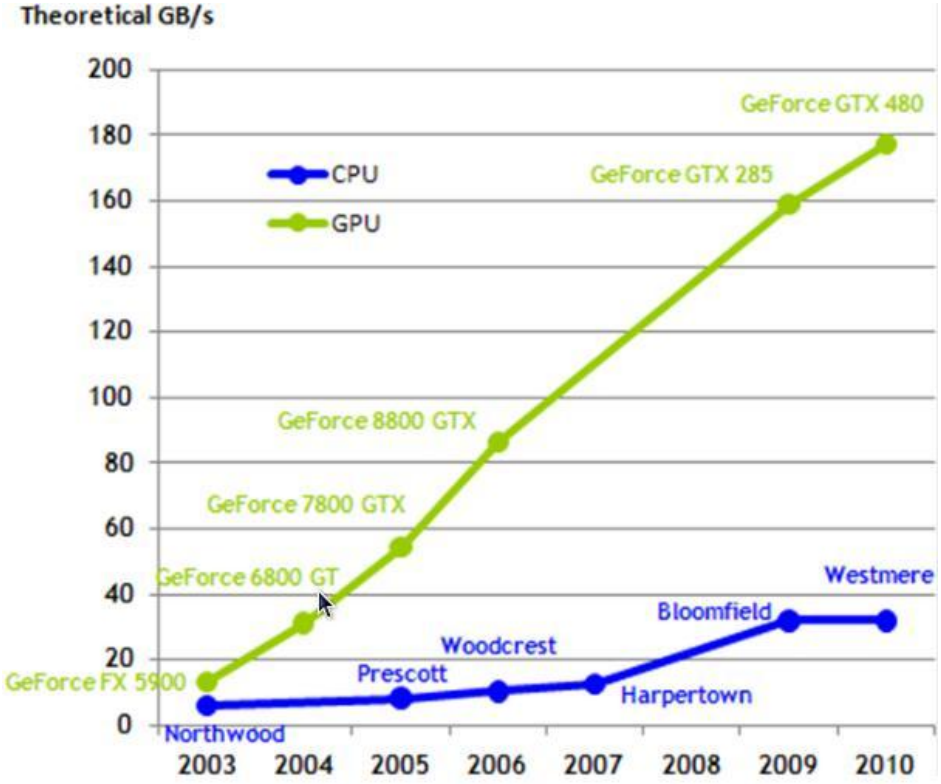


Figure 11. Augmentation de la bande passante mémoire entre processeurs Intel et cartes graphiques NVidia.

Ces deux arguments promettent en apparence une accélération pour tout algorithme de calcul, étant donné la supériorité de la vitesse de la mémoire et du calcul. La réalité est tout autre, pour plusieurs raisons. Le GPU est une unité massivement parallèle, contenant plusieurs centaines de cœurs de calculs, bien différents d'un cœur de multicoeur classique. Donc, pour atteindre la performance crête, l'application scientifique doit pouvoir exprimer un parallélisme massif de plusieurs centaines ou milliers de threads légers. En admettant que cela soit possible, il faut que chaque thread ait une manière d'accéder à la mémoire qui soit relativement régulière, à la manière d'une machine vectorielle. Dans le cas où les accès seraient complètement aléatoires ou irréguliers, à la manière d'une structure de matrice creuse, alors la performance sur GPU serait catastrophique. Cette contrainte a une incidence sur le traitement des branchements conditionnels, qui à chaque if-then-else divise la performance par deux, si le else est parcouru en même temps que le then. La figure 12 montre la différence par rapport à un CPU classique : la plus grande partie de la surface du processeur graphique est dédié au calcul (ALU), avec peu ou pas de gestion des branchements, à l'inverse du CPU. Ce dernier possède en effet une unité de contrôle respectable, accompagnée de caches relativement importants.

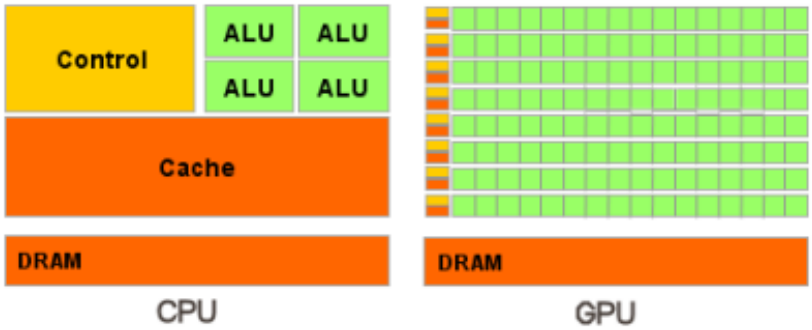


Figure 12. Différence d'organisation d'un processeur graphique et généraliste. La gestion du contrôle des branchements conditionnels est supérieure sur CPU, avec notamment un cache très important. Sur GPU par contre, la majeure partie de la surface est dédiée au calcul, massivement parallèle.

Donc, pour qu'un programme puisse tirer parti de ces promesses d'accélération d'un facteur de 5 à 10, il faut qu'il puisse tirer parti des différentes unités de calcul, qui sont réparties en multiprocesseurs. Le multiprocesseur contient plusieurs cœurs élémentaires de calcul, qui peuvent communiquer au sein de ce dernier mais pas avec le cœur d'un multiprocesseur différent. Sur la figure 13, on peut voir une vue abstraite de la répartition des unités de calcul du GPU et de la hiérarchie mémoire interne.

Une autre limitation des GPUs est liée à l'adressage mémoire différent de celui du CPU central. La mémoire de ce dernier n'est pas directement accessible au GPU, et vice-versa. De plus, le lien mémoire unissant GPUs et CPU est le PCI-Express, relativement lent par rapport à la mémoire principale ou sur GPU : 2 à 32 Go/s pour le PCI-Express, 5 à 26 Go/s pour la mémoire principale et 70 à 180 Go/s pour la mémoire vidéo. La latence du PCI-Express est relativement élevée par rapport aux autres interfaces mémoires : 15 microsecondes (10^{-9}s) pour le PCI-Express (4) contre environ 10 nanosecondes (10^{-12}s) pour la mémoire principale et 200 à 550 nanosecondes (10^{-12}s) (4) pour la mémoire vidéo. Ainsi, pour utiliser efficacement un GPU, il faut soit réussir à faire tenir les calculs dans la mémoire de ce dernier, ou pouvoir recouvrir les communications PCI-Express par du calcul. Le premier cas n'est pas toujours possible, la mémoire disponible sur GPU étant actuellement limitée à 6 Go, et le second n'est envisageable que si l'algorithme est plutôt dense en calculs.

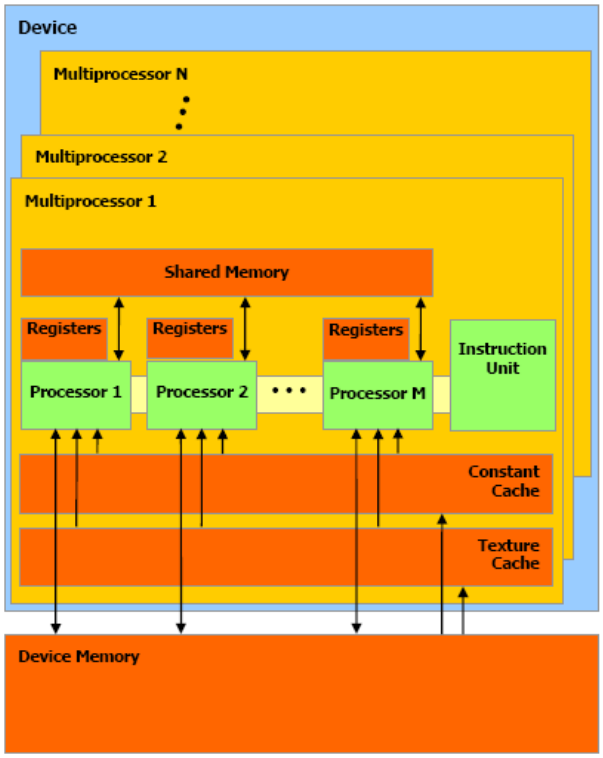


Figure 13. Répartition de la hiérarchie mémoire et des unités de calcul sur GPU Nvidia Tesla.

1.2. Évolution des paradigmes de programmation

Les évolutions matérielles présentées dans les sections précédentes s'accompagnent également d'une évolution des paradigmes de programmation. De manière générale, le matériel évolue plus vite que ces paradigmes, qui eux-mêmes évoluent plus rapidement que les logiciels les utilisant.

1.2.1. Mémoire distribuée

La mémoire distribuée est présente dans les architectures de type cluster ou grilles de calcul peuplant la majorité des superordinateurs du top500. Chaque processeur dispose de sa propre mémoire, et ne peut accéder directement à celle d'un autre processeur. La figure 14 présente un schéma de machine à mémoire distribuée. Pour qu'un processeur puisse accéder à la mémoire d'un autre, il doit effectuer des opérations mémoires à travers le réseau reliant les deux processeurs. A noter, chaque « processeur(s)-x » peut-être en lui-même un processeur multicoeur.

Le paradigme devenu de facto le standard en mémoire distribué est le passage de message, via l'API Message Passing Interface (MPI). Ce standard fut conçu en 1993-1994, et propose l'utilisation du modèle Single Program Multiple Data (SPMD) pour l'exécution parallèle d'une application. Via MPI, on lance un ensemble de processus exécutant la même application. La seule différence entre ces processus est leur numéro de rang initial au sein de l'exécution MPI. Grâce à ce numéro, on peut différencier le traitement à appliquer, et donc répartir les calculs de manière distribuée. MPI propose un ensemble de fonctions permettant des communications point-à-point et collectives entre les processus. Typiquement, l'échange illustré par la flèche rouge sur la figure 14 peut être réalisé par une opération d'envoi MPI « send » du processeur 2 vers le processeur n, et une opération de réception MPI « receive » par le processeur n en attente de 2. Au fur et à mesure des

versions, l'interface MPI s'est enrichie, proposant des transferts asynchrones entre les processus, des communicateurs, etc.

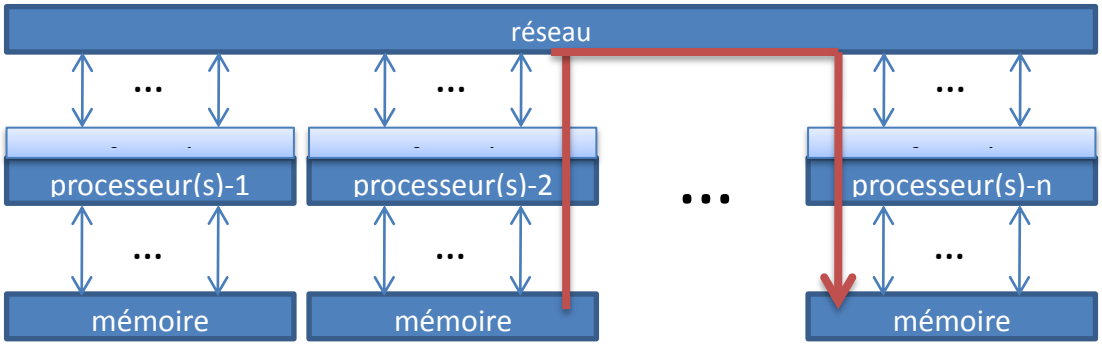


Figure 14. Vue d'une machine à mémoire distribuée possédant n processeurs. Dans cet exemple, le processeur 2 envoie des données au processeur n, nécessairement au travers du réseau.

Dans la section suivante, nous développons le principe du calcul en mémoire partagée, et il est intéressant de noter que la programmation en mémoire distribuée s'applique aussi en mémoire partagée, mais pas l'inverse.

1.2.2. Mémoire partagée

On appelle machine à mémoire partagée une machine possédant plusieurs processeurs partageant une même mémoire. La figure 15 illustre ce paradigme, que l'on retrouve par exemple au niveau d'un nœud de calcul dans un cluster. Sur la Figure 14, l'un des « Processeur(s) » pourrait ainsi être un nœud contenant plusieurs processeurs à plusieurs cœurs. Pour l'utilisation parallèle d'une machine à mémoire partagée, il est possible de faire appel au paradigme du passage de messages, dans le style de MPI. L'inconvénient potentiel de cette stratégie est l'envoi de messages explicites entrainant des copies mémoires superflues alors que les processeurs sont capables d'accéder à toute la mémoire partagée. Cela peut se traduire par une performance non optimale, même si plusieurs bibliothèques MPI proposent des optimisations pour les communications intranoeuds.

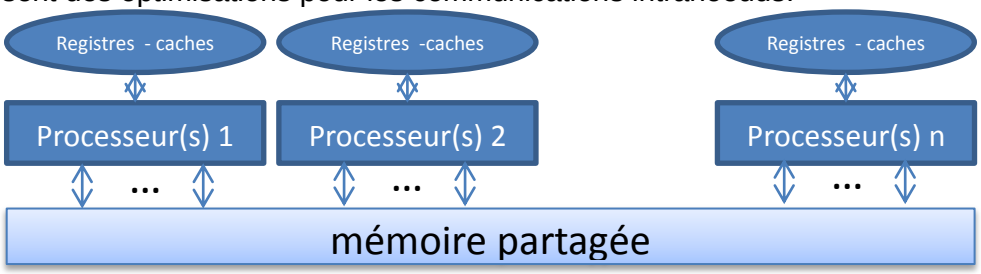


Figure 15. Vue d'une machine à mémoire partagée.

Traditionnellement, le paradigme de programmation utilisé pour la mémoire partagée est l'appel à des threads ou fils d'exécutions. Un thread peut être apparenté à un processus, mais beaucoup plus léger. Chaque thread sera associé à un cœur, et effectuera sa portion de calcul en utilisant potentiellement toutes les données en mémoire partagée. Ainsi, aucun transfert entre processeurs n'est nécessaire. L'exécution de threads peut être invoquée explicitement via la librairie disponible dans le système d'exploitation cible, telle que pthread pour linux. Ce style de programmation est assez bas niveau, et peu portable. Une technologie de plus haut niveau est apparue en 1997 : OpenMP. Le principe d'OpenMP est d'exécuter un thread maître, qui distribuera le travail parallèle à des threads esclaves invoqués dans les sections parallèles du code. La programmation OpenMP se fait par

directives de compilation, avec possibilité d'utiliser des fonctions de la librairie OpenMP pour les réglages de l'exécution parallèle. L'avantage majeur d'OpenMP est de permettre une parallélisation rapide d'un code séquentiel, sans modifications majeures de ce dernier. De plus, si le compilateur ne supporte pas les directives OpenMP, alors ces dernières seront ignorées, permettant au code une excellente portabilité. Le tableau 1 présente un exemple de parallélisation OpenMP du noyau axpy, qui agit sur deux vecteurs, et le programme exemple « HelloWorld ».

Tableau 1. Parallélisation OpenMP du noyau de calcul axpy, qui pour deux vecteurs y et x de taille n, calcule y = y + alpha * x, avec alpha un scalaire. Le programme Helloword est également présenté.

Noyau axpy	Programme HelloWorld

#pragma omp parallel for
for (int i = 0 ; i < n ; i++)
  y[i] = y[i] + alpha * x[i]
	#include <omp.h>
	#include <stdio.h>
	#include <stdlib.h>
	
	int main (int argc, char *argv[]) {
	  int th_id, nthreads;
	  #pragma omp parallel private(th_id)
	  {
	  th_id = omp_get_thread_num();
	  printf("Hello World from thread %d**\n**", th_id);
	  #pragma omp barrier
	  if (th_id == 0) {
	  nthreads = omp_get_num_threads();
	  printf("There are %d threads**\n**",nthreads);
	  }
	  }
	  return EXIT_SUCCESS;
	}

D'autres paradigmes de programmation faisant appel aux threads existent, tels que Thread Building Blocks d'Intel, qui propose une structure fortement orientée objet pour implémenter le parallélisme, avec de nombreux outils logiciels tels que les pipelines. Cilk peut également être cité, avec la possibilité de générer des tâches parallèles à la volée (spawn), automatiquement organisées par le scheduler interne de manière optimale.

1.2.3. Architectures hétérogènes / accélérateurs

Les premiers paradigmes de programmation permettant l'utilisation d'accélérateurs tels que le Cell, les GPUs ou les cartes Clearspeed, sont propriétaires. Par la suite, une norme unificatrice pour le calcul parallèle multicoeur ou accélérateur est apparue : OpenCL. Ce point de vue est centré sur les langages, mais les concepts illustrant l'utilisation du parallélisme massif existaient déjà auparavant, tel que présenté dans (5).

1.2.3.1. Processeur Cell

Pour le Cell, il s'agit d'extensions C pour d'une part le processeur central PPE, d'autre part les unités vectorielles SPEs, et finalement les transferts mémoires entre ces éléments. Il existe plusieurs manières d'utiliser ensuite le Cell. Généralement, le PPE initialise les données des calculs, puis lance l'exécution de threads sur les SPEs. Ces derniers peuvent être utilisés de plusieurs manières : ils exécutent le même programme (thread), à la manière d'une exécution OpenMP, pour partager les calculs uniformément. Une autre solution est une exécution asymétrique, pour des traitements de type pipelinés. L'exemple type serait le traitement d'image : le PPE charge une image en mémoire, qu'il transmet ensuite au premier SPE. Ce dernier applique alors un traitement sur l'image ou une partie de cette dernière, puis ensuite transmet le résultat au deuxième SPE qui applique un traitement ou filtre différent. Ainsi, le Cell peut être utilisé comme une unité pipelinée tirant partie de son réseau

d'interconnexion très rapide. Les SPEs pouvant accéder aux SPEs d'un autre Cell, ce pipeline peut être encore plus allongé, bénéficiant ainsi d'une efficacité très bonne typique à celle des machines vectorielles. La dernière solution pour exécuter des calculs sur le Cell est le SPUlet, un programme/thread exécuté uniquement sur un seul SPE. Ce mode est fortement limité, et rarement utilisé pour le calcul hautes performances.

Tableau 2. Programme HelloWorld sur le processeur Cell.

Programme source PPE	Programme source SPE

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <libspe.h>
#include <sched.h>
 	
extern spe_program_handle_t hello_spu;
spe_gid_t gid;
speid_t speids[8];
int status[8];
 	
int main(int argc, char *argv[]){
  int i;
 	
  printf("Hello World!\n");
 	
  gid = spe_create_group (SCHED_OTHER, 0, 1);
  if (gid == NULL) {
  fprintf(stderr, "Failed spe_create_group(errno=%d)\n", errno);
  return -1;
  }
 	
  if (spe_group_max (gid) < 8) {
  fprintf(stderr, "System doesn't have eight working SPEs. I'm leaving.\n");
  return -1;
  }
 	
  for (i = 0; i < 8; i++) {
  speids[i] = spe_create_thread (gid, &hello_spu,
  NULL, NULL, -1, 0);
  if (speids[i] == NULL) {
  fprintf (stderr, "FAILED: spe_create_thread(num=%d, errno=%d)\n",
  i, errno);
  exit (3+i);
  }
  }
 	
  for (i=0; i<8; ++i){
  spe_wait(speids[i], &status[i], 0);
  }
 	
  __asm__ __volatile__ ("sync" : : : "memory");
 	
  return 0;
}
	/* This union helps clarify calling parameters between the PPE and the SPE. */
	typedef union{
	  unsigned long long ull;
	  unsigned int ui[2];
	}addr64;
	
	int main(unsigned long long speid, addr64 argp, addr64 envp) {
	  printf("Hello world, from a SPU!");
	
	  return 0;
	}

Globalement, le Cell a connu un fort engouement car ce matériel était présent dans des consoles de salon abordables, et a bénéficié d'une bonne publicité grâce au supercalculateur Roadrunner, emblématique première machine à dépasser le Petaflops. Cependant, la nécessité de redévelopper le code des applications pour tirer parti des instructions spécifiques au Cell, sa difficulté de programmation due aux échanges mémoires explicites, la gestion explicite des mémoires locales des SPE, et la nécessité de vectoriser le code pour chaque SPE, ont progressivement entraîné l'abandon de cette plateforme, qui de plus n'était plus amenée à évoluer selon son constructeur IBM. Finalement, l'attrait de la programmation bas niveau du Cell aura été son avantage et également son inconvénient majeur. Une illustration de cette difficulté est présentée dans le tableau 2, pour un simple programme « HelloWorld ».

1.2.3.2. Cartes graphiques

Historiquement, les premières expérimentations sur cartes graphiques faisaient appel à des appels directs aux librairies graphiques qui calculent des polygones ou des manipulations d'image. Ainsi des opérations matricielles extrêmement contraintes étaient possibles, avec une programmabilité très faible. Par la suite, des initiatives telles que Brook (6) ont permis une utilisation plus aisée des GPUs pour le calcul scientifique, avec cependant quelques limitations importantes, la première étant l'exécution de noyaux de calculs nécessairement synchrones. On ne pouvait donc pas envisager de recouvrir les communications entre GPU et CPU par du calcul

1.2.3.2.1. CUDA

La première véritable généralisation du calcul sur carte graphique est liée à la création du langage CUDA par le constructeur de cartes graphiques Nvidia. Ce langage cible exclusivement les cartes graphiques de ce constructeur, et est incompatible avec les autres. CUDA fournit une abstraction haut-niveau de la programmation parallèle massive sur les centaines de cœurs à disposition sur GPUs. Ainsi, le processeur hôte, un CPU standard, va demander l'exécution de noyaux écrits en CUDA au pilote graphique du système d'exploitation. L'écriture de ces noyaux repose sur l'utilisation d'une grille contenant des blocs dédiés aux calculs. Chaque bloc possède des coordonnées dans cette grille, laquelle peut être à une ou deux dimensions. Au sein de chaque bloc, un groupe de threads sera exécuté en parallèle. Le bloc peut être organisé en une, deux ou trois dimensions. L'exemple de la figure 16 montre l'exécution d'une grille à deux dimensions contenant 2*3 blocs à deux dimensions, eux-mêmes exécutant 3*5 threads. Ainsi sur cet exemple, la carte graphique exécuterait 90 threads en parallèle. Cela peut paraître être un grand nombre de threads, mais sur GPU ces derniers sont extrêmement légers, et il est tout à fait normal et recommandé d'en exécuter plusieurs milliers ou plus pour cacher la latence mémoire.

Chaque thread dispose d'un identifiant au sein de sa grille et de son bloc. Sur le schéma précédent, « Thread (4 , 2) » se situe en colonne 4 et ligne 2 du bloc situé en ligne 1 et colonne 1 de la grille. Grâce à ces coordonnées et la connaissance de la taille de la grille et d'un bloc, le thread est capable d'agir sur les données le concernant. Au-delà de la gestion de l'identité du thread, CUDA propose d'exploiter explicitement les différents niveaux de mémoire présentés dans la figure 13 au travers de mots-clés. Il est ainsi possible d'utiliser explicitement la mémoire locale, extrêmement rapide, pour des techniques de blocking dans des opérations réutilisant beaucoup les mêmes données. Finalement, CUDA offre la possibilité d'utiliser à haut-niveau un parallélisme massif, tout en tirant partie des spécificités mémoires de la carte graphique.

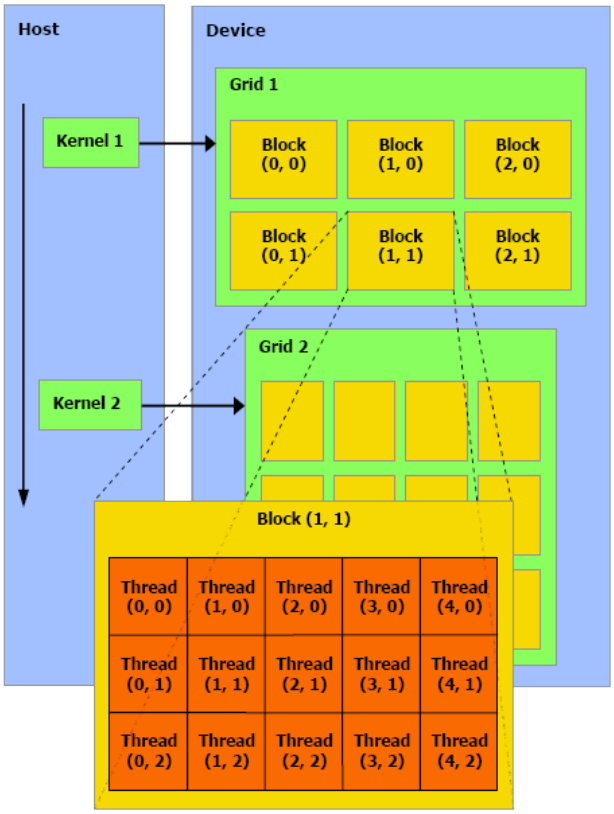


Figure 16. Abstraction du parallélisme au sein de CUDA.

1.2.3.2.2. OpenCL

Parallèlement à la création et l'adoption de CUDA, d'autres cartes graphiques provenant du constructeur AMD proposent une performance crête équivalente ou supérieure. Le langage pour exploiter ces GPUs a évolué, passant de CAL/IL (7), un langage très bas niveau, à Stream (8) un équivalent de CUDA pour cartes AMD. Cette disparité dans les langages est un frein à l'adoption d'accélérateurs graphiques, du Cell ou d'autres car une fois l'application optimisée pour utiliser l'un ou l'autre des accélérateurs, l'application devient dépendante de la stratégie à long terme de la société à l'origine du langage ou matériel.

Pour pallier à cet inconvénient, le consortium Khronos Goup, à l'origine de plusieurs normes ouvertes telles que OpenGL, pour le graphisme 3D, a travaillé à la création d'un langage unificateur appelé Open Compute Language (OpenCL). La vocation de ce dernier est de pouvoir avec un même langage utiliser plusieurs accélérateurs différents, et même des multicoeurs comme présenté dans la figure 17. Lors de la conférence SuperComputing de 2009, une démonstration présentait une machine regroupant des processeurs Cell, des multicoeurs IBM, Intel et AMD et des cartes graphiques Nvidia, qui participaient tous au même calcul dans le même programme (9).

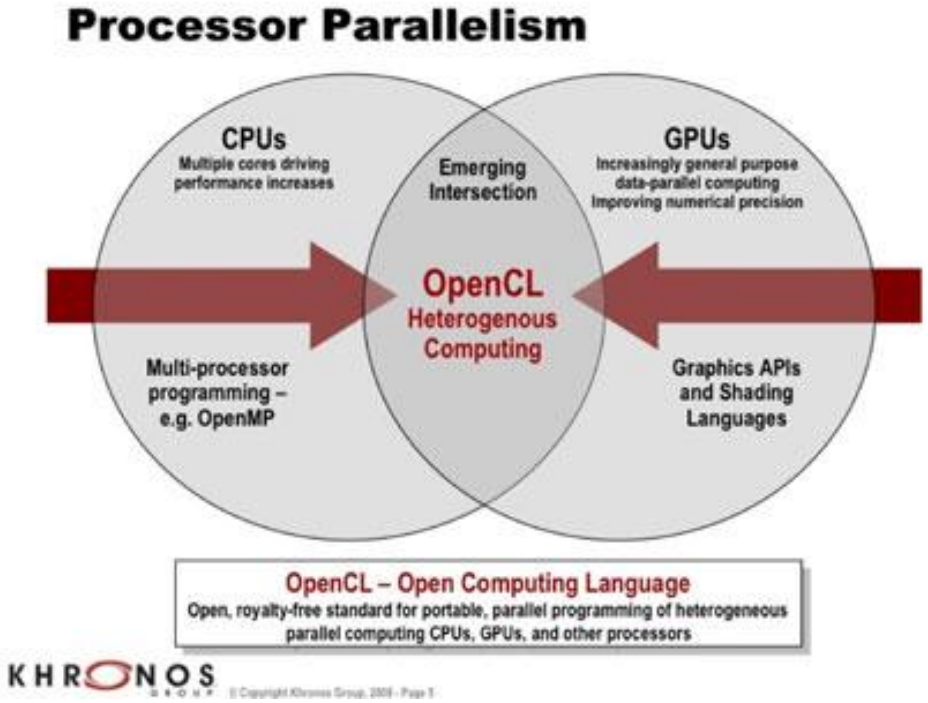


Figure 17. Positionnement d'OpenCL dans le parallélisme.

Le principe de fonctionnement d'OpenCL est fortement calqué sur celui de CUDA. Un hôte initialise les calculs, et lance l'exécution de noyaux OpenCL sur des « Compute Device ». La figure 18 présente une vue de cette exécution. Tout comme pour les threads CUDA, chaque « Processing Element » dispose de coordonnées et d'un identifiant lui permettant d'effectuer des calculs sur une partie des données globales. Tout comme en CUDA, il est possible d'explicitement utiliser une mémoire spécifique du Compute Device, telle que la mémoire locale, globale ou autre. La figure 19 présente les différents niveaux de mémoire explicitement atteignables en OpenCL.

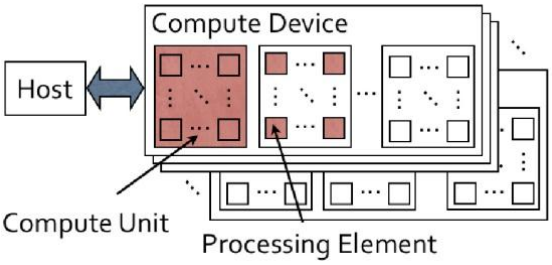


Figure 18. Principe du fonctionnement d'OpenCL.

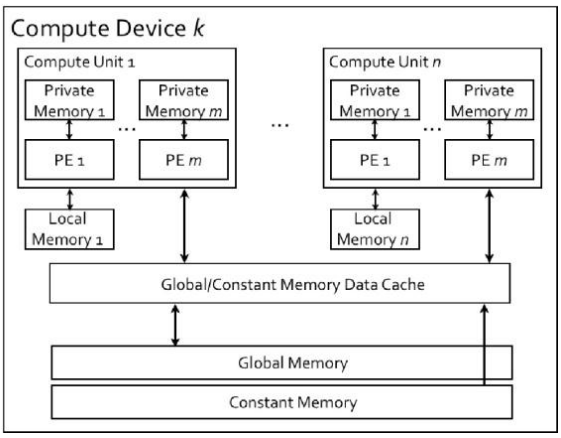


Figure 19. Hiérarchie mémoire en OpenCL.

1.2.4. Programmation hybride

Le terme « programmation hybride » a plusieurs significations en calcul parallèle. Il peut représenter l'utilisation conjointe d'OpenMP et MPI, c'est-à-dire l'utilisation hybride de la parallélisation en mémoire partagée et distribuée. Le but est de tenir compte des spécificités mémoires de la machine, et ainsi d'être le plus efficace possible et donc le plus performant. Dans le papier (10), l'auteur montre que pour des problèmes limités par la bande passante mémoire MPI est assez efficace, et que l'on peut gagner en performance en mêlant les deux styles de programmation lorsque le nombre de nœuds devient grand. Ces paramètres dépendent en réalité de la vitesse des processeurs, de la mémoire et la bande passante réseau disponible tout comme les latences offertes.

L'autre sens du terme « programmation hybride » est l'utilisation de plusieurs types de matériels, tels que des multicoeurs et des accélérateurs. La manière classique de réaliser cette approche est d'appeler les langages de chaque matériel pour exécuter les noyaux de calcul à partir de l'hôte dont le programme principal est écrit de manière usuelle. Ainsi, utiliser des multicoeurs et des GPUs Nvidia pourrait se réaliser avec un programme en C/C++ (hôte) faisant appel à OpenMP (multicoeurs) et CUDA (GPUs). Les difficultés liées à cette approche sont évidentes : maintenir autant de langages que de types de matériel, au sein du même projet global.

1.2.4.1. OpenCL

Plusieurs alternatives existent pour assouplir la programmation hybride. La première est OpenCL, présenté dans la section précédente. En effet, OpenCL adresse autant le parallélisme massif que le parallélisme de tâches, et donc la majorité des matériels de calcul existant. Ainsi, avec un seul langage, il est possible de programmer multicoeurs et accélérateurs. Les autres solutions sont l'utilisation de directives de parallélisation pour indiquer quelles portions du code sont exécutées en parallèle sur quel multicoeur ou accélérateur.

1.2.4.2. HMPP

Les directives Hybrid Multicore Parallel Programming (HMPP) sont un produit commercial de CAPS Entreprise, dont la philosophie est « Pour adresser le monde du multi-cœurs hybride le développeur doit annoter son application et non pas la modifier. ». Cette philosophie est finalement assez proche de celle d'OpenMP pour les multicoeurs, avec quelques différences notables. Par défaut, lorsqu'une section est annotée pour être exécutée en parallèle sur GPU par exemple, un « codelet » est généré. Ce codelet représente le noyau de calcul généré automatiquement par le framework HMPP. Lors de l'exécution du programme, ce codelet sera exécuté sur GPU si possible, et sinon le codelet CPU sera appelé sur processeur standard. La structure de HMPP permet de modifier et optimiser les codelets pour une performance optimale sur le matériel cible.

1.2.4.3. XMP

Le but premier du framework Xscalable MP (XMP) est de fournir une alternative à MPI pour l'utilisation de machines à mémoire distribuée, au travers de directives de compilation. Le compilateur effectue alors une transformation du code source d'origine en code source contenant des appels à la librairie MPI. Ainsi, le parallélisme de type mémoire distribuée est atteignable à partir de code séquentiel. Actuellement, le consortium à l'origine d'XMP travaille à l'ajout de directives pour accélérateurs tels que les GPUs au framework XMP. La

manière de fonctionner est proche de HMPP, mais peu de détails sont disponibles sur les possibilités finales d'XMP.

1.2.4.4. OpenMP

La constructeur de superordinateur Cray travaille sur une série d'extensions aux directives OpenMP pour ses prochaines machines Pétaflopiques contenant des accélérateurs GPUs. Parallèlement, le consortium à l'origine d'OpenMP évalue également l'intérêt d'ajouter ces directives pour accélérateurs. Quelques papiers présentés dans (11), tel que (12), montrent quelques expérimentations d'extensions GPU pour OpenMP.

1.3. Problématique générale de la performance parallèle

Lors de l'exécution parallèle de calculs scientifiques, plusieurs paramètres peuvent avoir une influence sur la performance finale de l'application. Ces paramètres sont liés essentiellement à deux tâches : calcul et communication.

1.3.1. Puissance de calcul

Le calcul représente les opérations arithmétiques classiques (addition, soustraction, multiplication, division) à la base de calculs plus complexes. La capacité de calcul est limitée par la puissance théorique de la machine, que l'on exprime en opérations flottantes par secondes, les Flops. Étant données les capacités actuelles des machines, l'unité élémentaire pour exprimer la puissance de calcul d'une machine standard est le GigaFlops (10^9 Flops). Lorsque l'on passe à l'échelle d'un supercalculateur, on parle plutôt de TeraFlops (10^{12} Flops), voire de PetaFlops (10^{15}) pour les meilleurs d'entre eux. Cette limite représente la performance maximale que peut atteindre un algorithme très intense en calculs.

1.3.2. Communications

Les communications englobent les opérations mémoires de toutes sortes, autant les accès au sein de la mémoire principale d'une machine que les transferts entre machines. Deux types de performances sont importants lors d'accès mémoire : la bande passante et la latence. Comme nous avons pu voir dans la section 1.1.2.1, les différents niveaux de mémoires au sein d'un multicoeur ont des performances et des tailles différentes. Le tableau 3 donne quelques indications à ce sujet.

Tableau 3. Bande passante mémoire et latence de la hiérarchie mémoire d'une machine de calcul. * La bande passante réseau est très dépendante de la technologie utilisée. En général, l'ordre est d'une dizaine de Go/s.

	Bande passante	Latence	Taille
Caches et registres	O(100) Go/s	(0/1)-10 ns	8-12 Mo et <1Ko
Mémoire principale	O(10) Go/s	10^2 ns	10-100 Go
Stockage SSD	O(0.3) Go/s	10^5 ns	10 – 1000 Go
Stockage classique	O(0.1) Go/s	10^7 ns	100 – 2000 Go
PCI-Express	O(1-30) Go/s	10.10^3 - 15.10^3 ns	-
Réseau local	O(1-100) Go/s*	10^6 - 10^8 ns	-
Réseau distant	O(1-100) Go/s*	10^9 ns	-

Dans une optique de recherche de la meilleure performance, l'idéal est de réussir à utiliser les mémoires les plus rapides disponibles. Cependant, à cause des tailles de ces dernières, il n'est pas toujours possible de placer toutes les données utiles dans le niveau le

plus performant. Cette possibilité dépend également de la densité de calcul des algorithmes utilisés.

1.3.3. Illustrations

Pour illustrer comment ces deux paramètres limitent la performance lors de calculs, prenons une machine représentative des architectures grand public, qui aurait une bande passante de 30 Go/s et une puissance crête double précision de 100 GFlops.

1.3.3.1. Limité par la bande passante mémoire

En prenant l'exemple de l'addition de deux vecteurs de taille n dans un troisième, alors 2n données seront lues et n données écrites. Le nombre d'opérations effectuées au total sera de n, et l'on effectuera donc 1 opération pour 3 données. Le ratio calculs/communications est donc de 1/3. Autrement dit, il faudra charger ou écrire trois données pour effectuer une opération flottante. Ce problème est limité par la bande passante mémoire, et les performances en flops que l'on peut espérer atteindre à partir d'une mémoire principale fournissant un débit de 30 Go/s sont calculées ainsi :

1. Avec 30 Go/s, on peut transférer 30 / 8 GigaDouble/s, soit 3.75 GDouble/s
2. Il faut 3 nombres pour effectuer un calcul, et on peut donc effectuer 3.75 / 3 GFlops, soit 1.25 GFlops

Malgré une puissance crête de 100 GFlops par secondes, l'application ne pourra développer au mieux que 1.25 GFlops si les données sont situées dans la mémoire principale. Si elles sont assez petites pour tenir dans les caches du processeur, alors la bande passante est plutôt de 300 Go/s, avec une performance finale de 12.5 GFlops. C'est mieux, mais encore bien inférieur la performance crête de la machine. Lorsqu'un algorithme est ainsi limité par la bande passante mémoire et que la limite de performance liée à cette bande passante est atteinte, alors il est inutile d'essayer d'utiliser des techniques de double buffering ou de pipelining pour améliorer les performances : elles sont déjà à leur maximum.

Très souvent, les algorithmes effectuant des opérations entre vecteurs ou entre matrices et vecteurs sont limités par la bande passante mémoire. C'est le cas des solveurs itératifs de Krylov que nous étudierons durant cette thèse, tout comme de la plupart des solveurs neutroniques déterministes.

1.3.3.2. Limité par la puissance de calcul

En prenant l'exemple du produit de deux matrices carrées d'ordre n dans une troisième, alors on effectue $2n^3$ opérations pour $3n^2$ lectures écritures mémoires. Le ratio calculs/communications est supérieur à 1, de l'ordre de $^2/_3\, n$. Cela implique qu'en moyenne, avec une donnée on peut effectuer $^2/_3\, n$ opérations. Autrement dit, il peut être intéressant de réussir à placer quelques données en cache pour ensuite effectuer les calculs. Si l'on prend une bande passante de 300 Go/s, alors pour traiter le problème de la multiplication des matrices, il faudra lire deux matrices et écrire la troisième. Cela implique $3n^2$ transferts de taille 8 octets (double précision). D'où $24n^2$ octets à transférer. Le temps nécessaire pour les transférer est de $24n^2 / (300.10^9)$ s. Concernant les calculs, $2n^3$ opérations sont à effectuer, et à raison d'une puissance de 100 GFlops, alors il faudra $2n^3 / (100.10^9)$ s pour effectuer le calcul. L'opération la plus coûteuse est le calcul ici, car le cas inverse ne se produit que si :

1. $2n^3 / (100.10^9) < 24n^2 / (300.10^9)$,
2. $n / (100.10^9) < 12 / (300.10^9)$,
3. $n < 12 / 3$,

4. et $n < 4$.

Donc, si le problème est suffisamment grand, n domine le calcul. Pour une taille de 1000 par 1000, le temps de communication serait de 0,00008s en mémoire cache, ou encore 0,0008s en mémoire principale. Le temps de calcul, lui, est de 0,02s. Dans le cadre de calculs limités par la puissance de calcul, alors il est nécessaire d'optimiser l'utilisation de la mémoire par des techniques de cache-blocking, c'est-à-dire de placement de données dans le cache par blocs, et donc de double buffering, et autres optimisations.

Quelques exemples d'applications limitées par la puissance de calcul incluent les opérations entre matrices de type produit matrice-matrice, les calculs Montecarlo ou les solveurs non-déterministes utilisant ces techniques, le cryptage ou décryptage de données, etc.

Conclusion

Dans ce chapitre d'introduction des différentes problématiques motivant ces travaux de thèse, nous avons vu successivement l'évolution du matériel de calcul pour les simulations numériques, ainsi que l'évolution des langages permettant de les utiliser, et finalement les problématiques du calcul hautes performances.

La performance a longtemps été dictée par l'augmentation de la performance brute liée à la fréquence d'horloge grandissante des processeurs, mais ce gain « gratuit » s'est arrêté face au « power wall », pour des raisons de consommations énergétiques et d'enveloppe thermique. La performance est maintenant dictée par l'utilisation du parallélisme au sein des processeurs à plusieurs cœurs, mais également par spécialisation de noyaux pour des matériels accélérateurs tels que les GPUs ou le Cell.

L'une des problématiques majeures des codes scientifiques est la maintenabilité, conditionnée par la portabilité et la performance. Ces critères sont perpétuellement contraints par un matériel évoluant très vite, tel que les machines monocoeur, puis vectorielles, multicoeurs et accélérées, et un code utilisant un langage relativement fixé. L'exemple du code APOLLO3® (voir le dernier chapitre) faisant appel au C++ et des modules Fortran illustre cette problématique. En particulier, ce code est développé depuis le début des années 2000, soit une dizaine d'années, et devrait être finalisé dans les prochaines années. Son utilisation en production est de l'ordre d'une dizaine d'années. Sur les 12-15 ans à venir, il est difficile de prédire comment évolueront les architectures matérielles, et le code doit donc être capable de pallier à ces changements parfois brutaux en offrant une bonne souplesse de développement tout en restant très performant et multifilières.

Ces aspects de performance, portabilité et évolutivité sont au cœur des travaux de cette thèse, développés dans les chapitres suivants.

Chapitre 2. Résolution parallèle de problèmes à valeurs propres non-Hermitiens

Sommaire du chapitre

Introduction

Le but de ce chapitre est de présenter le problème du calcul de valeurs propres de manière générale, puis de préciser les méthodes usuelles pour le résoudre. Nous centrons particulièrement la réflexion sur les méthodes itératives, et en particulier celles utilisant des espaces de Krylov. Nous présentons ensuite les manières usuelles de les paralléliser, et notamment quelques librairies connues en algèbre linéaire.

2.1. Problème numérique

L'une des classes de problèmes les plus connues en calcul scientifique est la résolution de problèmes à valeurs propres (13; 14; 15; 16). On la retrouve dans beaucoup de domaines tels que l'aéronautique, l'énergie, le bâtiment ou la mécanique quantique. Pour l'aéronautique par exemple, la stabilité d'un avion est déterminée par la position dans le plan complexe des valeurs propres d'une certaine matrice. Dans le domaine de l'énergie nucléaire, le bilan neutronique d'un cœur de réacteur peut-être obtenu en résolvant un problème de valeur propre dominante. Les problèmes traités peuvent utiliser différentes représentations pour leurs données. Une manière usuelle, que l'on étudiera dans le cadre de cette thèse, est l'utilisation de matrices assemblées de manière explicite : les termes de la matrice sont connus et explicitement stockés, exceptés les éléments nuls dans certains cas. Des vecteurs et des scalaires font également partie des structures mathématiques de base utilisées. Ce choix de formalisation du problème n'empêche pas l'application des résultats de cette thèse à des domaines utilisant des matrices non assemblées explicitement, c'est-à-dire dont les coefficients sont calculés à la volée par exemple. Quelques adaptations seront peut-être nécessaires, pour des résultats du même ordre.

2.1.1. Problème à valeur propre

Pour une matrice carrée A d'ordre n, l'équation à résoudre pour obtenir les valeurs propres et vecteurs propres associés est la suivante :

$$Av_i = \lambda_i v_i \, , \lambda_i \in \mathbb{C} \; et \; v_i \in \; \mathbb{C}^n \qquad (1)$$

Dans l'équation (1), (λ_i, v_i) représente un couple d'une valeur propre λ_i et un de ses vecteurs propres associés v_i. En effet pour une valeur propre donnée, plusieurs vecteurs propres différents peuvent exister, voir une infinité.

De manière générale, pour résoudre (1), on calcule d'abord les valeurs propres λ_i, puis l'équation suivante est résolue, où *Id* représente la matrice identité de C^n :

$$Det(Av_i - \lambda_i Id) = 0 \qquad (2)$$

Ces étapes montrent la manière formelle de résoudre un problème à valeur propre, qui n'est pas nécessairement optimale. En effet, pour chaque vecteur propre, un calcul de déterminant est nécessaire, opération très coûteuse en nombre d'opérations, et présentant des instabilités numériques. Différentes méthodes numériques existent pour résoudre ce problème, présentées dans la partie suivante.

2.1.2. Méthodes numériques

Pour résoudre l'équation (1), deux grandes classes de méthodes existent :

1. Méthodes directes
2. Méthodes itératives

2.1.2.1. Méthodes directes

Les méthodes directes permettent la résolution d'un problème en un nombre fini d'étapes. En arithmétique exacte, ces méthodes donneraient la solution exacte du problème. En termes de nombre d'opérations, les méthodes directes demandent souvent beaucoup plus de calculs qu'une méthode itérative. En effet, les méthodes directes résolvent exactement le problème, avec une précision maximale. Les solveurs itératifs effectuent une approximation, et dans certains cas un résultat à 10^{-5} est suffisant. Cela demande bien moins de calculs qu'un résultat convergé à 10^{-10}, et encore moins que le calcul d'une solution exacte. Pour les

problèmes à valeurs propres, ces méthodes directes effectuent en général les étapes suivantes (17) :

1. Trouver une matrice orthogonale Q telle que Q* A Q = T, avec T tridiagonale
2. Calculer la décomposition propre de la matrice tridiagonale T

Une matrice orthogonale est une matrice carrée satisfaisant la relation : A . A^t = I et la décomposition propre d'une matrice est la résolution de l'équation (1). Des exemples de méthodes directes pour les problèmes à valeurs propres sont la méthode de tridiagonalisation suivie de la méthode de la bissection ou divide-and-conquer (18) pour les matrices symétriques ou hermitiennes.

2.1.2.2. Méthodes itératives

Les méthodes itératives (19) fonctionnent de manière complètement différente par rapport aux méthodes directes. Elles utilisent une solution initiale, puis effectuent des approximations successives afin de converger vers une solution exacte. Des conditions d'arrêt sont nécessaires pour arrêter la méthode, et bien souvent elles sont constituées de deux conditions : le nombre d'itérations maximum et le seuil de précision à atteindre. Le premier critère est présent pour borner le temps effectif de résolution et parer aux éventuels problèmes de convergence lente ou asymptotique. Le second critère spécifie l'erreur maximale pour considérer que la solution calculée est satisfaisante. Pour évaluer cette précision, on utilise une quantité appelée résidu. Par exemple, on peut calculer $\|Av_i - \lambda_i v_i\|$ pour obtenir le résidu lié à l'équation (1). Ainsi, en prenant en compte ces considérations, l'algorithme générique d'une méthode itérative serait le suivant :

Tableau 4. Algorithme générique d'une méthode itérative

s = solution approchée initiale
seuil = précision souhaitée
tant que nb_iterations < maximum_iterations et résidu > seuil
  1. approximer une nouvelle solution en utilisant les données de l'itération précédente
  2. calculer le résidu de la nouvelle solution
fin tant que

Contrairement aux méthodes directes, les méthodes itératives ne donnent pas nécessairement la solution exacte du problème en un nombre fini d'étapes, même si l'on avait accès à une précision arithmétique exacte. Par contre ces méthodes sont généralement moins coûteuses pour obtenir une solution viable à un problème, et en particulier nécessaires lorsque l'on cherche à résoudre des problèmes de très grandes tailles. Quelques exemples de méthodes purement itératives appliquées aux problèmes à valeurs propres sont la méthode QR (20), la méthode de la Puissance (21), la méthode de Lanczos (22) (23) ou la méthode d'Arnoldi (24). Il existe quelques différences entre ces méthodes. La méthode QR calcule toutes les valeurs propres d'une matrice A, là où les autres méthodes n'en calculent qu'une partie. Nous nous intéressons plus particulièrement à cette catégorie de méthode.

2.1.2.3. Méthodes itératives pour le calcul d'un sous-ensemble de valeurs propres

Plusieurs méthodes de calcul de valeurs propres ont été évoquées dans le paragraphe précédent : Puissance, Lanczos et Arnoldi. Ces méthodes ont des caractéristiques et algorithmes différents, mais peuvent atteindre le même but. La méthode de la Puissance

calcule la valeur propre dominante d'une matrice, là où Lanczos et Arnoldi calcule un sous-ensemble arbitraire de valeurs propres de la matrice. D'autres méthodes existent, telles que Jacobi-Davidson ou la méthode de Ricatti. Dans le cadre de cette thèse, nous nous concentrerons essentiellement sur les méthodes de la Puissance et Arnoldi.

2.1.2.3.1. Méthode de la Puissance

La méthode de la Puissance est extrêmement simple à mettre en place, et assure la convergence de l'algorithme vers la valeur propre dominante d'une matrice diagonalisable fournie en entrée. Son algorithme utilise le produit matrice-vecteur et une opération de normalisation de vecteur. Le choix de la norme est arbitraire. L'algorithme général de la méthode de la Puissance est décrit dans le tableau suivant :

Tableau 5. Algorithme de la méthode de la Puissance

entrée:

- matrice A de taille nxn
- vecteur initial v_0 de taille n
- critères de convergence : seuil, max_iterations

sortie :

- valeur et vecteur propre dominante : λ_{dom} et v_{dom}

tant que résidu > seuil et nb_iterations < max_iterations faire

1. $v_i = A \cdot v_{i-1}$
2. $\lambda_i = norm(v_i)$
3. $vi = vi / \lambda_i$
4. résidu = $| \lambda_i - \lambda_{i-1}|$

Fin tant que

$\lambda_{dom} = \lambda_{nb\_iterations}$

$v_{dom} = v_{nb\_iterations}$

La convergence de cette méthode dépend du coefficient $\lambda_{dom} / \lambda_{dom2}$, où λ_{dom2} est la deuxième plus grande valeur propre absolue. Plus le coefficient est proche de 1, plus la méthode aura un taux de convergence faible, et nécessitera un grand nombre d'itérations.

Cette méthode est une méthode de référence pour de nombreux problèmes ayant des enjeux majeurs, comme dans le domaine de la physique des réacteurs où la recherche du coefficient de criticité d'un cœur de réacteur nécessite la résolution de l'équation de transport des neutrons. Elle est notamment utilisée dans le solveur MINOS (25) du code APOLLO3® développé par le CEA. Un autre exemple est le calcul de l'indice *pagerank* du moteur de recherche internet Google, permettant le classement des pages lors de l'affichage des résultats d'une recherche. Étant donné les volumes d'informations traités, et potentiellement les tailles de matrice à résoudre, une méthode itérative telle que la méthode de la puissance est nécessaire et efficace.

2.1.2.3.2. Méthodes de Lanczos et Arnoldi

Le but de la méthode de Lanczos (22; 23; 26) et de celle d'Arnoldi (24) est de calculer un sous-ensemble des valeurs propres / vecteurs propres de la matrice problème. Lanczos ne s'applique qu'aux matrices symétriques ou hermitiennes, et Arnoldi reste plus général et résout les problèmes non symétriques ou non-hermitiens.

Ces deux méthodes ont pour base la méthode de la puissance, et l'améliore en utilisant plus les informations générées à chaque étape. En effet, la méthode de la puissance calcule plusieurs vecteurs jusqu'à convergence à une étape p. On peut donc voir cette méthode

comme la création d'une matrice [Av_0, A^2v_0, …, A^pv_0] dont A^pv_0 est le vecteur propre dominant. L'information utilisée pour une étape i est celle de l'étape précédente i-1, et pas celles des étapes 1 à i-2. Lanczos et Arnoldi propose d'effectuer une orthogonalisation des vecteurs des étapes 1 à p, pour extraire le plus d'information possible des étapes précédentes pour l'étape courante.

Ces deux méthodes calculent les matrices H et V telles que l'on a la relation :

$$H = V^* AV \quad (3)$$

H est une matrice de taille m+1 lignes par m colonnes, et V possède m+1 colonnes de taille n, le nombre de lignes de A. Si l'on utilise Lanczos, alors H est tridiagonale, et de ce fait souvent appelée T. Dans le cas d'Arnoldi, la matrice H est de forme Hessenberg supérieure, c'est-à-dire triangulaire supérieure avec des éléments sous la diagonale. Dans les deux cas, V doit être orthonormale, ce qui implique que ses vecteurs sont normés, et orthogonaux. Il faut prendre en considération l'arithmétique finie des ordinateurs, qui a une influence sur l'orthogonalité de la base V. En arithmétique exacte, V est forcément orthogonale suivant les algorithmes d'orthogonalisation que l'on utilise. En précision finie, l'algorithme utilisé tout comme le choix de l'orthogonalisation ont un impact important sur la précision des valeurs propres et vecteurs propres calculés (27).

(i) Matrices symétriques

Pour calculer T et V par l'algorithme de Lanczos, on effectue les opérations suivantes :

Tableau 6. Algorithme de Lanczos

entrée:
- matrice symétrique A de taille nxn
- vecteur v1 normalisé
- nombre de valeurs propres désiré

sortie :
- α_j et β_j qui constituent la matrice T

$v0 = 0$
$\beta 1 = 0$
pour j de 1 à m
faire
1. $w_j = Av_j$
2. $\alpha_j = (w_j , v_j)$
3. $w_j = w_j – \beta_j.v_{j-1} – \alpha_j.v_j$
4. $\beta_{j+1} = ||w_j||$
5. $v_{j+1} = w_j / \beta_{j+1}$

Fait

Les coefficients α_j et β_j permettent de constituer la matrice T suivant le motif de la figure 20.

$$\mathbf{T} = \begin{pmatrix} \alpha 1 & \beta 2 & & & & \\ \beta 2 & \alpha 2 & \beta 3 & \mathbf{0} & & \\ & \beta 3 & \ddots & \ddots & \beta m\text{-}1 & \\ & & \ddots & \beta m\text{-}1 & \alpha m\text{-}1 & \beta m \\ \mathbf{0} & & & & \beta m & \alpha_m \end{pmatrix}$$

Figure 20. Constitution de la matrice T après exécution de l'algorithme de Lanczos.

On peut ensuite résoudre le problème à valeur propre d'ordre m de la matrice T pour calculer les couples de valeur propre, vecteur propre (λ_i, z_i). Les valeurs propres calculées

correspondent à des approximations des valeurs propres de la matrice A, et les vecteurs propres z_i de taille m<n nécessitent une transformation avant de pouvoir être vu comme des approximations des vecteurs propres de A. Pour ce faire, on calcule l'approximation de vecteur propre $y_i = Vz_i$, où V = [v1, … , vm], et ensuite le résidu peut être calculé. Si la précision atteinte est suffisante, on arrête l'algorithme, sinon plusieurs solutions sont possibles. Premièrement, on peut ré-exécuter une itération de Lanczos, avec un sous espace plus grand, ou encore redémarrer un Lanczos de même taille avec un vecteur initial v1 constitué d'une combinaison linéaire des vecteurs propres de l'itération précédente (28). On appelle ce redémarrage un redémarrage explicite de la méthode. Une dernière solution est le redémarrage implicite proposé dans (26).

(ii) Matrices non symétriques

Dans le cas des matrices non symétriques, on peut appliquer la méthode d'Arnoldi. Les mêmes principes que pour la méthode de Lanczos sont suivis, avec :

1. Une étape de réduction permettant de calculer les matrices H et V de l'équation 3. Cette étape est appelée réduction d'Arnoldi ou orthogonalisation.
2. Le problème à valeurs propres est résolu dans H.
3. Les vecteurs propres de H sont projetés grâce à la matrice V pour obtenir les vecteurs de Ritz de la matrice A, une approximation de quelques-uns de ses vecteurs propres.

1. Orthogonalisation

Pour l'orthogonalisation, plusieurs schémas peuvent être sélectionnés avec chacun ses avantages et ses inconvénients. Les schémas les plus connus sont l'orthogonalisation de Gram-Schmidt, les transformations de Householder ou les rotations de Givens. Dans le cadre de cette thèse, nous nous concentrerons sur la version Gram-Schmidt, même si les autres versions restent également valables.

Parmi les orthogonalisations de Gram-Schmidt (29), on distingue 3 grandes variantes : l'orthogonalisation classique (CGS), l'orthogonalisation modifiée (MGS) et Gram-Schmidt avec réorthogonalisation (CGSr). Les algorithmes de chaque orthogonalisation sont présentés dans les tableaux suivants. « v_i » est le i-ème vecteur colonne de V, que l'on pourrait noter V(: , i) soit toutes les lignes de la colonne i. A l'inverse, V(i , :) représenterait toutes les colonnes de la ligne i et donc le i-ème vecteur ligne de V. On notera H(1 :i , i) les éléments 1 à i de la i-ème colonne de H. L'opérateur « * » représente la transposée d'une matrice réelle ou la matrice adjointe dans le cas complexe.

D'un point de vue performances, l'un des intérêts des orthogonalisations CGS* par rapport à MGS est leur parallélisme intrinsèque. Les opérations sont effectuées sur des matrices denses et des vecteurs, ce qui ajoute une bonne régularité des données. Dans le cas de la méthode MGS, les opérations portant sur V et H sont effectuées entre vecteurs, et dépendantes de l'étape précédente. On a donc une moins bonne parallélisation intrinsèque de cette orthogonalisation.

Concernant la précision numérique, ces trois orthogonalisations ne retournent pas la même précision sur l'orthogonalité de la base V construite, point critique pour la précision des valeurs/vecteurs propres calculés. Ainsi, la plus précise est CGSr, suivie de MGS puis CGS. Le chapitre 3 explicitera ces différences ainsi que l'influence de la précision du matériel sur cette orthogonalité.

Tableau 7. Algorithme de la Réduction d'Arnoldi par orthogonalisation de Gram-Schmidt Classique (gauche) ou avec réorthogonalisation (droite)

Entrée:	Sortie :
- Vecteur v0 initial - Matrice A_{nxn}	- Matrices H_{m+1xm} et V_{nxm+1}

Gram-Schmidt Classique (CGS)

$v0 = v0 / ||v0||_2$
Pour i de 1 à m
Faire
1. $vi = A.v_{i-1}$
2. $H(1:i , i) = V(: , 0:i-1)*.vi$
3. $vi = vi - V(: , 0:i-1) . H(1:i , i)$
4. $H(i , i+1) = ||vi||_2$
5. si $H(I , i+1) = 0$
   alors stop
6. fin si
7. $vi = vi / H(i , i+1)$

Fait

Gram-Schmidt Classique avec réorthogonalisation (CGSr)

$v0 = v0 / ||v0||_2$
Pour i de 1 à m
Faire
1. $vi = A.v_{i-1}$
2. $H(1:i , i) = V(: , 0:i-1)*.vi$
3. $vi = vi - V(: , 0:i-1) . H(1:i , i)$
4. $C(1:i , i) = V(: , 0:i-1)*.vi$
5. $vi = vi - V(: , 0:i-1) . C(1:i , i)$
6. $H(i , i+1) = ||vi||_2$
7. si $H(I , i+1) = 0$
   alors stop
8. fin si
9. $vi = vi / H(i , i+1)$

Fait
$H = H + C$

Tableau 8. Algorithme de la Réduction d'Arnoldi par orthogonalisation de Gram-Schmidt Modifié (MGS).

Gram-Schmidt Modifié (MGS)

Entrée :
- vecteur v0 initial
- matrice A_{nxn}

Sortie :
- Matrices H_{m+1xm} et V_{nxm+1}

$v0 = v0 / ||v0||_2$
Pour i de 1 à m
Faire
1. $vi = A.v_{i-1}$
2. pour j de 1 à i
3. faire
   a. $H(j , i) = v_j*.v_i$
   b. $vi = vi - v_j . H(j , i)$
4. fait
5. $H(i , i+1) = ||vi||_2$
6. si $H(I , i+1) = 0$
   alors stop
7. fin si
8. $vi = vi / H(i , i+1)$

Fait

Les méthodes CGS et CGSr sont très proches dans leur algorithmique, et la différence essentielle se situe dans la réorthogonalisation du vecteur colonne de V par chaque étape de la réduction d'Arnoldi. On peut voir la réorthogonalisation aux lignes 4 et 5 et la prise en compte des données créées par cette dernière en dernière ligne de l'algorithme de CGSr. CGSr effectue plus d'opérations que CGS, comme le montre le tableau 9.

Tableau 9. Détail du nombre d'opérations flottantes (flop) nécessaire pour chaque orthogonalisation. Le nombre de non zeros de A est appelée nzA. Pour une matrice dense carrée d'ordre n, cela représente n^2.

Orthogonalisation	CGS	MGS	CGSr	Flop par op.
Matrix vector (A)	1. m	1. m	1. m	2 non zeros A
Axpys	3. m(m+1)/2	3.b. m(m+1)/2	3+5. m(m+1)	2 n
Dots	2. m(m+1)/2	3.a. m(m+1)/2	2+4. m(m+1)	2 n
Scal	7. m	8. m	9. m	n
Norm 2	4. m	6. m	5. m	2 n
Total flop	m(2nzA + m + 3)	m(2nzA + m + 3)	m(2nzA + 2m + 4)	N/A

2. Résolution du problème à valeurs propres et projection

Une fois les matrices H et V constituées, les valeurs propres et vecteurs propres de H sont calculés en utilisant une méthode numérique au choix. L'idéal est d'utiliser une méthode prenant en compte la forme Hessenberg de la matrice H. La factorisation de Schur proposée dans la méthode *hseqr* de LAPACK est une bonne option. Elle calcule la forme de Schur de la matrice H, telle que $H=STS^T$. La matrice S contient les vecteurs de Schur de H, et la matrice T tridiagonale, contient les valeurs propres de H sur sa diagonale. Un appel à la méthode de la puissance inverse permet par la suite de retrouver les vecteurs propres de H à partir de ses valeurs propres. Ensuite, pour obtenir les vecteurs propres associés aux valeurs propres de A, on utilise la matrice V pour projeter les vecteurs propres de la même manière que pour la méthode de Lanczos : $y_i = Vz_i$, avec z_i vecteur propre de H.

On peut alors évaluer le résidu ε_i lié à chaque couple de valeur propre λ_i et vecteur propre y_i suivant la formule suivante :

$$\mathcal{E}_i = \|(A - \lambda_i Id)y_i\|$$

Ce résidu peut également être calculé suivant la formule analytique proposée par Saad dans (30; 31), qui ne fait intervenir que le produit de deux scalaires et borne le résidu. La formule est alors :

$$\mathcal{E}_i = |H(m+1, m)y_i(m)|$$

Si ce résidu est supérieur au seuil fixé, alors la méthode doit être redémarrée.

3. Arnoldi itéré

La méthode d'Arnoldi peut être itérée implicitement et explicitement. La manière implicite a été proposée par Sorensen (32). Elle considère l'utilisation efficace de l'information liée aux valeurs propres intéressantes pour le problème, et la non-prise en compte de l'information des valeurs propres non désirées. La convergence est meilleure dans le cas implicite et demande en général moins d'itérations qu'un Arnoldi explicitement redémarré. Cependant le redémarrage explicite offre des possibilités intéressantes en termes d'hybridation comme nous le verrons au chapitre 5. La version explicite a été proposée par Saad (31) (31), et offre l'avantage d'un schéma de redémarrage simplifié. Une combinaison linéaire des vecteurs de Ritz de l'itération précédente est utilisée pour constituer le vecteur initial de l'itération suivante. Ainsi, la convergence est forcée vers le sous-espace de Krylov contenant les valeurs propres désirées. L'algorithme de la méthode d'Arnoldi explicitement redémarrée (ERAM) peut s'écrire :

Tableau 10. L'algorithme de la méthode d'Arnoldi explicitement redémarrée (ERAM)

Méthode d'Arnoldi explicitement redémarrée (ERAM)	
Entrée : - vecteur v0 initial - matrice A_{nxn} - critères de convergence : seuil ε, nombre d'itérations maximum iter_max	Sortie : - couples valeurs/vecteurs propres (λ_i, w_i)
iter = 0 tant que iter < iter_max et residu > ε faire 1. effectuer une réduction d'Arnoldi : A = V*HV 2. résoudre le problème à valeurs propres de H : $H.y_i = \lambda_i.y_i$ 3. projeter les vecteurs propres y_i : $wi = V.y_i$ 4. mettre à jour le résidu fait	

2.2. Résolution parallèle

La résolution de problèmes à valeurs propres est souvent une partie très importante du temps de calcul des applications scientifiques. Ainsi certains problèmes de grandes tailles nécessitent une puissance de calcul supérieure à celle qu'offre une machine de calcul locale. On utilise alors des grappes de calcul ou cluster pour résoudre ces problèmes dans un temps acceptable. Ces grappes font travailler plusieurs unités de calcul en parallèle, et il est nécessaire d'adapter les méthodes à cette manière de calculer. De plus, comme nous avons vu dans le chapitre 1, les machines de travail actuelles tirent leur puissance de la parallélisation plus que de la puissance brute d'une unité de calcul unique. La parallélisation est donc aussi intéressante pour utiliser toute la puissance de calcul d'une machine de travail locale.

Dans le cadre des méthodes itératives pour résoudre des systèmes linéaires ou des problèmes à valeurs propres, l'étape la plus consommatrice en calcul est le produit matrice vecteur. Pour des matrices denses, il représente souvent plus de 90% du temps de calcul complet de la méthode, et dans le cas creux bien souvent plus de 50%. Ainsi il s'agit d'un noyau de calcul crucial dans l'étape de parallélisation, et il nous servira d'illustration sur la manière d'adapter une méthode séquentielle à du calcul parallèle.

Pour utiliser les architectures parallèles récentes telles que les multicœurs ou les accélérateurs, plusieurs solutions existent. On peut en dénombrer trois, qui sont :

1. Parallélisation des noyaux de calculs existant
2. Utilisation de noyaux de calculs parallèles
3. Appel à des bibliothèques parallèles

Les deux premières solutions sont proches en termes d'application au code, mais ont des contraintes différentes.

2.2.1. Adaptation des méthodes numériques

Pour illustrer les solutions utilisant des noyaux de calculs parallèles, nous pouvons prendre un exemple simple : la parallélisation du produit matrice-vecteur. C'est un noyau de calcul bien connu et très utilisé pour la résolution itérative de systèmes linéaires ou de

problèmes à valeurs propres, et nous allons voir comment il peut être parallélisé suivant les trois points précédemment énoncés.

La définition mathématique du produit matrice vecteur w = A.v, avec w vecteur de taille m, v vecteur de taille n et A une matrice de taille m lignes par n colonnes, est la suivante :

$$w = \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij}.\mathrm{v_j})$$

Le produit matrice-vecteur peut être implémenté par lignes ou par colonnes. Dans le cas du produit par lignes, on effectue une succession de m produits scalaires que l'on nommera « dot » :

$$w_i = \sum_{i=1}^{m} \mathrm{dot}(\,\mathrm{v}\,, \text{ième ligne de A})$$

Si l'on choisit l'implémentation par colonnes, alors on effectue une succession d'additions vectorielles dont l'un des vecteurs est multiplié par un scalaire. Cette opération est assez connue, et nommée axpy. Elle effectue l'opération vectorielle y = y + alpha x, où y et x sont des vecteurs et alpha un scalaire. Nous l'écrirons axpy(y,alpha,x). D'où l'algorithme par colonne du produit matrice vecteur :

$$w = \sum_{j=1}^{n} \mathrm{axpy}(\,\mathrm{w}, \mathrm{vj}\,, \text{jème colonne de A})$$

On peut donc paralléliser le produit matrice vecteur de plusieurs manières selon l'approche choisie : lignes ou colonnes.

2.2.1.1. Parallélisation à grain fin

Si l'on regarde l'utilisation de machine à mémoire partagée ou d'accélérateurs, alors une manière naturelle de paralléliser le produit matrice-vecteur par lignes ou colonnes est de paralléliser chaque opération vectorielle. Ainsi, on répartit la charge de calcul de chaque produit scalaire ou axpy sur les unités de calcul parallèles. Si l'on prend comme exemple une machine à 2 cœurs, alors on a les illustrations suivantes.

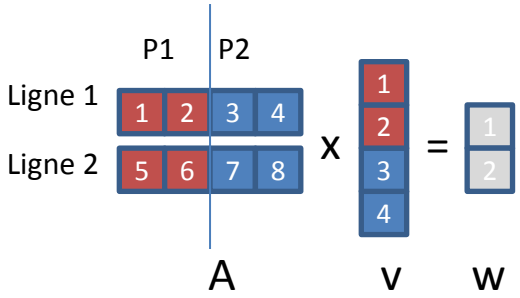


Figure 21. Produit matrice-vecteur par ligne (dots) avec deux processeurs P1 et P2. P1 traite 1 et 2 sur la ligne 1 avec 1 et 2 du vecteur V, alors que P2 fait de même avec 3 et 4. Les résultats de P1 et P2 sommés donnent 1 du vecteur w. Le même processus se reproduit alors pour la ligne 2.

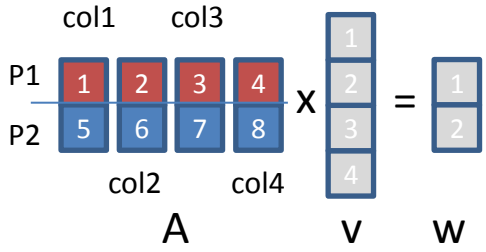


Figure 22. Produit matrice-vecteur par colonnes (axpy) avec deux processeurs P1 et P2. P1 traite son élément de la colonne 1, calculant w(1) = w(1) + col1(1) * v(1). P2 fait de même : w(2) = w(2) + col1(2) * v(1), Les deux processus passent ensuite à la colonne 2 et l'élément 2 du vecteur V pour effectuer les mêmes opérations, et continuent jusqu'au parcours de toutes les colonnes.

Dans le cas de la figure 21, chaque ligne est séparée suivant le nombre de processeurs, ici deux. Ensuite, chaque processeur effectue le produit scalaire de sa portion de ligne avec la portion de vecteur v correspondante. Les résultats de tous les produits scalaires d'une ligne sont ensuite additionnés (opération de réduction), pour obtenir un élément du vecteur w. Pour le produit matrice-vecteur par colonnes, figure 22, chaque colonne est distribuée sur le nombre de processeurs. Cela implique un élément par processeur dans notre exemple. Un axpy est alors effectué avec le morceau de colonne présent sur le processeur, l'élément de v correspondant, et le vecteur w complet. Pour obtenir le résultat final, il faut simplement effectuer tous les axpys.

En effectuant cette analyse des opérations, nous sommes capables de paralléliser chaque opération élémentaire en vue de l'exécution parallèle du produit matrice-vecteur. Les algorithmes correspondants seraient alors :

Tableau 11. Algorithmes du produit matrice-vecteur par ligne ou colonne parallélisé à grain fin.

Pour chaque ligne i de la matrice A_{mxn}	Pour chaque colonne j de la matrice A_{mxn}
Faire en parallèle	Faire en parallèle
tmp = vecteur ligne i de la matrice A	w = 0
portion = n / nb_procs	tmp = vecteur colonne j de la matrice A
dot_loc = 0	portion = m / nb_procs
deb = id_proc * portion	deb = id_proc * portion
end = (id_proc + 1) * portion	end = (id_proc + 1) * portion
pour j de deb à fin faire	alpha = v(j)
dot_loc = dot_loc + tmp(j) * v(j)	pour i de deb à fin faire
fait	w(j) = w(i) + alpha * tmp(i)
sommer les dot_loc sur dot_glo(réduction)	fait
w(i) = dot_glo	Fait
Fait	

Il suffit de transposer ce pseudocode dans le langage cible, avec la parallélisation désirée. Une cible-exemple serait le langage C/C++, avec utilisation d'OpenMP pour le multi-cœurs, ou le langage OpenCL adressant multi-cœurs et accélérateurs.

2.2.1.2. Parallélisation à grain moyen

Le même algorithme peut être parallélisé en utilisant un grain plus gros. Cela implique potentiellement moins de communications, et surtout une efficacité parallèle meilleure de manière générale. Par exemple, la méthode par ligne possède un avantage : les différents produits scalaires sont complètement indépendants. Il est donc possible d'exécuter en parallèle des produits scalaires sur chaque unité de calcul, sans synchronisation. Le même principe s'applique à la méthode par colonne utilisant des axpys, avec une opération de synchronisation/réduction supplémentaire. Les algorithmes précédents deviennent alors :

Tableau 12. Algorithme parallèle du produit matrice-vecteur à grain moyen.

Lignes_par_proc = m / nb_procs
Deb = id_proc * lignes_par_proc
Fin = (id_proc+1) * lignes_par_proc
Faire en parallèle
  Pour i variant de deb à fin
  Faire
  w(i) = dot(ligne i de A, v)
  Fait
Fait
	cols_par_proc = n / nb_procs
	Deb = id_proc *cols_par_proc
	Fin = (id_proc+1) * cols_par_proc
	Faire en parallèle
	  w_loc = 0
	  pour j variant de deb à fin
	  Faire
	  w_loc = w_loc + v(j) * colonne j de A
	  Fait
	Fait
	Sommer tous les w_loc en w_glo (réduction parallèle)
	W = w_glo

Ce type de parallélisation à grain moyen peut être mêlé à de la parallélisation à grain fin sur les opérations dot ou axpy. On est ainsi potentiellement capable d'exploiter de manière plus efficace la hiérarchie mémoire d'une machine parallèle à plusieurs niveaux : plusieurs processeurs adressant chacun leurs blocs de lignes ou colonnes, et plusieurs cœurs dans chaque processeurs adressant leurs propres morceaux de ligne ou colonne.

2.2.2. Bibliothèques d'algèbre linéaire parallèles

Nous avons vu dans les paragraphes précédents différentes manières de paralléliser un noyau numérique connu, dont les principes s'appliquent également à d'autres noyaux dans le calcul scientifique. Ces noyaux étant très utilisés, des librairies spécialisées ont été créées, permettant de traiter la problématique de l'utilisation de ces noyaux en environnement parallèle. Il n'est donc pas nécessaire de développer nos propres noyaux numériques parallèles, permettant une économie de temps substantielle.

2.2.2.1. Noyaux de calculs élémentaires

En algèbre linéaire, les opérations élémentaires impliquent des scalaires, des vecteurs et des matrices. On organise ces opérations en trois catégories, trois niveaux : 1, 2 et 3. Le niveau 1 adresse les opérations scalaire-vecteur et vecteur-vecteur. Quelques illustrations sont « scal », qui permet la mise à l'échelle d'un vecteur, « axpy » et « dot » précédemment vues. Le niveau 2 regroupe les opérations matrice-vecteur. Le produit matrice-vecteur en est une. Le niveau 3 contient toutes les opérations matrice-matrice. De manière générale, les niveaux 1 et 2 impliquent peu d'opérations par élément impliqué, et donc un volume de communication proportionnellement important. Si le volume de données à traiter est en $O(n^2)$ ou $O(n)$, alors le volume de calculs est respectivement en $O(n^2)$ et $O(n)$. Le niveau 3 est beaucoup plus dense en calculs et les communications sont moins importantes en volumes qu'aux niveaux 1 et 2. L'ordre des données traitées y est en $O(n^2)$, et l'ordre des calculs en $O(n^3)$. En termes de performance parallèle, les niveaux 1 et 2 sont limités par la bande passante, et le niveau 3 par la puissance de calcul crête.

2.2.2.2. Bibliothèques d'algèbre linéaires

Une librairie d'algèbre linéaire faisant référence est la librairie « Basic Linear Algebra Subroutines » (33). Elle regroupe la plupart des opérations de bases sur les matrices et vecteurs d'algèbre linéaire et couvre les trois niveaux d'opérations. On parle de BLAS 1, 2 et 3. Plusieurs implémentations des BLAS existent, certaines ouvertes et d'autres propriétaires.

2.2.2.2.1. Bibliothèques ouvertes

L'implémentation ouverte la plus connue provient de l'équipe de Jack Dongarra (34). Elle est séquentielle, c'est-à-dire qu'elle n'utilise qu'un seul processeur/cœur explicitement, et non optimisée pour une architecture cible. Elle n'est donc pas la meilleure en termes de performances, mais très robuste numériquement. Parmi les versions optimisées et ouvertes, on peut citer ATLAS (35) et GotoBLAS (36). ATLAS signifie « Auto-Tuned Linear Algebra Subroutines ». Cette librairie possède des capacités d'autotuning, lui permettant d'optimiser les opérations parallèles sur les matrices par microbenchmarking lors de l'installation. Elle sonde les caches de la machine, et évalue la performance parallèle sur quelques noyaux représentatifs. ATLAS s'adapte donc à toute architecture parallèle de manière dynamique. GotoBLAS est une librairie développée par le « Texas Advanced Computing Center», et propose des implémentations optimisées pour plusieurs architectures de calcul. Les librairies sont optimisées à la main, avec prise en compte des caches et du Translation Look-aside Buffer (TLB) pour de meilleures performances. Globalement, GotoBLAS donne de meilleures performances qu'ATLAS, elle-même plus performante que BLAS.

2.2.2.2.2. Bibliothèques propriétaires

Dans les librairies propriétaires, on peut citer la Math Kernel Library (MKL) d'Intel, et AMD Core Math Library (ACML) d'AMD. Ces deux bibliothèques offrent des performances parallèles optimisées pour les architectures de chaque développeur de processeur. Elles sont souvent plus efficaces que les bibliothèques ouvertes. D'autres librairies propriétaires existent, liées aux différents constructeurs de supercalculateurs : libsci de CRAY, ESSL d'IBM, PDLIB.SX de NEC ou SCSL de SGI pour ne citer que les plus connues.

2.2.2.3. Bibliothèques pour le calcul de valeur propres

Le calcul de valeur propre étant récurrent dans les applications scientifiques, des librairies ont été développées pour ce problème d'algèbre linéaire. Il y a deux classes importantes de librairies : celles qui adressent les problèmes denses, et celles qui adressent les problèmes creux. Ces deux classes sont ensuite parallélisées ou non.

2.2.2.3.1. Bibliothèques pour matrices denses

Basé sur les BLAS, Linear Algebra PACKage (LAPACK) (37) offre un ensemble de fonctions permettant la résolution de systèmes linéaires et de problèmes à valeurs propres. Les implémentations proposées sont numériquement très abouties, ce qui implique une bonne performance séquentielle et une excellente robustesse numérique. Comme LAPACK s'appuie sur les BLAS, la parallélisation de LAPACK peut se faire via ATLAS, GotoBLAS, et toute autre version des BLAS parallèles. Cette parallélisation exploitera les machines à mémoire partagée et les accélérateurs. Pour utiliser des machines à mémoire distribuée, on peut soit effectuer la parallélisation type MPI manuellement, ou utiliser une librairie dédiée telle que Scalable LAPACK (scaLAPACK) (38). LAPACK ne s'adresse qu'aux problèmes

denses, c'est-à-dire où les matrices et vecteurs sont explicitement et intégralement stockés en mémoire, zéros y compris. Cette librairie ne résout donc pas les problèmes où les matrices sont stockées aux formats creux tels que Compressed Sparse Rows(CSR), Compressed Sparse Columns(CSC), …

2.2.2.3.2. Librairies pour matrices creuses

Pour les problèmes creux, deux librairies faisant référence sont « Portable, Extensible Toolkit for Scientific Computation »(PETSc) (39) et « Scalable Library for Eigenvalue Problems »(SLEPc) (40), respectivement dédiées à la résolution parallèle de systèmes linéaires et de problèmes à valeurs propres. Contrairement à LAPACK, ces librairies s'appuient explicitement sur une librairie tierce pour le parallélisme, « Message Passing Interface »(MPI) en l'occurrence. SLEPc permet par exemple d'utiliser les solveurs à valeurs propres Arnoldi/Lanczos, Jacobi-Davidson, et d'autres encore. Une autre librairie intéressante est Arnoldi PACKage (ARPACK), centrée essentiellement sur la méthode d'Arnoldi. Elle permet la résolution de problèmes à valeurs propres sur matrices creuses notamment. La version parallèle de cette librairie, Parallel ARPACK(PARPACK) (41), utilise BLACS (42) pour ses communications parallèles.

Conclusion

Pour conclure ce chapitre dédié à la résolution de problèmes à valeurs propres, nous avons vu dans un premier temps la définition du problème numérique et un aperçu de quelques champs d'application. Certaines méthodes permettant de résoudre ce problème ont été exposées, et nous nous sommes concentrés essentiellement sur la méthode de la Puissance pour les problèmes à valeur propre dominante et la méthode d'Arnoldi pour calculer un sous-ensemble des valeurs propres. La manière de résoudre ce problème en parallèle a été abordée, où l'on peut paralléliser les noyaux numériques élémentaires manuellement, utiliser des noyaux parallélisés, ou encore faire appel à des librairies contenant des méthodes numériques parallélisées. Le chapitre suivant met l'accent sur l'influence de la précision arithmétique du matériel pour les méthodes de Krylov et la résolution de problèmes à valeurs propres.

Chapitre 3. Étude de la précision des accélérateurs pour les méthodes de Krylov

Sommaire du chapitre

Introduction

Lorsqu'un scientifique raisonne sur la résolution abstraite d'un problème numérique, la précision dont il dispose est supposée exacte ou infinie. Par exemple, le nombre 0.01 vaut effectivement 0.01 quand il est manipulé dans des opérations arithmétiques. Dans le calcul scientifique, l'outil informatique intervient pour la résolution de problèmes de grandes tailles. Dans ce cadre, l'arithmétique et la représentation des nombres utilisés pour effectuer les calculs peut ne plus être exacte (dans le sens de précision infinie). C'est le cas lorsque l'on utilise la représentation à virgule flottante des nombres réels, formalisée par la norme IEEE-754. Le papier *« What every scientific should know about floating point arithmetic »* (43) montre quelques-uns des problèmes que l'on peut rencontrer en calcul scientifique sur ordinateur.

Pour résumer ce qui y est présenté, le problème majeur que l'on rencontre provient de la représentation flottante des nombres réels. Les nombres flottants sont représentés en base binaire, suivant le format suivant : signe | mantisse |exposant. Le signe vaut 1 si le nombre est négatif et 0 sinon. La mantisse débute forcément par un 1 implicite et l'exposant est additionné à 127 pour les nombres flottants simple précision. Le nombre flottant est donc représenté par :

Nombre = signe x 2^(exposant-127) x 1.[mantisse]

Par exemple, suivant la norme IEEE en 32 bits, le nombre 0.01 vaut en fait :

1. 0.01(base IEEE) = 0 x (01111000 (base 2)-127)x 1.[10001111010111000010 10] (base 2)
2. 0.01(base IEEE) = + 2^(120-127) x 1,279998779
3. 0.01(base IEEE) = + 2^(-7) x 1,279998779
4. 0.01(base IEEE) = 0.009999990461

On voit que 0.01 n'est pas représenté de manière exacte selon le format IEEE-754 en simple précision, car la mantisse nécessaire pour représenter le nombre est « infinie ».

Un autre phénomène apparaissant lors de certaines opérations flottantes IEEE est la troncature. Si l'on additionne ou soustrait deux nombres dont les exposants sont très éloignés, alors le nombre le plus petit va être ignoré. Ce phénomène s'appelle « cancellation ». Lorsque ce dernier se produit en cascade, on parle alors de « catastrophic

cancellation ». Ce genre d'erreur arithmétique arrive pour toute architecture supportant complètement la norme IEEE, et un exemple d'opération sensible à ces phénomènes est l'opération de réduction du produit scalaire. Une succession d'additions a lieu, avec des éléments qui potentiellement sont d'amplitude différente.

Nous avons rappelé que le support complet de la norme IEEE n'implique pas que les opérations flottantes seront « exactes ». Les matériels accélérateurs tels que le processeur Cell ou les cartes graphiques n'ont pas nécessairement un support complet de la norme IEEE. Cela peut entraîner une erreur supplémentaire par rapport à celle encadrée par la norme IEEE, qui pourrait avoir un impact significatif sur la convergence d'algorithmes itératifs comme ceux présentés précédemment pour la recherche de valeurs propres par exemple.

3.1. Précision arithmétique du processeur IBM Cell

Dans le chapitre 1, nous décrivions brièvement l'architecture du processeur Cell. Ce dernier est constitué d'un cœur de type PowerPC fonctionnant à 3.2 GHz appelé PowerPC Element(PPE). Ce cœur est complètement IEEE, et fournit donc une précision au moins équivalente à celle des architectures processeurs x86 classiques. Les autres cœurs du Cell, les Synergistic Processing Elements(SPE), fournissent l'essentiel de la puissance du Cell, et ne respectent pas complètement la norme IEEE. Plus précisément, les unités de calculs simple précision des SPEs sont dans ce cas, et les unités double précision sont complètement IEEE. La simple précision est prévue pour être utilisée lors de calculs temps réel nécessitant de la performance au détriment de la précision. En effet, pour le jeux-vidéo qui est l'une des cibles privilégiées du Cell au sein de la console de jeux Playstation 3, il est plus important d'avoir un nombre d'images par seconde suffisant avec éventuellement quelques défauts d'affichage occasionnels. Ce n'est malheureusement pas vrai pour le calcul scientifique en général, et certaines expérimentations sur les Fast Fourier Transform sur le Cell montrent une perte de 2 à 3 chiffres après la virgule en simple précision (44). Les informations concernant le Cell et son respect de la norme IEEE pour le calcul flottant sont sporadiques, et les quelques-unes officiellement décrites par IBM concernent les opérations simple précision :

1. Racine carrée
2. Division (calcul d'inverse également)

Au-delà de la perte de précision, l'opération d'arrondi est également limitée. Par défaut, le mode round-to-even est utilisé par la norme IEEE. D'autres modes existent, tels que round-to-zero, round-to-inifinity … Le mode utilisé par le Cell est le mode round-to-zero, qui n'est pas le plus approprié ou précis pour le calcul scientifique. Cela revient à arrondir tout nombre flottant résultat d'une opération simple précision par son arrondi inférieur le plus proche. Dans une note d'IBM concernant le Cell, il est indiqué « The programmer should be aware that, in some cases, the computation results will not be identical to IEEE Standard 754 ones ». Nous testerons cette indication dans la partie 3.3.

3.2. Précision arithmétique des cartes accélératrices Nvidia

Les cartes graphiques en général sont dédiées premièrement au calcul de scènes 3D temps réel, et l'on retrouve la même problématique que pour le Cell : la performance a plus d'importance que la précision du résultat. Les cartes graphiques Nvidia furent les premières à massivement être adoptées par le domaine du calcul hautes performances, et nous concentrerons l'étude de précision sur ces cartes. Plusieurs générations se sont succédées, apportant tout d'abord l'arithmétique flottante simple précision avec le matériel de génération 1.0 puis double précision avec le matériel 1.3. De même, le respect de la norme IEEE s'est amélioré au fil du temps et de l'évolution des matériels, avec un support presque complet de cette dernière avec le matériel 2.0. Les descriptions des générations de matériel et du terme compute capability peuvent être trouvées dans (45). Pour le calcul scientifique, le matériel 1.3 est le minimum envisageable, représenté par les cartes C1060/S1070. La génération suivante représentée par les cartes C20XX fait partie du matériel 2.x, possédant un meilleur support de la norme IEEE. La documentation NVidia est assez fournie au niveau du respect de la norme IEEE, et l'on y retrouve à peu près le même ordre de limitations que pour le Cell en simple précision :

1. L'addition et la multiplication sont souvent combinées en une seule opération flottante « Fused Multiply ADd »(FMAD), qui tronque le résultat intermédiaire de la multiplication ;
2. La division n'est pas implémentée de manière standard ;
3. La racine carrée est implémentée via la racine carrée de l'inverse de manière non standard ;
4. Pour l'addition et la multiplication, seul round-to-nearest-even et round-towards-zero sont supportés de manière statique, et l'arrondi vers +/- l'infini n'est pas supporté

Un extrait provenant du guide de programmation CUDA de Nvidia (45) montre les erreurs courantes sur quelques opérations flottantes élémentaires :

Tableau 13. Erreur Unit in the Last Place(ULP) maximum. Plus ULP est grand, plus grande est l'erreur de calcul. Le terme CC représente la Compute Capability de la carte graphique. Voir (45) pour une description de cette dernière.

Fonction	Erreur ulp maximum	
	Simple précision (SP)	Double précision (DP)
x+y	0	0
x*y	0	0
x/y	- 0 si CC >=2 - 2 sinon	0
1/x	- 0 si CC >= 2 - 1 sinon	0
1 / sqrt(x)	2	1
powf(x,y)[SP] / pow(x,y)[DP]	8	2

Dans le tableau 13, on voit que selon les opérations, on peut presque perdre une décimale de précision en simple précision en une seule opération. En double précision, l'erreur est beaucoup plus réduite, ce que nous pourrons vérifier dans la section 3.3.

3.3. Expérimentations sur la précision pour les méthodes de Krylov

Cette section a pour but d'expérimenter l'influence de la précision des accélérateurs sur des calculs de sous-espaces de Krylov. Quelques tests préliminaires ont été effectués sur 2 générations de cartes graphiques dédiées au calcul scientifique : les Tesla C1060 et C2050. Ces tests préliminaires seront suivis par les expérimentations sur les méthodes de Krylov avec un processeur multi-cœur classique, des cartes graphiques et le processeur Cell.

3.3.1. Tests élémentaires de précision sur carte graphique

Ce test assez simple présenté lors de (46; 47), expérimente les étapes clés d'un calcul de valeur propres par méthode itératives. Le noyau axpy, dot et scal sont utilisés, pour réaliser l'algorithme présenté dans le tableau 14, dont le résultat final devrait être égal à 1.

Cet algorithme a été exécuté sur une architecture Intel Nehalem, et sur cartes graphiques C1060 et C2050. On attend un résultat égal à 1, et le calcul d'erreur est donc effectué de la manière suivante :

$$Erreur = |1 - result|$$

Tableau 14. Test préliminaire de précision pour cartes graphiques.

1. Générer un vecteur v aléatoire
2. w = v + v
3. norm2 = $||w||_2$
4. w = w / norm2
5. result = $||w||_2$ (Théoriquement, result = 1)

Les résultats sont présentés dans la figure 23. Pour la simple précision. On y voit que le processeur x86 Nehalem offre une précision exacte, sauf pour une taille de vecteurs de 1000 et 1 million, où l'erreur absolue est de 10^{-7}. La carte graphique C1060 n'offre quant à elle qu'un seul résultat exact pour une taille de 1000 de vecteur, puis une erreur allant croissant de 10^{-7} à 10^{-5} selon la taille des vecteurs. Le respect de la norme IEEE a donc bien une influence sur des opérations élémentaires d'algèbre linéaire. La carte C2050 offre un respect de la norme IEEE supérieur à celui de la carte C1060, et la figure 23 confirme ce fait. La précision peut être très proche ou presqu'exacte par rapport aux processeurs X86. Les calculs simple précision sur carte graphique peuvent donc être biaisés par le non-respect de la norme IEEE. Les tests précédents étaient effectués en simple précision, qui souffre d'un support moindre de la norme IEEE sur les cartes graphiques utilisées. Le tableau 15 confirme ce meilleur support en double précision. On voit que les deux cartes graphiques offrent une précision supposée exacte pour des tailles de vecteurs allant jusque 10^5. Au-delà de cette taille, une très faible erreur apparait, identique sur les deux générations de cartes pour une taille de vecteur de 10^6. Pour une taille de vecteur de 10^7, la carte C2050 confirme sa supériorité sur la carte C1060 pour le respect de la norme IEEE en double précision. Son erreur est 2,5 fois plus petite que celle de la carte de génération précédente.

Tableau 15. Tests de la précision double arithmétique des cartes graphiques C1060 et C2050. Le tableau indique les erreurs constatées. Dans tous les autres cas, le résultat mesuré était supposé exact.

Vector size	C1060	C2050
1.0e+6	9,99E-15	9,99E-15
1.0e+7	4,00E-14	1,59E-14

Ces quelques tests préliminaires ont permis de vérifier l'hypothèse selon laquelle les cartes graphiques pourraient impacter la précision des calculs scientifique faisant appel à l'arithmétique flottante. Les expérimentations à suivre montreront l'impact de cette différence de respect de la norme IEEE pour des procédés d'orthogonalisation très utilisés dans les méthodes de Krylov.

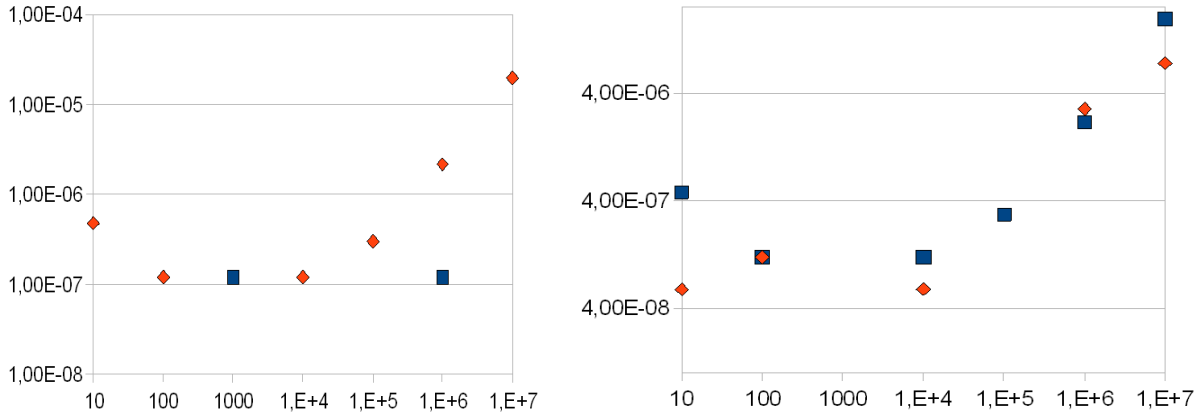


Figure 23. Erreur du calcul de l'algorithme du tableau 14 sur processeur Intel Nehalem (carré bleu) et carte graphique Nvidia C1060 (losanges orange/rouge)en simple précision à gauche et entre deux GPUs différents à droite : Nvidia C1060(carré bleu) et Nvidia C2050(losange orange). L'absence de point indique un résultat supposé exact. Globalement, le Nehalem est plus précis que la carte graphique C1060, et la carte graphique C2050 est plus précise que la carte C1060.

3.3.2. Expérimentation des accélérateurs pour l'orthogonalisation

L'orthogonalisation est un processus clé dans de nombreuses méthodes itératives, telles que GMRES, le Gradient Conjugué, Arnoldi ou Lanczos. Dans cette section, nous nous intéresserons aux orthogonalisations de Gram-Schmidt, dans le but de calculer les matrices V et H à partir de la matrice non-hermitienne A, telles qu'elles satisfassent la relation:

$$A.V = V.H$$

En particulier, trois versions de l'orthogonalisation de Gram-Schmidt seront abordées : Classique (CGS), modifiée(MGS) et avec réorthogonalisation (CGSr). Les algorithmes de ces orthogonalisations sont présentés dans « 2.1.2.3.2 Méthodes de Lanczos et Arnoldi » dans le tableau 7 et le tableau 8. Ces trois versions sont plus ou moins sensibles à la précision arithmétique flottante utilisée.

3.3.2.1. Méthodologie de mesure

Les expérimentations visent à mesurer l'impact de la précision arithmétique flottante sur l'orthogonalisation d'une matrice. Lorsque les matrices V et H sont calculées, V est orthonormale, c'est-à-dire orthogonale avec des vecteurs normés. Ainsi, si l'on utilise le calcul de norme choisi pendant l'orthogonalisation (norme 2 par exemple), alors la norme de chaque vecteur doit être égale à 1. Concernant le critère d'orthogonalité, cela implique que $Q^t = Q^{-1}$ de manière formelle, mais aussi et surtout que chaque vecteur est orthogonal à chaque autre vecteur colonne de la matrice. Autrement dit, nous avons la relation suivante, pour tout vecteur v_i et v_j de la matrice V :

$$(v_i^t , v_j) = 0, \text{ si } i \neq j$$

Nous utiliserons cette relation comme base pour caractériser la précision de l'orthogonalisation sur les différents matériels avec les différentes méthodes. Les produits scalaires entre vecteurs de la base V sont compris entre 0 et 1. Plus ils sont éloignés de 0, plus la base perd son orthogonalité, avec un impact potentiel sur la précision de la méthode nécessitant une orthogonalisation (27). Nous choisirons comme critère le maximum des produits scalaires entre tous les vecteurs de V. Ce critère vaut donc :

$$\text{Max}(v_i^t . v_j),\ i \neq j,\ 1 \leq i \leq p \text{ et } 1 \leq j \leq p, \text{ p taille du sous-espace de Krylov choisie.}$$

Les graphes qui seront présentés dans cette section montreront l'évolution de la perte d'orthogonalité lorsque la taille du sous-espace grandit. En effet, de manière générale plus le sous-espace est grand, plus la perte d'orthogonalité augmente. Cela est dû à la précision arithmétique finie. Les tailles de sous-espace usuellement utilisées dépendent grandement des problèmes à résoudre, et elles peuvent varier de quelques vecteurs à plus d'une centaine.

3.3.2.2. Expérimentations

Les expérimentations ont été effectuées sur plusieurs types de matrices. Des résultats avec des matrices denses générées seront d'abord présentés, puis des résultats impliquant des matrices creuses provenant de collections de matrices telles que MatrixMarket (48) ou de l'Université de Floride (49). Les unités de calculs testées sont des processeurs Intel Nehalem supportant totalement la norme IEEE flottante, les cartes graphiques Nvidia de générations 1.3 et 2.x, supportant la double précision nativement. Ces cartes graphiques ne sont pas complètement IEEE, comme stipulé dans la partie 3.2. Le processeur IBM Cell est également employé pour les expérimentations sur les matrices denses, et respecte partiellement la norme IEEE, voir la partie 3.1 pour plus de détails. Les résultats présentés ci-dessous ont été présentés lors de (50).

3.3.2.2.1. Matrices denses

Pour illustrer les expérimentations faites, nous choisirons une matrice de Hilbert. Cette matrice est très connue dans le domaine de la résolution de systèmes linéaires car elle possède un conditionnement très élevé. Cela implique par exemple des difficultés pour résoudre le système linéaire correspondant en précision arithmétique finie. Cette matrice peut être générée suivant la formule :

$A_{ij} = 1 / (i + j - 1)$

Ainsi, une matrice de Hilbert d'ordre 3 vaudrait :

$$A_{3x3} = \begin{array}{|c|c|c|} \hline 1 & 0.5 & 0.333 \\ \hline 0.5 & 0.333 & 0.25 \\ \hline 0.333 & 0.25 & 0.2 \\ \hline \end{array}$$

(i) Orthogonalisation de Gram-Schmidt Classique et Modifiée (CGS & MGS)

Les résultats sont présentés pour l'orthogonalisation de Gram-Schmidt Classique(CGS) en simple et double précision sur la figure 24. Pour les deux précisions, on remarque une perte d'orthogonalité régulière liée à la précision choisie. En simple précision, on mesure la meilleure perte d'orthogonalité proche de 10^{-7} pour une petite taille de sous-espace, et qui augmente ensuite jusque 1 pour un sous-espace de taille supérieure ou égale à 15. En double précision, cette perte débute à 10^{-16}, pour augmenter régulièrement jusque presque 0,01 ou 1 pour une taille de sous-espace de 32. Cette orthogonalisation confirme sa forte sensibilité à l'arithmétique flottante finie, et si l'on regarde plus en détail chaque figure, on voit des comportements très différents pour le GPU et le Cell.

En simple précision, le Cell est le matériel le moins précis, avec une précision 10 fois inférieure à celle de la carte graphique, ou plus de 100 fois inférieure à celle du processeur x86. Dans les faits, cela implique la perte d'une décimale en termes de précision entre Cell et carte graphique, et potentiellement 2 décimales comparé au processeur. Lorsque l'on passe en double précision, le classement change, et la carte graphique devient le matériel le

moins précis, avec une perte de précision d'un facteur 10 comparé au processeur x86. Le Cell s'approche très fortement du processeur x86, logiquement, car il respecte aussi bien la norme IEEE pour le calcul double précision. Les résultats sont très similaires avec la méthode de Gram-Schmidt modifiée, avec une précision générale meilleure.

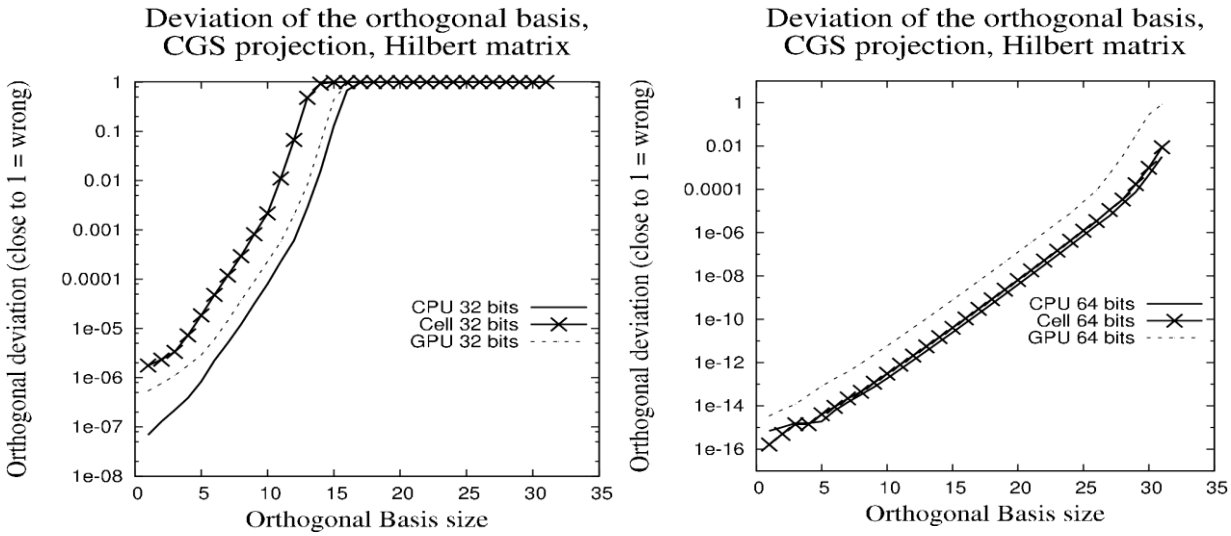


Figure 24. Précision de l'orthogonalisation de Gram-Schmidt Classique en simple précision sur processeur x86, carte graphique C1060 et processeur Cell à gauche et double précision à droite. Plus le résultat est proche de 1, moins bonne est la précision. L'idéal est d'avoir une courbe plate la plus basse possible.

(ii) Orthogonalisation de Gram-Schmidt avec réorthogonalisation

La réorthogonalisation apporte une meilleure tolérance à la précision arithmétique flottante, que l'on peut constater sur la figure 25. Dans les deux cas, la perte d'orthogonalité est fortement diminuée par rapport à CGS, et l'on reste très proche de la précision machine pour les deux précisions flottantes : 10^{-8} à 10^{-7} en simple précision et 10^{-16} à 10^{-15} en double précision.

Le comportement en simple précision reste globalement le même que pour CGS ou MGS en termes de précision : le processeur x86 offre la meilleure précision, suivi par la carte graphique et le processeur Cell. En double précision par contre, le classement change légèrement. Tous les matériels sont extrêmement proches les uns des autres, et la carte graphique semble plus précise que les autres matériels. Ce résultat est à tempérer. En effet, lorsque la base V est construite durant l'orthogonalisation de Gram-Schmidt, la carte graphique, moins précise, n'est pas capable d'effectuer une opération de réduction assez précise en double précision. Plus spécifiquement, lorsque l'on utilise des vecteurs de taille de l'ordre de 10000, s'il faut accumuler des résultats de grande et de petite amplitude, alors il y aura troncature, et perte d'information. Cela arrive également sur processeur x86 classique, mais c'est encore plus présent sur le modèle de carte graphique utilisé, étant donné son non-respect complet de la norme IEEE. Des nombres sont donc ignorés, ce qui implique plus de zéros potentiellement dans la base V. Lors du calcul de la perte d'orthogonalité, on effectue donc le produit scalaire entre des vecteurs contenant plus de zéros avec carte graphique qu'avec processeur x86. Cela implique une erreur qui artificiellement semble plus petite sur la figure 25. Nous verrons ce genre de comportement se reproduire pour les matrices creuses. Pour résumer ces expérimentations sur matrices denses, on constate clairement que le Cell offre une précision bien moindre que les autres matériels en simple précision, à cause de son non-respect plus fort de la précision IEEE. En

double précision, les résultats du Cell sont aussi précis que ceux d'un processeur x86, et donc satisfaisants. Concernant la carte graphique, le résultat, même s'il est biaisé, est proche de ce qu'offre un processeur x86 classique.

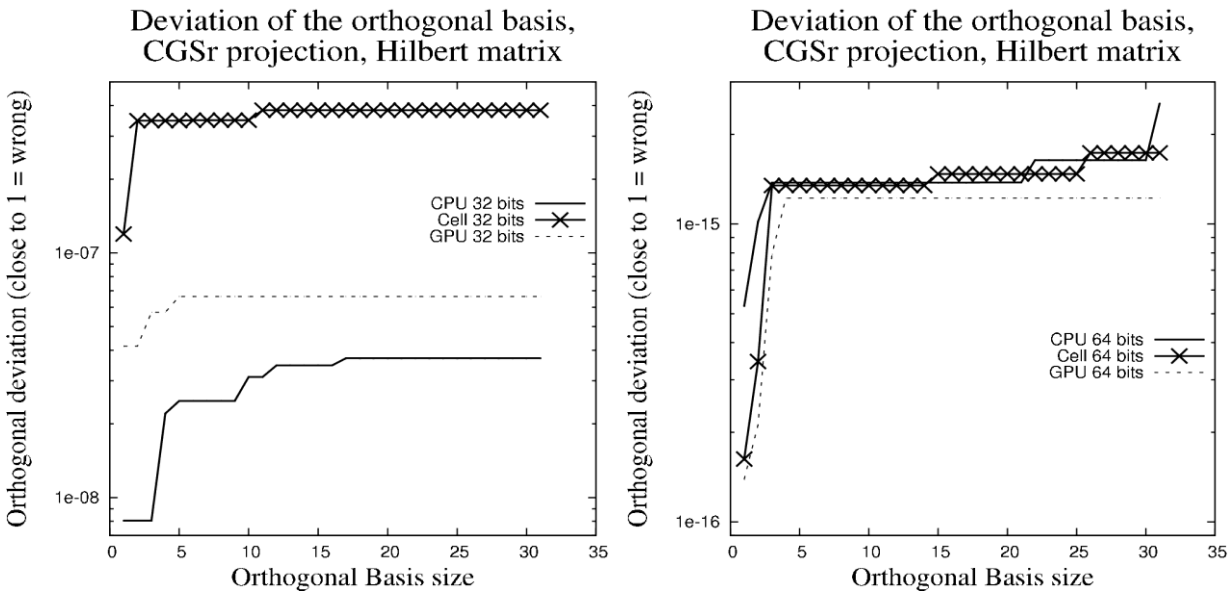


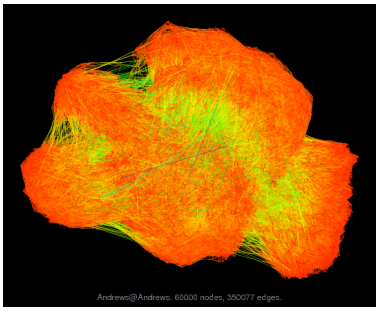
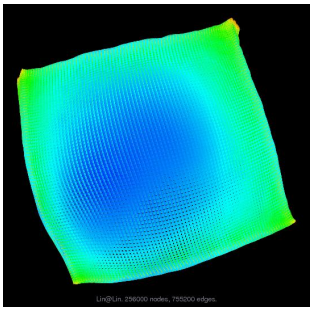
Figure 25. Précision de l'orthogonalisation de Gram-Schmidt avec réorthogonalisation en simple précision sur processeur x86, carte graphique C1060 et processeur Cell à gauche et double précision à droite. Plus le résultat est proche de 1, moins bonne est la précision. L'idéal est d'avoir une courbe plate la plus basse possible.

3.3.2.2.2. Matrices creuses

Pour les matrices creuses, nous appliquerons la même méthodologie que pour les matrices pleines, en ne testant que le processeur x86 et la carte graphique. La précision du Cell par rapport à un processeur x86 est bien définie, mais pas celle des cartes graphiques et quelques expérimentations supplémentaires permettront peut-être d'établir une règle. L'orthogonalisation de Gram-Schmidt classique avec réorthogonalisation caractérise mieux la précision arithmétique finale du matériel, comme montré dans (27) et également précédemment.

Les matrices utilisées proviennent des collections MatrixMarket (48) et de l'Université de Floride (49). Deux matrices ont été sélectionnées pour illustrer cette section: Lin et Andrews. Leurs caractéristiques sont présentées dans le tableau 16.

Tableau 16. Matrices de test pour les orthogonalisations de Gram-Schmidt sur processeur x86 et carte graphique C1060.

Matrice	Andrews	Lin
Ordre	60.000	256.000
Non-zeros	760.154	1.766.400
Problème	Vision	Problème de structure
Graphe		

Les premiers tests concernent la simple précision pour les deux matrices, où la perte d'orthogonalisation est montrée dans la figure 26. Le comportement des matériels est plus marqué que lors des expérimentations sur matrices denses, et l'on constate que la carte graphique donne un résultat apparemment meilleur ou égal à celui du processeur x86 dans les deux cas. Pour la matrice Lin, l'erreur est dans les deux cas excellente car très proche de la précision limite que permets la norme IEEE simple précision : 10^{-8}. Pour la matrice d'Andrews, l'erreur du processeur x86 est proche de 10^{-6}, alors que celle de la carte graphique se situe entre 10^{-8} et 10^{-7}.

Le passage à la double précision permet les mêmes observations qu'en simple précision (voir la figure 27): pour la matrice de Lin, les deux matériels offrent une précision de l'ordre de 10^{-16} soit la précision limite de la norme IEEE. Avec la matrice d'Andrews, la carte graphique semble 10 fois plus précise que le processeur x86, avec une précision de l'ordre de 10^{-16} contre environ 10^{-15}.

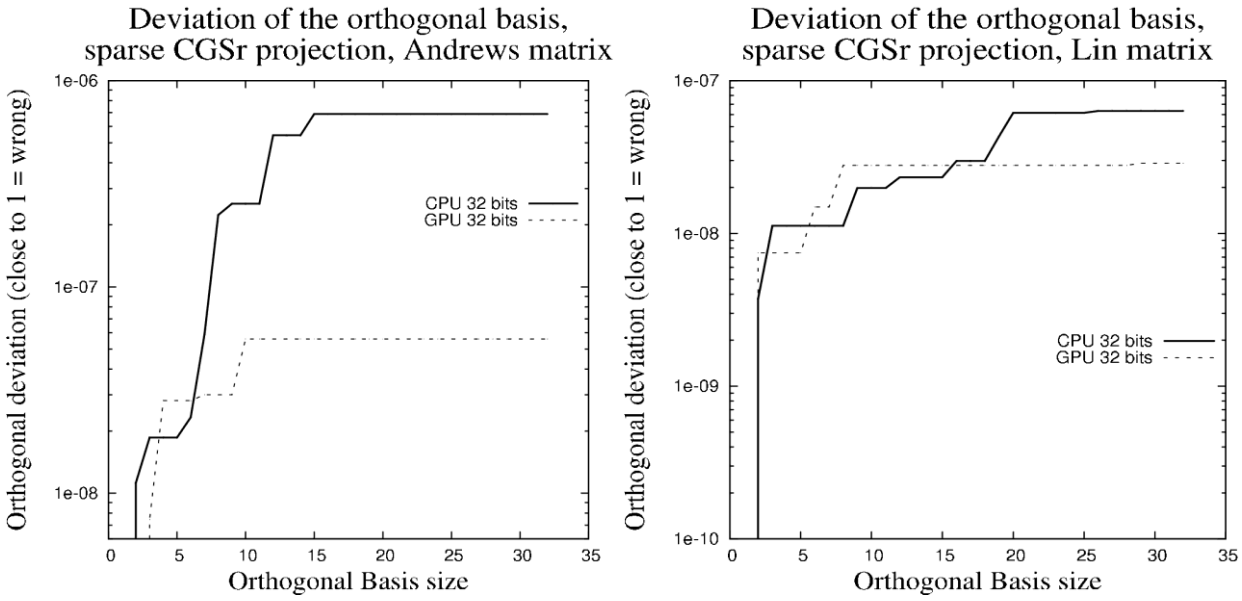


Figure 26. Orthogonalisation de Gram-Schmidt simple précision avec réorthogonalisation sur processeur x86 et carte graphique. Plus la courbe est plate et basse, meilleure est la précision.

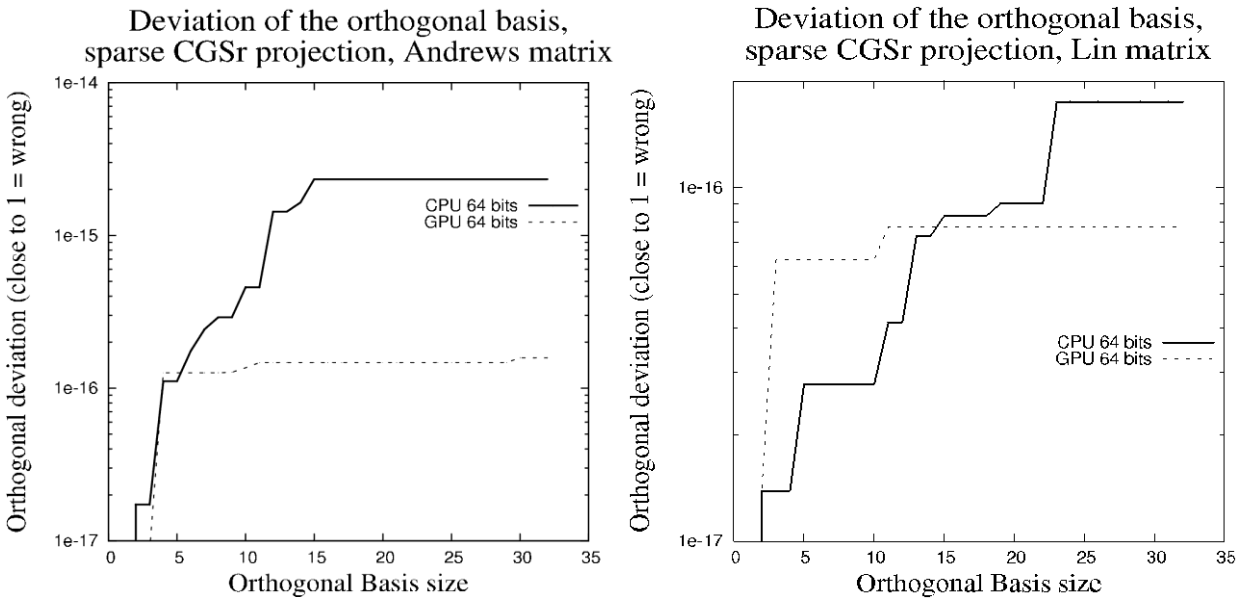


Figure 27. Orthogonalisation de Gram-Schmidt double précision avec réorthogonalisation sur processeur x86 et carte graphique.

Ces résultats diffèrent de ceux présentés dans la section matrice dense pour plusieurs raisons. Premièrement, les matrices impliquées ont une structure différente. Les matrices denses possédaient plus de 10^8 éléments non nuls, alors que les matrices creuses testées en possédaient moins de 2.10^6. Parallèlement, les ordres des matrices creuses sont très supérieurs à ceux des matrices denses : 6.10^4 et $2.5.10^5$ contre 10^4. Ces différences impliquent une répartition des calculs différente. Le tableau 17 résume la répartition des calculs entre produit matrice vecteur et opérations vectorielles. On constate la différence importante qu'il existe : en dense, le produit matrice vecteur est l'opération la plus consommatrice en calcul. Dans le cas creux, la répartition est variable en fonction de la densité de la matrice : plus la matrice est creuse, plus les opérations entre vecteurs dominent le calcul.

Tableau 17. Répartition des calculs entre opérations vectorielles et matrice-vecteur. La taille du sous-espace choisie est 16, et l'orthogonalisation de Gram-Schmidt avec réorthogonalisation est utilisée. La densité est calculée suivant la formule : (non zeros) / (ordre * ordre)

	Dense	Andrews	Lin
non zeros	1,00E+08	7,60E+05	1,77E+06
ordre	1,00E+04	6,00E+04	2,56E+05
Densité	100,00%	0,02%	0,0027%
matrice-vecteur %	99,65%	26,30%	16,27%
op vectorielles %	0,35%	73,70%	83,73%

Sur les cartes graphiques testées, l'une des opérations courantes qui effectue le plus de troncatures est l'opération FMAD, addition et multiplication fusionnée. Cette opération est très présente dans les produits scalaires, l'une des opérations vectorielles de l'orthogonalisation. La précision de ces produits scalaires influence le calcul de norme de vecteurs, utilisé à chaque itération de l'orthogonalisation. Ce biais permanent peut avoir une influence différente selon les données de la matrice. Avec les matrices denses utilisées, les éléments présents dans les vecteurs de la base V étaient d'une amplitude comparable. Dans le cas des matrices creuses, les éléments de la base V pouvaient avoir une forte différence d'amplitude, et certains sont très proches de la précision machine pour les deux précisions :

10^{-7} à 10^{-8} ou 10^{-15} à 10^{-16}. Dans le cas du calcul sur carte graphique, ces éléments pouvaient être tronqués et réduit à zéro. D'où une base V contenant des vecteurs où plus de zéros sont présents que sur processeur. La perte d'orthogonalité est mesurée avec les produits scalaires entre vecteurs de la base V, et donc l'erreur d'orthogonalité mesurée peut sembler plus proche de zéro si plus d'éléments sont tronqués vers le seuil de la précision machine.

Conclusion

Au travers de ce chapitre, nous avons mis en avant la différence de précision pouvant apparaitre à cause d'une implémentation non complètement conforme à la norme IEEE flottante. Les informations divulguées par les constructeurs de matériel nous ont appris avant expérimentation que :

- Le processeur x86 est complètement IEEE pour les précisions simples et doubles
- Le processeur Cell est complètement IEEE en double précision, et non complètement en simple précision avec des pertes de précisions potentielles substantielles
- La carte graphique est proche de la précision IEEE en simple et double précision

Le troisième point concernant la carte graphique est à nuancer : les cartes graphiques utilisées lors du commencement de cette thèse ne supportaient pas la double précision (1.0), puis la génération suivante l'a apporté (1.3), et la génération actuelle offre une précision globale encore meilleure (2.x). Le tableau 18 rappelle brièvement ces différences, et plus de détail sur les Compute Capability peuvent être trouvés dans (45).

Tableau 18. Cartes graphiques et capacité de calcul, notamment double précision.

Carte graphique	Compute Capability	Double précision
Génération GTX400-GTX500	2.x	Oui
Génération GT200	1.3	Oui
Génération 8000-9000	1.0 – 1.2	Non

Également, il faut signaler que la nature intrinsèquement parallèle des cartes graphiques entraîne des calculs à grain fin répartis sur des dizaines ou centaines de cœurs. L'ordre des opérations peut donc changer, entraînant des différences de résultats pour le calcul scientifique, ce qui n'implique pas nécessairement un résultat meilleur dans un sens ou dans l'autre. Ce problème vient de la non associativité des calculs en arithmétique flottante. Le cas typique où peut se produire ce phénomène est l'opération de réduction présente dans les produits scalaires pour le calcul de norme de vecteur.

Les expérimentations effectuées permettent une première caractérisation de l'influence de la précision IEEE des matériels utilisés. Le Cell, pour toutes les orthogonalisations et toutes les matrices offre une précision bien inférieure à un processeur x86 classique en simple précision. Cette perte de précision constante peut être dangereuse pour des applications nécessitant une grande précision flottante. En double précision par contre, le résultat atteint est très proche et acceptable. Concernant la carte graphique, le résultat est plus mitigé. Lors de tests élémentaires, nous constatons que plus la carte graphique respecte la précision IEEE, plus elle offre un résultat précis. Lors d'expérimentations plus complexes au travers de l'opération d'orthogonalisation, la carte graphique offre tour à tour un résultat 10 fois moins précis de manière constante pour des matrices denses, et aussi précis ou 10 fois plus précis qu'un processeur x86 pour des matrices creuses. Cette différence est en partie due à la répartition des calculs entre opération matrice-vecteur et celles purement vectorielles et également aux données numériques des matrices. Lorsque les opérations agissent sur de grands vecteurs qui contiennent des éléments dont la différence d'amplitude atteint la limite de la précision IEEE utilisée, alors la carte graphique tend à remplir les vecteurs avec plus de zéros que la version processeur x86. Cela impacte la mesure d'orthogonalité, et donc donne une précision apparemment meilleure par moment à la carte graphique. Cela étant, la carte graphique fournit tout de même des résultats très proches de ceux d'un processeur x86, et l'influence du non-respect de la précision IEEE des cartes graphiques est relativement acceptable pour le calcul scientifique.

Chapitre 4. Optimisation parallèle de la méthode ERAM

Sommaire du chapitre

Introduction

Durant le chapitre 3, nous nous sommes essentiellement intéressés à la précision de deux accélérateurs d'architecture très différentes : le processeur IBM Cell et les cartes graphiques. Les cartes graphiques offrent un niveau de précision satisfaisant, et une performance théorique supérieure à celle du Cell. Nous nous concentrerons donc sur l'utilisation de cartes graphiques pour des méthodes de calculs de valeurs propres, ainsi que la comparaison avec les processeurs multicoeurs x86. Nous verrons comment uniformiser l'utilisation d'accélérateurs dans le cadre d'architectures hétérogènes multicoeurs ou accélérateurs, et les performances atteignables, atteintes, ainsi que les limitations de cette catégorie de méthodes.

4.1. Parallélisation multicoeur et accélérateur

L'utilisation des accélérateurs actuels peut être problématique pour deux raisons essentielles :

1. La différence de langage
2. La différence d'adressage de l'espace mémoire

Le premier point est évident : pour utiliser un accélérateur dont le langage machine est différent, un langage de haut niveau adapté est nécessaire. Cela peut-être CUDA pour les accélérateurs Nvidia, ou bien OpenCL de manière plus générale pour tous les types d'accélérateurs, et même les multicoeurs. Dans le cadre de projets de grande envergure, industriels ou académiques, ce point implique des contraintes pour les prérequis logiciels (librairies), ou surtout la chaîne de compilation.

Le second point est actuellement en cours de résolution, mais encore très présent sur la plupart des accélérateurs répandus. La mémoire de l'accélérateur n'est accessible que par ce dernier, et non directement par le processeur hôte, et vice-versa. Cela implique des transferts mémoire explicites à la charge du développeur, qui doit donc maintenir la cohérence des données manuellement. De nouveaux kits de développements permettent de cacher cette contrainte, avec un impact négatif sur les performances si cette différence d'adressage mémoire n'est pas traitée avec attention.

Des solutions sont apparues pour traiter le second point dans le cadre du calcul scientifique lors de la rédaction de cette thèse. La première est un adressage unifié de la mémoire du processeur x86 et de la mémoire de l'accélérateur. Un exemple est le kit de développement CUDA 4.0 pour carte graphique Nvidia. La seconde est une librairie de vecteurs cachant les opérations mémoires au développeur. La librairie Thrust, intégrée à CUDA 4.0, en est une illustration. Elle fournit une implémentation de certaines structures de données et algorithmes présents dans la librairie standard STL du langage C++. Des conteneurs templatés permettent de stocker des éléments de tous types, notamment simple et double précision, ou entiers. Ce stockage peut avoir lieu dans la mémoire vive du processeur ou la mémoire de la carte graphique. Le code présenté dans le tableau 19 présente un exemple d'utilisation de la librairie Thrust. La librairie CUSP (51) pour Cuda SParse, utilise Thrust et propose une implémentation haut-niveau de format de matrices creuses ainsi que d'algorithmes itératifs connus tels que le gradient conjugué, la méthode GMRES, biCGstab.

Pour cette partie parallélisation multicoeur et accélérateur, nous détaillerons tout d'abord le choix d'un framework parallèle pour la gestion des accélérateurs lors de calculs de méthodes de Krylov, puis les performances obtenues lors d'expérimentations de ces implémentations. Ces dernières ont été présentées dans (52; 53).

Tableau 19. Exemple de code utilisant la librairie Thrust.

```
int main(void)
{
    // initialize all ten integers of a device_vector to 1
    thrust::device_vector<int> D(10, 1);

    // set the first seven elements of a vector to 9
    thrust::fill(D.begin(), D.begin() + 7, 9);

    // initialize a host_vector with the first five elements of D
    thrust::host_vector<int> H(D.begin(), D.begin() + 5);

    // set the elements of H to 0, 1, 2, 3, ...
    thrust::sequence(H.begin(), H.end());

    // copy all of H back to the beginning of D
    thrust::copy(H.begin(), H.end(), D.begin());

    // print D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    return 0;
}
```

4.1.1. Création d'un framework parallèle pour accélérateurs

Cette section est en partie liée au chapitre concernant l'application aux solveurs neutroniques. En effet, au-delà de l'applicabilité de cette thèse pour des algorithmes connus de calculs de valeurs propres, les contraintes liées aux solveurs du projet APOLLO3® ont été prises en compte lors du développement des solveurs de ce chapitre. En particulier, le style de programmation est orienté objet, et le langage utilisé est C++. Cela implique quelques contraintes sur les solutions possibles pour utiliser les cartes graphiques. Également, le matériel adressé concerne les multicoeurs x86 ainsi que les cartes graphiques Nvidia. Le langage choisi pour les cartes graphiques s'est donc porté sur CUDA, OpenCL n'étant pas disponible lors du commencement de ces travaux de thèse. Les idées et applications potentielles de ce framework ont été présentées dans (54).

Parmi les caractéristiques communes entre les solveurs du code APOLLO3® et les solveurs d'algèbre linéaire, il y a les structures de données et les opérations de calcul. On peut trouver l'utilisation de scalaires, vecteurs et matrices. Les scalaires font partie des types primitifs du langage cible, et les vecteurs et matrices sont des structures ou objets plus évolués. Également, les opérations utilisées sont proches : opérations vectorielles (BLAS 1) et matrice-vecteur (BLAS 2). Pour une utilisation d'accélérateurs facilitée dans le cadre des solveurs de problèmes à valeurs propres, il faut à la fois des abstractions de structures de données et des noyaux de calculs adaptés aux accélérateurs et multicoeurs à la fois.

4.1.1.1. Structure de données

Plusieurs implémentations sont possibles, et plusieurs solveurs du code APOLLO3® développé au CEA utilisent la librairie STL (Standard Template Library) et notamment les vecteurs. Ces derniers offrent une implémentation haut-niveau efficace d'un tableau dynamique, avec quelques fonctions supplémentaires, telles que les initialisations et changements de taille. Concernant la matrice, elle peut être vue comme une structure ou un

objet plus évolué bâti sur les scalaires et vecteurs. Ainsi, pour utiliser de manière facilitée des matrices ou des vecteurs sur carte graphique, une structure de haut niveau peut permettre de résoudre les problèmes liés aux deux points présentés précédemment : langage différent et adressage mémoire séparé. Plusieurs idées ont été proposées lors de réflexions menées au sein du laboratoire d'accueil de cette thèse, et en sont ressorties deux solutions viables: une carte de correspondance mémoire ou une classe de vecteur spécialisé.

Pour les deux solutions, et au vu de la structure du code APOLLO3®, il a été choisi d'imposer le choix de l'utilisation de la carte graphique dans le projet de manière statique. Cela implique que dès la compilation, il est indiqué que la carte graphique est active ou non. Il n'est donc pas possible de débrancher l'utilisation de la carte graphique lors de l'exécution du code.

4.1.1.1.1. Solution 1 : carte de correspondance mémoire

Cette solution représente une manière d'unifier l'adressage mémoire, grâce à un système de correspondances mémoires. Ainsi, dans une phase d'initialisation, le développeur indique quelles sont les allocations mémoires nécessaires avant l'exécution du code. Cela concerne les matrices et vecteurs dans le cadre des solveurs étudiés. Les matrices étant basées sur des vecteurs, cela revient à déclarer tous les vecteurs nécessaires aux calculs.

A chaque vecteur serait associé en interne au système un identifiant. Cette liste de déclarations et identifiants peut ensuite être utilisée pour soit allouer la mémoire sur le processeur classique, soit allouer la mémoire sur le processeur graphique. Il serait également possible de l'allouer sur les deux, pour permettre des calculs partagés et maintenus à jour automatiquement par le système. Ce système, le Transcoder, maintiendrait la cohérence des données lors d'opérations de calcul sur les vecteurs. Ces opérations vectorielles encapsuleraient donc un appel au transcoder avant chaque opération. Le tableau 20 présente un exemple d'utilisation.

Ce dernier montre les avantages et inconvénients de cette solution. Les avantages résident dans l'absence de refactoring des fonctions de calcul. Il suffit d'ajouter les conversions d'adresse, et les calculs s'effectuent selon la compilation statique du code soit sur GPU soit sur CPU, suivant le noyau de calcul approprié. Les inconvénients sont cependant nombreux avec le transcoder : la recherche et le maintien des correspondances d'adresse peut être assez coûteux, la structure est peu dynamique, et finalement la maintenabilité, réutilisabilité et le calcul hybride sont très limités. Pour ces raisons, ce choix logiciel a été écarté. Cependant, il faut noter que cette solution est envisagée par Microsoft dans le cadre de son langage parallèle C++ Accelerated Massive Parallelism (C++ AMP). Une partie de ce langage propose un mappage implicite de la mémoire centrale du processeur et de celle de la carte graphique, avec des transferts automatiques lorsque les données sont nécessaires sur un matériel ou l'autre.

Tableau 20. Illustration de l'utilisation du transcoder. (1) La fonction convertir effectuera l'allocation mémoire et le transfert de données entre CPU et GPU si nécessaire. Les calculs auront alors lieu sur le même matériel pour l'opération d'addition : soit complètement GPU, soit complètement CPU.

Code	Exécution
Fonction initialise du Transcoder : Créer vecteur v1 de taille n sur CPU Créer vecteur v2 de taille n Fonction additionner_vecteur du code : V1_ptr = transcoder.convertir(v1) V2_ptr = transcoder.convertir(v2) Addition_effective_vecteur(V1_ptr , V2_ptr)	 Allocation de mémoire sur CPU Allocation de mémoire sur CPU ou GPU (statique) V1_ptr contient la référence CPU de v1(1) V2_ptr contient la référence CPU ou GPU de v2 L'addition est effectuée entre les vecteurs
1. Création du Transcoder Transcoder transcoder 2. Initialisation des données Transcoder.initialise() 3. Opérations de calcul (…) Additionner_vecteurs(v1 , v2) (…) 4. Destruction des données Transcoder.finalise()	 La fonction initialise contient toutes les allocations mémoires et transferts initiaux.

4.1.1.1.2. Solution 2 : classe de vecteur spécialisé

La deuxième solution, le vecteur spécialisé que l'on nommera AcceleratedVector, propose de remplacer les vecteurs utilisés comme conteneur élémentaire du code pour les calculs par des conteneurs plus généraux. Un des intérêts est de cacher la différence d'adressage mémoire par rapport à l'utilisation de pointeurs explicites sur la mémoire du GPU. Cette option peut permettre aussi d'utiliser la vectorisation des calculs sur processeur x86 via les unités SSE (55), ou encore une parallélisation multicoeur automatique. Il serait même possible d'imaginer un vecteur dont la répartition se ferait sur plusieurs matériels : accélérateurs ou multicoeurs, pour une utilisation de toute la capacité de calcul d'une machine donnée.

Plusieurs solutions ont été étudiées pour ce conteneur, proche dans l'utilisation mais aux impacts directs et indirects différents :

1. Une classe héritant du vecteur de la STL
2. Une classe héritant statiquement du vecteur de la STL ou d'un vecteur pour Accélérateurs
3. Branchement statique de la classe :
    a. Soit vecteur de la STL
    b. Soit vecteur pour GPU

(i) Héritage direct du vecteur de la STL

L'héritage direct du vecteur de la STL implique une spécialisation de cette classe pour nos besoins. Ainsi, si l'accélérateur n'est pas utilisé, alors le programme ne contiendra que des vecteurs héritant directement de ceux de la STL, sans redéfinition ou spécialisation de méthodes : la classe AcceleratedVector sera vide de fonctions. Si le GPU est nécessaire, alors les méthodes liées au calcul ou chargements mémoires seront redéfinies pour l'accélérateur.

L'avantage de cette solution est l'absence de changement de signature pour les fonctions utilisant des std ::vector. L'AcceleratedVector peut se substituer en toute transparence. Mais l'héritage direct a plusieurs inconvénients : si certaines méthodes du STL vector ne sont pas modifiées pour l'accélérateur, alors des valeurs erronées peuvent être retournées. On peut prendre comme exemple « size() », qui retourne la taille du vecteur. Si

size fait référence au tableau interne de STL vector lorsque l'accélérateur est activé, alors qu'un tableau interne à AcceleratedVector est utilisé, alors la taille retournée peut être erronée. A l'exécution, il est très difficile de différencier les deux classes, et des confusions peuvent apparaître, notamment dans les noyaux de calculs appelant un std ::vector. Aussi, la possibilité de redéfinir les fonctions de std ::vector existe, altérant possiblement leur comportement et leurs performances.

(ii) Héritage sélectif du vecteur de la STL

Cette solution offre une différenciation supérieure à celle de la solution précédente : une classe par matériel existe, avec sa propre implémentation des méthodes liées aux vecteurs. La signature des fonctions qui employaient des std ::vector devra changer, pour faire appel à AcceleratedVector. Cela entraine un impact assez important sur des codes existants tels que les solveurs du code APOLLO3®. Également, la possibilité de réécrire les fonctions des std ::vector existe, avec les impacts mentionnés précédemment.

(iii) Branchement statique

La dernière solution envisagée concerne le choix de la classe de vecteur par branchement statique entre std ::vector et AcceleratedVector. Une chaîne de caractère codée sous forme de macro-définition permettrait d'indiquer l'utilisation d'un vecteur de la STL ou pour accélérateur. Les deux classes seraient complètement séparées, et il faudrait simplement que la classe de vecteur accéléré respecte l'interface de std ::vector, et propose donc les mêmes fonctions. Cette solution implique un changement des déclarations de vecteurs dans les fonctions où l'on souhaite utiliser l'accélérateur, avec les mêmes impacts que la solution précédente. En termes d'avantages, il n'y aurait aucun impact sur les performances sur processeur x86, car le vector de la STL serait utilisé tel quel. Le débogage est plus aisé qu'avec les solutions précédentes également, étant donné que std :: vector et AcceleratedVector sont complètement séparés et identifiés. De plus, il est possible d'utiliser en même temps dans le programme un std ::vector et un AcceleratedVector.

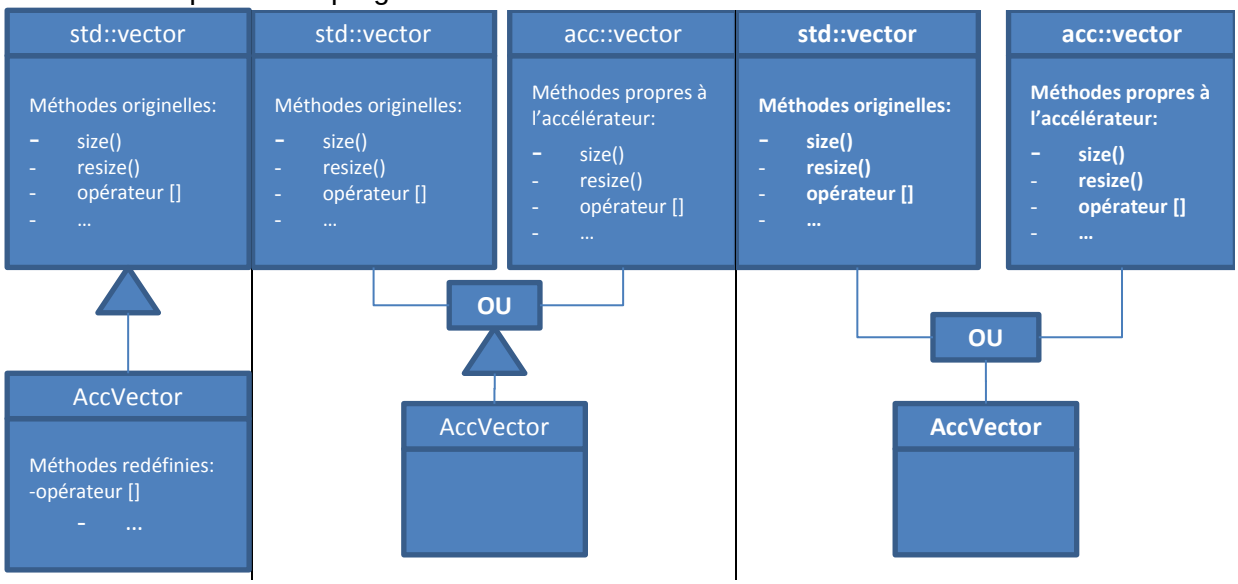


Figure 28. Illustration des trois solutions proposant un conteneur de type vecteur évolué pour multicoeur ou accélérateur.

4.1.1.1.3. Synthèse et choix

Parmi les trois solutions présentées la troisième offre globalement plus de possibilités et de souplesse dans son utilisation. Un exemple d'implémentation en C++ est donné dans le Tableau 21. On voit que les solutions par héritage n'empêchent pas un développeur de réimplémenter différemment les fonctions de std ::vector, pouvant amener des dysfonctionnements ou une performance sub-optimale. Par rapport à std ::vector. La solution avec branchement statique permet de conserver le comportement de std ::vector, tout en offrant la possibilité d'utiliser un accélérateur. Il suffit juste que la classe de vecteur pour l'accélérateur se conforme à l'interface de std ::vector nécessaire pour le solveur.

Tableau 21. Exemple d'implémentation des trois possibilités utilisant les AcceleratedVectors.

Héritage	Héritage sélectif	Branchement statique

Template <class T>
Class MyVector : public std ::vector<T>{
void resize(int new_size)
{
#ifdef GPU
CODE GPU
#else
std ::vector<T> ::resize(new_size) ;
//possible altération du resize CPU
#endif
}
	#ifdef GPU
	#include « GPU_Vector.h »
	#define VEC GPU_Vector
	#else
	#include <vector>
	#define VEC std ::vector
	#endif
	
	Template <class T>
	Class GPU_Vector{
	T * ptr_mémoire ;
	int nb_éléments ;
	void resize(int new_size){
	CODE GPU
	}
	
	Template <class T>
	Class AcceleratedVector : public VEC <T>{
	//si Vec est std ::vector, la réécriture de fonctions de std ::vector est possible ici, et peut-être non souhaitée
	} ;
	} ;
		#ifdef GPU
		#include « GPU_Vector.h »
		#define AccVec GPU_Vector
		#else
		#include <vector>
		#define AccVect std ::vector
		#endif
		
		Template <class T>
		Class GPU_Vector{
		//idem que version 2
		}

4.1.1.2. Opérations flottantes

Une fois la structure de données établie, la manière de choisir les noyaux de calculs sur accélérateur ou multicoeur peut être discutée. Principalement, deux choix s'offrent à nous, non incompatibles : la voie de la programmation orientée objet ou celle plus classique de l'utilisation de fonctions séparées des conteneurs de données. Les possibilités envisagées sont donc la localisation des opérations de calculs au sein de l'objet vecteur accéléré, ou dans une bibliothèque séparée.

4.1.1.2.1. Opérations portées par l'objet

Le conteneur vecteur possédant toutes les informations nécessaires pour effectuer les calculs, un des idiomes de la programmation orientée objet voudrait que le vecteur en question encapsule les opérations flottantes l'impliquant fortement. Les avantages de ce type de programmation sont nombreux, et permettent de tirer partie des propriétés des objets, telles que l'héritage/la spécialisation ou encore des données encapsulées au même endroit que les fonctions, et donc facilement accessibles. Ce genre de technique s'applique également à l'objet matrice, avec les mêmes propriétés. Cela permettrait globalement d'écrire l'algorithme d'orthogonalisation de Gram-Schmidt Modifié de la manière suivante :

Tableau 22. Gram-Schmidt Modifié avec utilisation d'encapsulation des fonctions dans les objets matrices et vecteur (à gauche) et avec des bibliothèques séparées (à droite).

Gram-Schmidt Modifié (MGS), encapsulation	**Gram-Schmidt Modifié (MGS), bibliothèque séparée**
V0.scal(1 / v0.norm2()) Pour i de 1 à m Faire 1. A.matrix_vector(v_{i-1}, v_i) 2. pour j de 1 à i 3. faire a. H(j , i) = v_j.dot(v_i) b. Vi.axpy(-H(j , i), v_j) #axpy(alpha, x) 4. fait 5. H(i , i+1) = vi.norm2() 6. si H(i , i+1) = 0 alors stop 7. fin si 8. vi.scal(1 / H(i , i+1)) Fait	Blas ::scal(v0, 1 / v0.norm2()) Pour i de 1 à m Faire 1. Blas ::matrix_vector(A, v_{i-1}, v_i) 2. pour j de 1 à i 3. faire a. H(j , i) = Blas ::dot(v_j, v_i) b. Blas ::axpy(-H(j , i), v_j, v_j) #axpy(alpha, x,y) 4. fait 5. H(i , i+1) = Blas ::norm2(vi) 6. si H(i , i+1) = 0 alors stop 7. fin si 8. Blas ::scal(vi, 1 / H(i , i+1)) Fait

Avec cette solution, l'algorithme reste très lisible, et permet par exemple d'abstraire complètement quel est le type de matrice utilisé, et de même pour le type de vecteur. Il est ainsi possible de définir une interface ou super classe « MatriceAbstraite » qui sera utilisée pour les calculs, avec une instanciation effective d'une « MatriceDense » ou « MatriceCreuse ». MatriceAbstraite déclarerait l'existence d'une opération « matrix_vector(vector_multiplication, vecteur_résultat), effectivement et obligatoirement implémentée dans les sous-classes « MatriceDense » et « MatriceCreuse ». On peut remarquer que ce choix d'implémentation n'empêche pas l'utilisation effective de bibliothèque séparée pour les opérations de calculs. Cependant, une contrainte existe dans notre cas. Nous visons l'intégration des travaux effectués dans ce chapitre dans les solveurs neutroniques du code APOLLO3®. Ces solveurs utilisent des vecteurs standards de la STL C++, ce qui implique que nous ne pouvons pas ajouter nos propres opérations de calcul à ces derniers. La seule solution qui permet de gagner en souplesse serait de créer une interface ou super classe VecteurAPOLLO3, qui encapsulerait un vecteur de la STL et les différentes opérations nous intéressant. Le problème avec ce type de solution est un possible surcoût lié aux indirections de la classe VecteurAPOLLO3® vers son attribut std ::vector.

4.1.1.2.2. Opérations localisées dans une bibliothèque séparée

La solution classique pour stocker les opérations de calculs d'algèbre linéaire est la bibliothèque séparée, reposant par exemple sur l'interface BLAS. Cette interface est standardisée, et permet de changer de bibliothèque simplement lors de la compilation, pour bénéficier des avantages d'une implémentation particulière. Quelques exemples de ces librairies sont présentés dans la section 2.2.2.2. L'algorithme MGS écrit en utilisant des librairies séparées est présenté dans le tableau 22. Dans cet exemple, le type de produit matrice vecteur est porté par la fonction BLAS matrix_vector. Il est possible d'invariablement appeler cette fonction avec les arguments (matrice, vecteur_multiplication, vecteur_résultat) pour une matrice dense ou creuse si l'on a accès à la surcharge des fonctions dans le langage cible. C'est la cas ici avec le C++, et donc on peut déclarer une fonction Blas ::matrix_vector(MatriceDense, Vecteur, Vecteur) et une autre de même nom : Blas ::matrix_vector(MatriceCreuse, Vecteur, Vecteur). Le comportement de type spécialisation objet de l'encapsulation est mimé implicitement, en définissant une interface commune Blas ::matrix_vector.

4.1.1.2.3. Choix de l'implémentation des opérations flottantes

Pour les raisons évoquées dans cette section, il n'est pas possible d'ajouter des opérations de calculs à la classe vector de la STL C++, sans surcoût potentiel lors des calculs. Ce surcoût n'est pas acceptable dans l'hypothèse de l'utilisation de cette implémentation pour le code APOLLO3® du CEA. La solution de l'encapsulation des calculs pour les vecteurs, malgré ses qualités, n'est donc pas envisageable. Le choix se portera donc sur l'utilisation de bibliothèque séparée, avec une implémentation judicieuse développée dans les lignes suivantes.

Les BLAS standards reposent sur l'utilisation de pointeurs, alors que nous utilisons des structures objets de type vecteurs ou matrices, bien plus riches en informations. Nous pouvons donc créer une couche statique qui inclura les appels vers les BLAS choisies. Elle pourra utiliser la surcharge pour cacher les appels simple ou double précision, explicitement différents dans les BLAS classiques. Cette couche statique définira ainsi l'interface BLAS nécessaire à notre programme, et son implémentation variera en fonction des paramètres de compilation : utilisation de BLAS écrites manuellement pour notre application, de BLAS standards ouvertes ou encore propriétaires. La figure suivante représente cette implémentation.

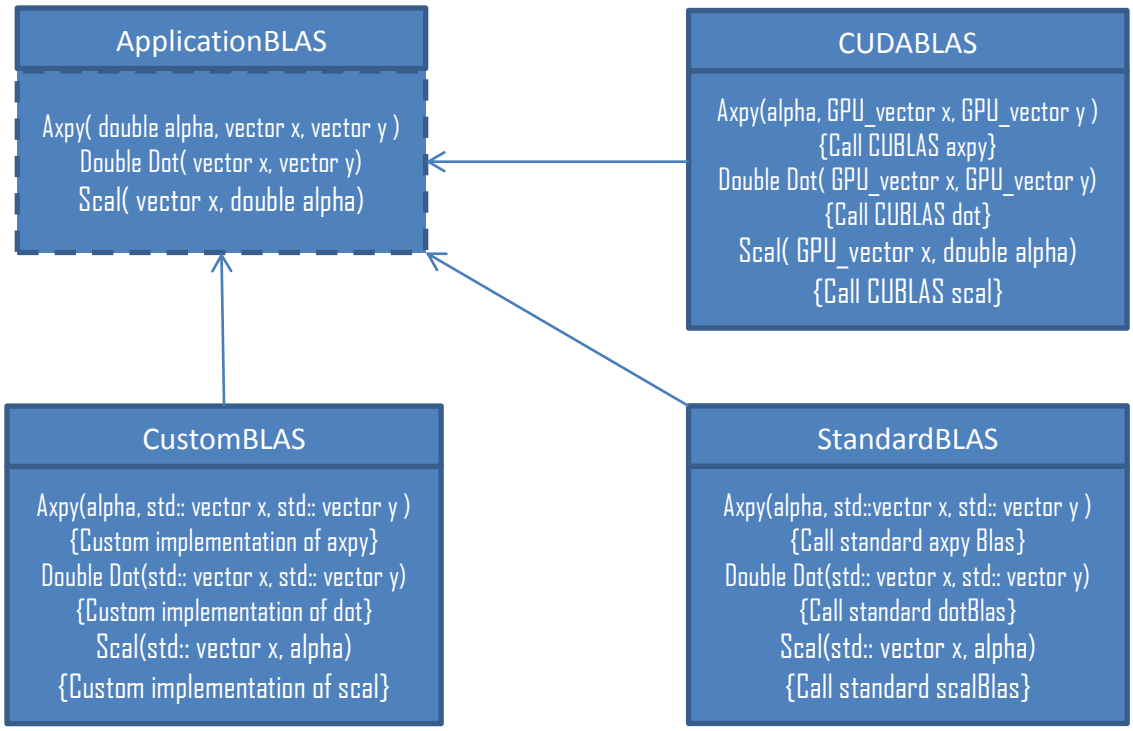


Figure 29. Vue UML de l'implémentation des appels BLAS par bibliothèque. ApplicationBLAS n'est pas un objet existant dans le code, mais plutôt une vue des appels de l'application. Chaque BLAS représente une implémentation de l'interface demandée par le programme. Si l'application est compilée en mode GPU, alors les vector d'ApplicationBLAS seront des GPU_Vector, pris en compte par l'implémentation CUDABLAS des Blas. A noter qu'il est possible d'appeler manuellement les BLAS appropriées, et donc d'utiliser conjointement CustomBLAS, StandardBLAS et CUDABLAS.

L'utilisation des BLAS est en fait très liée au choix d'implémentation des vecteurs. Ainsi, si l'application a besoin de BLAS spécifiques au type de vecteur utilisé, tel que le GPU_Vector, alors cette dernière devra être compilée avec les CUDABLAS. Si le type de vecteur est plus « standard », alors l'application pourra être compilée avec des BLAS plus classiques. Ce choix d'architecture logicielle permet avec la même application de ne compiler et utiliser que les bibliothèques de code nécessaires. Si une machine ne dispose pas de CUDA ou de GPUs Nvidia, alors il est possible d'utiliser des conteneurs vecteurs classiques et des opérations liées appropriées à la machine, sans jamais compiler une ligne de code CUDA. Le projet global est alors beaucoup plus portable, et facile à maintenir. Ce sont des qualités cruciales pour des codes industriels tels que le code APOLLO3®.

4.1.2. Parallélisation multicoeur et accélérateur

Dans un premier temps, nous nous intéressons à l'utilisation de plusieurs cœurs de processeurs au sein d'une machine à mémoire partagée. Les paradigmes de programmation les plus répandus à cette fin sont OpenMP et MPI. Le premier s'applique uniquement à la mémoire partagée, et le second couvre également les machines à mémoire distribuée. Comme nous avons pu le voir dans la section 2.1.2.3.2, l'essentiel des calculs se situe au sein de la projection d'Arnoldi, appelée également réduction d'Arnoldi. Cette dernière occupe plus de 95% du temps de calcul, et agit sur de grandes tailles de matrices et de vecteurs. La partie calculant effectivement les valeurs propres agit sur des problèmes de taille très réduite, et consomme donc peu de ressources de calcul. En particulier, paralléliser cette partie ne permettrait pas de gain, à cause du coût élevé des communications. Toute la parallélisation aura donc lieu au sein de la projection d'Arnoldi pour ce qui concerne la méthode d'Arnoldi redémarrée.

4.1.2.1. Mémoire partagée

Pour utiliser des multicoeurs en mémoire partagée, nous pouvons utiliser deux approches : par librairie BLAS multithreadée ou par threading des noyaux de calcul. La première option est possible par l'utilisation des librairies BLAS telles que la librairie ATLAS de Netlib, ou encore des librairies propriétaires. Concernant le threading, nous faisons appel à la programmation OpenMP, quand nous utilisons les CustomBLAS. Des pragmas ajoutés aux boucles de plus haut niveau permettent une répartition des calculs automatique dans un environnement multicoeur. Dans les deux solutions utilisées, les données restent inchangées par rapport à la version séquentielle, contrairement à une parallélisation pour machine à mémoire distribuée.

4.1.2.2. Mémoire distribuée

La mémoire distribuée implique une répartition explicite des données de calcul entre les différents processus utilisés. Pour échanger des données entre ces derniers, l'interface MPI sera utilisée.

La matrice d'entrée est découpée suivant le format échiquier et le produit matrice vecteur est complètement réparti entre les processus suivant la méthode décrite dans (56), sans l'aspect pipelining multithreadé. Le schéma de communication est le même : un produit matrice-vecteur local pour chaque processus MPI, puis une réduction par ligne i dans le i-ème processus de la ligne, et un broadcast vers tous les éléments de la colonne de ce processus. Elle offre une excellente scalabilité en limitant les échanges réseaux à leur strict minimum. De plus, le résultat stocké dans chaque processus MPI contient toute l'information nécessaire pour effectuer le produit matrice vecteur suivant pour chaque processus. Les opérations vectorielles sont uniformément distribuées, chaque processus travaillant sur sa partie du vecteur. Ainsi, pour une matrice d'ordre N dense et P processus, chaque processus travaillera sur N^2/P données de la matrice et N/P données de chaque opération vectorielle. L'avantage de cette répartition est une parallélisation très uniforme des calculs, mais l'inconvénient se situe au niveau des produits scalaires. Ils sont nécessaires pour effectuer les opérations subséquentes, et des réductions globales sont à effectuer pour chaque produit scalaire. Il est possible de les grouper pour optimiser la performance, ce qui implique au minimum une réduction globale par itération de la projection. Dans le cas de CGSr, où il y a réorthogonalisation, cela implique deux réductions globales. Une de plus est à ajouter pour le calcul de la norme 2 nécessaire à la normalisation, d'où 2 à 3 communications collectives au minimum selon l'orthogonalisation, sans compter celles du produit matrice vecteur.

4.1.2.3. Utilisation d'accélérateurs

Pour utiliser les accélérateurs, nous faisons appel au framework développé dans la section 4.1.1. Pour la partie accélérateurs, le conteneur utilisé est le GPU_Vector, et les opérations BLAS font appel à la librairie CUBLAS de Nvidia. Pour les opérations creuses, les algorithmes de produits matrice vecteur avec différents formats de matrices ont été implémentés, suivant les travaux présentés dans (57). Une autre solution a été expérimentée durant les travaux de cette thèse, avec notamment la librairie CUSP, cherchant à fournir des outils de haut-niveau pour le calcul creux. Au moment de la rédaction de cette thèse, quelques opérations BLAS creuses ont été intégrées au sien du SDK CUDA sous le nom de cudasparse. Là où CUSP propose de manipuler des structures de haut-niveau tel que les matrices creuses, cudasparse utilise directement des tableaux dynamiques présentés sous forme de pointeurs. Les essais utilisant CUSP se sont révélés concluant, à une exception

près : l'intégration industrielle pose problème de par l'implémentation de CUSP : des templates. Ces derniers ne proposaient pas la précompilation et imposaient la présence de la suite de compilation Nvidia sur la machine de test, même pour du calcul purement CPU. Il était donc nécessaire de modifier la librairie pour précompiler les structures intéressantes pour le calcul. Cette étape implique qu'à l'évolution suivante de la bibliothèque, il serait nécessaire d'apporter encore une fois les nombreuses modifications pour précompiler les matrices creuses. Ce n'est pas envisageable dans un cadre industriel. Pour cette raison, CUSP faisant tout de manière appel aux noyaux de (57), nous utiliserons directement ces noyaux au sein de nos matrices propres.

4.1.3. Matériel utilisé pour les expérimentations

Les tests de performance ont été effectués sur la machine Titane du Centre de Calcul Recherche et Technologie (CCRT). Ce calculateur, fourni par BULL et financé par GENCI, a été installé au CCRT début 2009. Il met à la disposition des utilisateurs une puissance de calcul crête de 100 Tflops obtenus grâce aux processeurs Intel Nehalem et environ 200 Tflops obtenus grâce aux serveurs GPU (graphics processing unit) Tesla de Nvidia. Ce calculateur hybride fut l'un des tout premiers installés au monde.

Le supercalculateur Titane est un cluster hybride. Pour la partie Intel, il dispose de 1068 nœuds de calcul et 24 nœuds dédiés aux entrées-sorties et à l'administration. Chaque nœud comprend 2 processeurs Intel Nehalem quadri-cœurs X5570 cadencés à 2.93 Ghz ainsi que 24 Go de mémoire. Pour la partie Nvidia, il comprend 48 serveurs Tesla S1070 disposant chacun de 4 GPUs C1060 à 4 Go de mémoire. Chaque serveur Tesla est rattaché à 2 nœuds de calcul par l'intermédiaire du bus PCI-express. Les nœuds de calculs sont interconnectés par un réseau Voltaire, basé sur la technologie InfiniBand DDR. Plus de détails sont donnés dans le tableau suivant :

Tableau 23. Description détaillée de la machine de test. La performance crête DP des Nehalems est calculée par la formule: fréquence * 2 (SSE) * 2 (FMA) * 4 (cores). En SP, 4 SSE sont posibles. Pour les GPUs en SP, c'est 1.296 (fréq) * 240 (cores) * 3 (flops). En DP, le GPU dispose d'une unité DP pour 8 SP, et n'effectrue que 2 flops par clock. [a] un nœud dispose de 24 Go de ram, soit 12 par quadcore.

Processeur	Intel Nehalem X5570	Nvidia Tesla C1060
Nombre cœurs	4	240
Fréquence (GHz)	2.93	1.296
GFlops (SP / DP)	93.76 / 46.88	933 / 78
Mémoire	12[a]	4
Bande passante mémoire mesurée (max) Go/s	19 (25.6)	75.2 (102)

4.1.4. Note sur la performance atteignable

Les algorithmes utilisés durant cette thèse sont tous limités par la bande passante mémoire. Si le problème est de petite taille, alors il est possible que l'exécution du solveur tienne dans les caches du processeur, et s'exécute très vite. Dès que le solveur sort des caches du processeur, alors la limitation principale devient la bande passante de la mémoire principale. Lorsque les calculs sont effectués sur des matrices denses, plus de 99% des opérations ont lieu dans le produit matrice vecteur. La performance de ce dernier est donc assez représentative de la performance globale de l'algorithme. Ainsi, pour une matrice d'ordre n, n^2 données sont utilisées pour effectuer $2n^2$ opérations. On constate un ratio de 2 opérations flottantes par élément.

Donc, en prenant l'exemple des Nehalems, si l'on dispose d'une bande passante mémoire de 19 Go/s, alors on peut effectivement transférer $^{19}/_{8} = 2.375$ GNombres double précision (64 bits = 8 octets). Par nombre, on, effectue deux opérations flottantes, d'où une performance maximum de 4.75 GFlops en double précision par processeur Nehalem. Cela implique une performance de 9.5 GFlops en simple précision (32 bits = 4 octets).

Pour les GPUs Tesla, la bande passante mesurée étant de 75 Go/s, on en déduit un transfert potentiel de 9.375 GNombres double précision, avec une performance de 18.75 GFlops à la clé. En simple précision, cette dernière est doublée à 37.5 GFlops.

4.1.5. Performances intranoeud hybrides

Nos premières expérimentations ont été faites avec des matrices denses facilement reproductibles : les matrices de Hilbert et Dingdong. La première est décrite dans la section 3.3.2.2.1, et la seconde est générée à partir de la séquence :

$$A_{ij} = \frac{1}{2 * (N - i - j + 1.5)}$$

La particularité de la matrice de DingDong est que ses valeurs propres sont presque toutes situées autour de $\pm\frac{\pi}{2}$. Ce problème de valeurs propres est ainsi extrêmement difficile à résoudre, au fur et à mesure que la taille de la matrice croit. Ces deux matrices sont symétriques, et la méthode de Lanczos semble plus appropriée pour résoudre le problème à valeur propre lié à ces dernières. La méthode d'Arnoldi peut s'appliquer à ces matrices, mais avec un coût supérieur à Lanczos, et une convergence moins bonne. Grâce à ces matrices, nous avons la possibilité de générer des matrices de grandes tailles, dont les valeurs propres sont connues ou facilement calculables. Ainsi nous pouvons évaluer dans différentes conditions la performance de nos algorithmes.

Les premiers tests ont été effectués avec le code compilé en mode processeur x86 pur, sur un nœud de 2 quadcores de la machine Titane. La taille de la matrice est de 12288x12288, le sous-espace est de taille 32, et les trois orthogonalisations sont utilisées, en simple ou double précision. Ces résultats sont présentés dans la figure 30. On voit la performance augmenter à chaque ajout de cœurs de calcul pour presque toutes les orthogonalisations. A gros grain, cette performance débute à 4 GFlops en SP et 2 GFlops en DP sur un seul cœur de calcul, pour atteindre 14 GFlops en SP et 7 GFlops en DP sur les huits cœurs de calcul. Des expérimentations entre processus MPI et threads ont été effectuées, et globalement l'utilisation de processus MPI explicite fournissait les meilleures performances. Cela rejoint les conclusions établies dans (58). Ajouter plus de processus MPI que de cœurs physiques de la machine ne permettait pas de gains de performance supplémentaires, et introduisait même des pertes de performances.

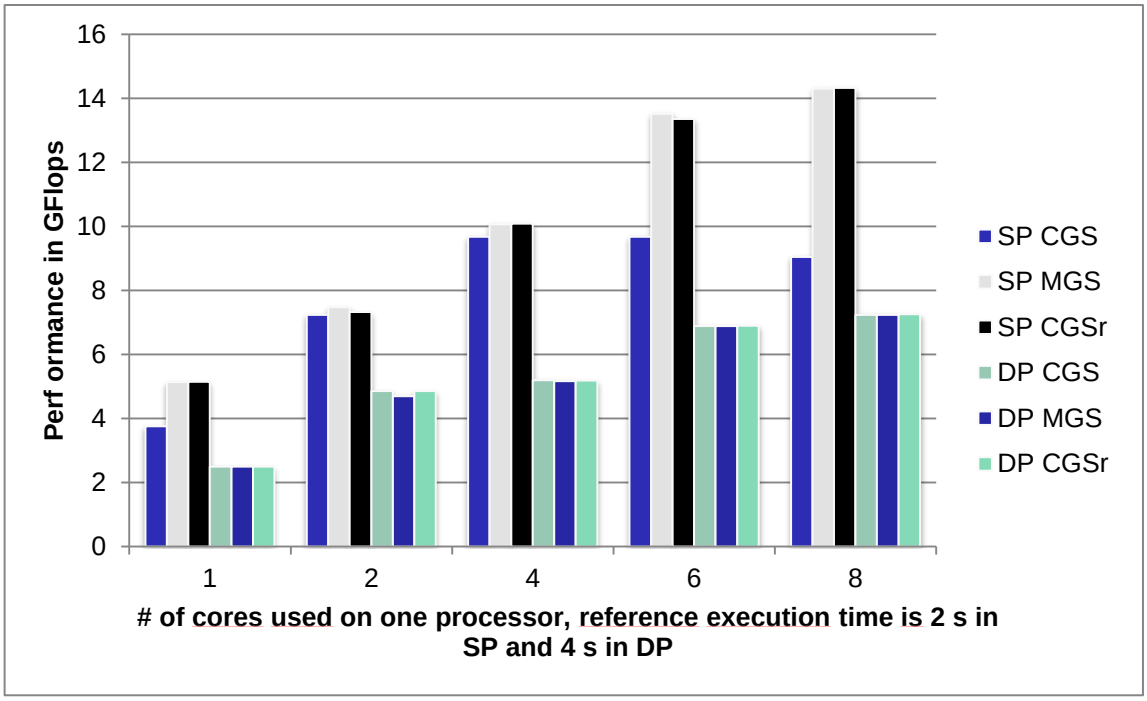


Figure 30. Performances d'Arnoldi redémarré avec différentes orthogonalisations de Gram-Schmidt sur un noeud de la machine Titane, en simple et double précision (SP&DP). La matrice d'Hilbert est de taille 12288x12288, dense. Le sous-espace est de taille 32.

L'orthogonalisation CGS a apparemment moins profité de la parallélisation que les autres méthodes, sans raison évidente. MGS utilise les caches de manière efficace par nature, et CGSr effectue deux fois plus d'opérations vectorielles, ce qui peut se traduire par une meilleure réutilisation des données. Mis à part ces raisons expliquant la meilleure performance de MGS et CGSr, aucune autre explication n'a été trouvée. Globalement, le temps de calcul diminue de 4 secondes à 1.28 secondes en double précision, et de 2 à 0.68 secondes en simple précision.

Si l'on s'intéresse à l'efficacité par rapport à la bande passante mémoire effective totale, alors on obtient le tableau suivant, en prenant comme performances maximum 4.75x2 et 9x2 GFlops en double et simple précision respectivement :

Nombre coeurs	1	2	4	6	8
Eff CGS SP / DP	21% / 25%	39% / 44%	55% / 51%	54% / 68%	47% / 76%
Eff MGS SP / DP	51% / 25%			72% / 68%	78% / 76%
Eff CGSr SP / DP					

Nous constatons une efficacité finale de presque 80% en double précision pour les trois orthogonalisations et pour MGS et CGSr en simple précision. Pour CGS en simple précision, l'efficacité atteinte est d'environ 50%. Pour un problème limité par la bande passante, obtenir presque 80% de la performance maximum est un score plutôt honorable.

Après avoir essayé le calcul sur plusieurs multicoeurs au sein d'un nœud de calcul, nous nous intéressons maintenant à la performance des cartes graphiques à disposition dans la partie hybride de Titane. Nous exécutons le même benchmark, avec cette fois-ci moins de choix pour la configuration des calculs : utilisation complète de la carte graphique.

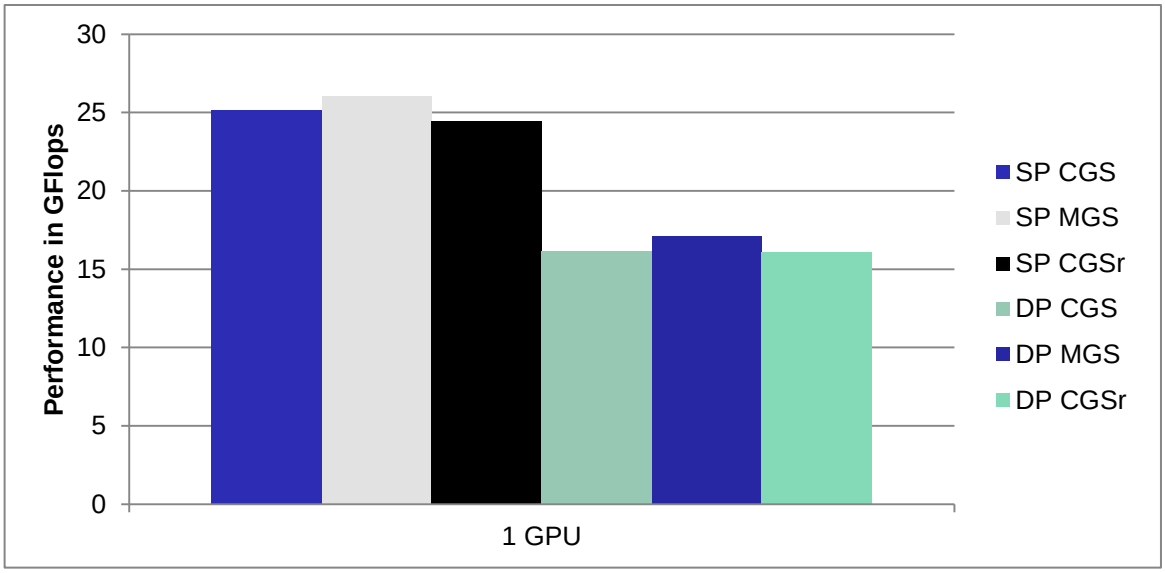


Figure 31. Performances de différentes orthogonalisation de Gram-Schmidt sur un GPU de la machine hybride Titane, en simple et double précision (SP&DP).

La performance est stable pour les trois orthogonalisations selon la précision choisie : 24 à 26 GFlops en simple précision et 16 à 17 GFlops en double précision. Cela implique l'efficacité présentée dans le tableau ci-dessous, en utilisant 37.5 et 18.75 GFlops comme maximum en simple et double précision respectivement :

	SP	DP
Eff CGS	66%	85%
Eff MGS	69%	91%
Eff CGSr	64%	85%

En termes d'efficacité, le noyau de calcul BLAS simple précision manque certainement d'optimisation. Les données à traiter sont très régulières, et on attend plus une efficacité telle que les 85-90% obtenus en double précision que les 65 à 70% obtenus ici. Cependant, le temps de calcul effectif est tout de même bien réduit par rapport aux processeurs x86 : 0.38s en simple précision et 0.58 en double précision. L'accélération par rapport au code exécuté en mode pur processeur est présentée dans le tableau suivant :

	2xquadcores	1xC1060	Speed-up GPU
Temps SP	0.68s	0.38s	1.78x
Temps DP	1.28s	0.58s	2.21x

Concrètement, un seul GPU est capable d'exécuter notre solveur Arnoldi environ deux fois plus vite qu'un nœud de deux quadcores. Autrement dit, un GPU C1060 équivaut presque à 2 nœuds de la machine Titane, soit quatre quadcores ! Ces résultats sont plutôt encourageants, et nous allons voir si cette accélération se produit toujours en utilisant des structures plus irrégulières dans le cadre de calculs creux.

Dans l'algorithme d'Arnoldi, presque rien ne change lorsque l'on utilise une structure creuse : seul le produit matrice vecteur est touché. Cependant, ce changement mineur dans la structure de la matrice a des impacts non négligeables sur la performance globale et la répartition de cette dernière au sein des calculs. Nous avons vu cela dans la section 3.3.2.2.2, au tableau 17, et nous l'illustrons au travers du tableau suivant :

	Dense	Creuse1	Creuse2
non zéros	1,00E+08	1,00E+05	5,00E+05
ordre	1,00E+04	1,00E+04	1,00E+04
matrice-vecteur %	99,65%	21,98%	58,48%
op vectorielles %	0,35%	79,02%	42,52%

Nous comparons la répartition des opérations flottantes entre une matrice dense et des matrices creuses de même ordre : 10000. La matrice creuse Creuse1 possède en moyenne 10 éléments par ligne, et la matrice Creuse2 50 éléments par ligne. L'influence de la performance du produit matrice-vecteur va en augmentant suivant sa densité: pour Creuse1 les opérations matricielles représentent un cinquième du calcul, et pour Creuse2, 4/7. En augmentant le nombre d'éléments par ligne à 100, ces opérations demandent alors les trois quarts du calcul global.

De plus, au-delà de la répartition des calculs, la performance du produit creux a de fortes chances d'être bien moins bonne que pour une matrice dense. Selon le format creux, de nombreuses indirections sont nécessaires, ce qui implique des opérations de calculs d'adresse et des transferts mémoires supplémentaires et potentiellement des temps d'attentes de la fin de chargements mémoire. La figure 32 montre le résultat d'expérimentations avec la matrice de Lin provenant du site MatrixMarket, sur un seul processeur de la machine Titane. Notons bien que seuls quatre cœurs d'un processeur sont utilisés et non huit cœurs d'un nœud de deux quadcores. Les caractéristiques de cette matrice sont décrites dans le tableau 16, page 52. Le noyau de calcul utilisé pour le multicoeur est un CSR classique, multithreadé via OpenMP et profitant de la mémoire distribuée via MPI. Dans notre cas, c'est encore MPI qui offrait la meilleure performance, avec ici 2.4 GFlops en simple précision et 1.6 GFlops en simple précision. La version GPU, utilisant CUBLAS et nos noyaux spmv inspirés de (57), offre une performance de 10.2 GFlops en simple précision et 6.3 GFlops en double précision. Le format de matrice utilisé pour le GPU est CSR, avec activation de la mémoire de texture et calcul de type vectoriel sur chaque ligne.

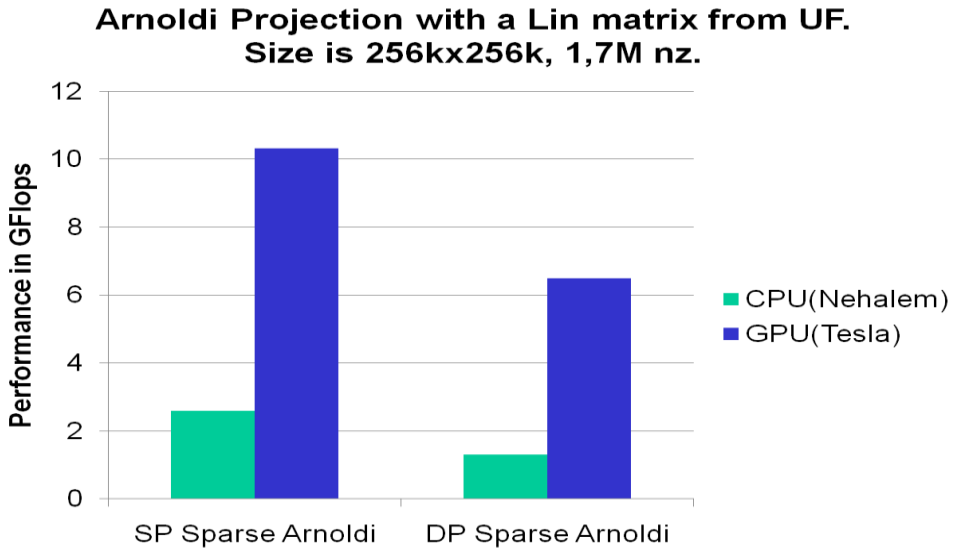


Figure 32. Projection d'Arnoldi sur une matrice creuse, utilisant un processeur multicoeur ou un accélérateur graphique.

Le speed-up obtenu est d'environ 4x avec le GPU, et l'efficacité des deux matériels est de 25 à 34% sur CPU et 26 à 34% sur GPU. Étonnement, alors que les implémentations des

noyaux de calcul et les matériels sont complètement différents, l'efficacité des calculs est presque la même sur CPU ou accélérateur. Plusieurs essais ont ensuite été conduits sur différentes matrices creuses. Alors que la performance CPU restait stable, celle sur GPU variait d'une accélération de 8x à un ralentissement par rapport au calcul CPU. Quelques expérimentations sur les formats de matrices nous ont amené à avoir de meilleurs résultats selon le format de matrice, de manière notable. Nous avons donc pensé à une stratégie d'autotuning pour obtenir les performances optimales sur GPU de manière automatique.

4.1.6. Autotuning du produit matrice-vecteur

Dans la section 3.2, nous avons vu que selon les générations de carte Nvidia, la précision double pouvait ne pas être supportée, ou partiellement. Au-delà de l'aspect arithmétique IEEE, d'autres caractéristiques matérielles évoluent. Essentiellement, le contrôleur mémoire a gagné en souplesse d'accès. Avec les premières cartes, il fallait accéder la mémoire de manière contigüe ou très régulière. Ainsi, si lors de la lecture un seul thread GPU lisait une donnée différemment des autres threads, alors la performance pouvait être divisée par deux. Pour chaque différence, une division peut avoir lieu, impactant très fortement la performance dans le cas de lectures très irrégulières. Sur la génération 1.3 des cartes (S1070 et C1060 par exemple), les contrôleurs mémoires ont été améliorés, permettant plus de souplesse et moins de pénalité en cas d'accès irréguliers. La génération 2.x (Tesla C2050 et plus) a encore amélioré la performance du contrôleur mémoire, et surtout ajouté une autre amélioration matérielle : deux niveaux de caches, L1 et L2. Le cache L1 est partagé par un multiprocesseur regroupant plusieurs unités de calcul (à la manière d'un multicoeur), et est partagé en deux parties : la partie destinée au programmeur et l'autre partie, gérée automatiquement par le processeur. Le cache L2, lui, est partagé par tous les multiprocesseurs, et agit automatiquement. L'intérêt des caches, comme sur processeur classique, est de cacher les accès mémoire en gardant dans les caches les données nécessaires aux calculs.

Pour ces détails techniques, un produit matrice vecteur creux, à implémentation et bande passante équivalente, n'obtiendra pas la même performance selon le type de matériel GPU utilisé. Par exemple, nous présentons les résultats du produit matrice vecteur creux pour la matrice nlpkkt80 décrite dans le tableau 24, provenant de l'Université de Floride, sur les GPUs de la machine Titane.

Format	CSR scalar	CSR vector	COO	ELL	HYB
Perf . (GFlops)	2	6.5	3.5	17	17
Speed-up	Réf.	3.25x	1.75x	8.5x	8.5x

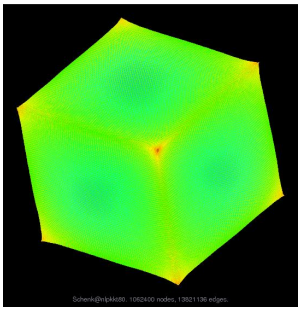
La performance varie de 2 GFlops à 17 GFlops selon le format, soit un speed-up de 8.5x. Sur un GPU de station de travail, une quadro FX 4600, qui ne supporte pas la double précision, le speed-up est de 5x au maximum entre CSR vector et CSR scalar. Dans (59), nous avons proposé une recherche automatique de ce format optimal, à l'exécution.

L'idée appliquée est très basique, et peut être grandement améliorée comme nous le verrons par la suite. Pour une exécution d'Arnoldi, la matrice n'est utilisée qu'en lecture, autant de fois qu'il y a d'itérations de la méthode. Il y a donc un intérêt à optimiser sa structure pour une exécution globale plus rapide. Nous proposons de le faire juste avant l'exécution du solveur, à la construction de la matrice. Cette solution cherche la meilleure solution pour chaque matrice, avec un léger surcoût. Dans (60), l'utilisation d'une

métaheuristique permet, par analyse de la structure de la matrice, de trouver le format optimal avec un taux de réussite moyen de 66%.

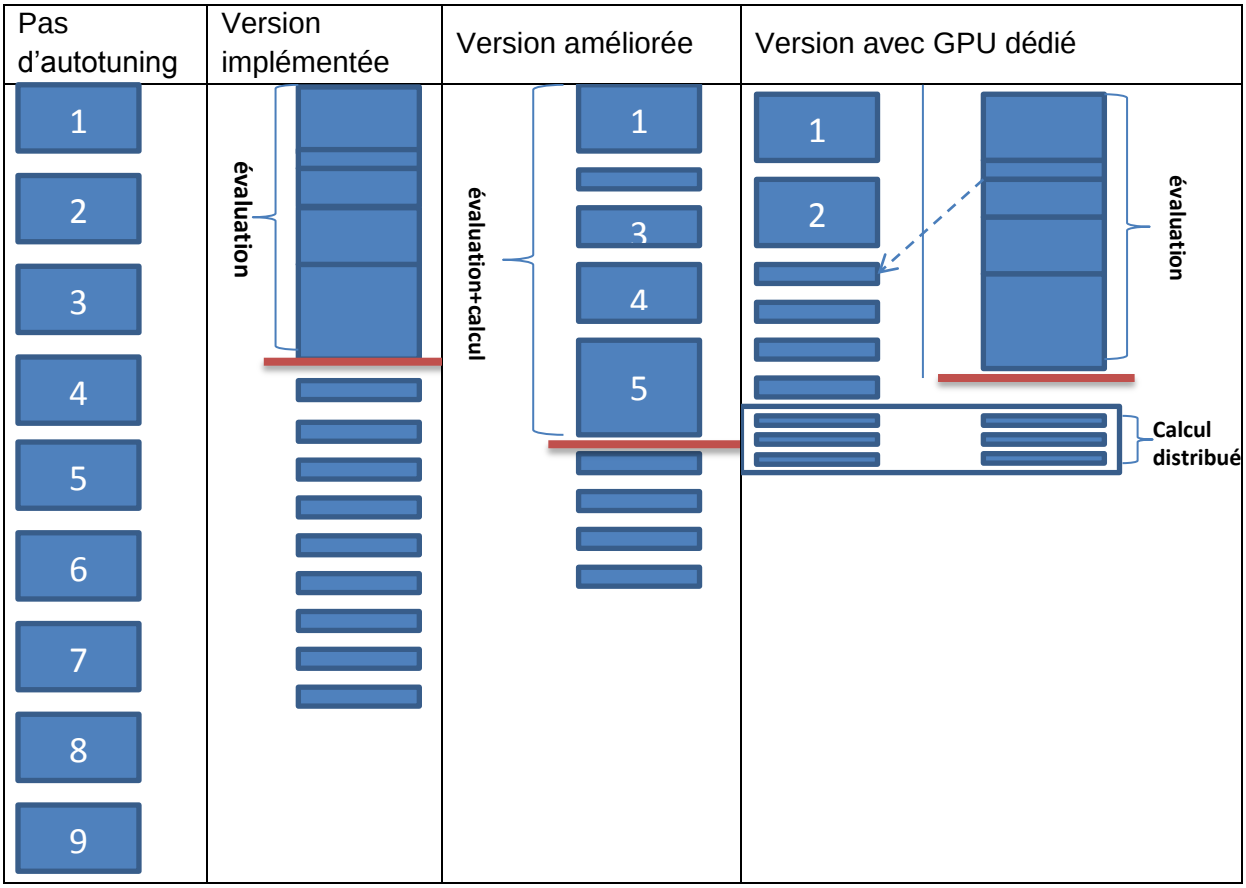
Dans notre cas, un par un, nous mesurons la performance des différents formats de matrices à disposition dans notre bibliothèque. Ces formats incluent tous les noyaux optimaux proposés dans (57). Nous choisissons alors le format de matrice le plus performant, et instancions cette matrice pour notre solveur. L'avantage de cette approche est d'utiliser le format de matrice optimum pour notre GPU. L'inconvénient est un léger surcoût lié à l'évaluation avant exécution du solveur. Cependant, ce type d'approche peut être amélioré en tirant partie des transferts mémoires asynchrones des GPUs.

Tableau 24. Description de la matrice nlpkkt80 de l'Université de Floride.

Matrice	Nlpkkt80
Ordre	1.062.400
Non-zéros	28.192.672
Problème	Problème d'optimisation
Graphe	

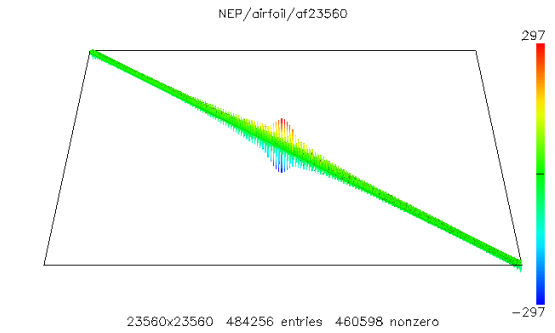
Ainsi, un autre schéma pourrait être le suivant. Nous lançons l'exécution du solveur avec le format de matrice par défaut, et mesurons sa performance. Pendant les calculs effectués à l'itération 1 de la projection (et l'itération 1 du solveur), nous instancions un autre format de matrice de manière asynchrone. A l'itération 2 de la projection (et toujours la première du solveur), nous utilisons alors ce format pour effectuer le produit matrice vecteur, tout en mesurant la performance. Pendant l'itération 2 de la projection, nous pouvons alors charger un troisième format de matrice, et continuer ce processus jusqu'au parcours de toute la bibliothèque de produit matrice vecteur. Une fois tous les noyaux benchmarkés, nous choisissons le meilleur pour les itérations de la projection courante et des projections restantes nécessaires. Avec cette solution, le temps passé à benchmarker les noyaux est utilisé à bon escient, avec certes une performance variable. Une autre solution, dans le cas où plusieurs GPUs sont disponibles, et d'associer un GPU à la phase d'évaluation des produits matrices vecteurs pendant que les autres résolvent le problème à valeur propre. Dès qu'un produit plus optimal que celui par défaut est identifié, alors les GPUs participant au solveur changent le format de matrice de manière asynchrone. L'autre choix est d'attendre que le GPU dédié à l'évaluation ait fini sa tâche pour que les autres solveurs choisissent le format optimal. Logiquement, le solveur dédié à l'évaluation peut rejoindre les calculs des autres GPUs par la suite. Le tableau 25 montre ces différentes approches. Il paraît assez clair que la version améliorée va obtenir un temps d'exécution meilleur que la version implémentée. Ceci dit, pour l'exemple, seules 9 itérations sont nécessaires pour résoudre le problème. Si ce nombre était plus important, alors l'impact de l'évaluation deviendrait de plus en plus négligeable, au vu du gain de temps obtenu.

Tableau 25. Différentes approches pour l'autotuning du produit matrice-vecteur. A noter, dans le dernier cas, le calcul distribué avantage cette solution.



Par contre, trancher entre la solution améliorée et avec GPU(s) dédié(s) est plus délicat. Dans l'hypothèse où l'on allouerait plusieurs GPUs aux deux versions, et en admettant un passage à l'échelle correct, alors plusieurs scénarios sont possibles. La version améliorée bénéficiera d'une réduction globale liée au nombre de GPUs. Dans le cas avec GPU dédié, la phase de calcul distribué s'exécutera de la même manière sur les itérations 7 et suivantes que pour la version améliorée. Si l'on admet une répartition des GPUs uniforme entre les étapes 1 à 6 et la partie évaluation de la version avec GPU dédié, alors la réduction du temps ne sera que d'un facteur 2. L'évaluation à priori de la version avec GPU dédié est également difficile car dans l'exemple choisi, le meilleur produit matrice vecteur est trouvé dès le deuxième test. Si cela avait été au dernier test, alors la version avec GPU dédié aurait certainement été moins performante que la version améliorée.

Dans tous les cas, ces pistes sont à explorer dans la suite de cette thèse, et les premiers résultats obtenus avec la version basique implémentée sont plutôt encourageants. Nous utiliserons les matrices nlpkkt80 et AF23560. Cette dernière est décrite dans le tableau suivant :

Matrice	AF23560
Ordre	23560
Non-zeros	460.598
Problème	Eigenvalue problem
Graphe	

On obtient les performances suivantes sur l'exécution complète d'un ERAM de sous-espace égal à 16 avec orthogonalisation CGSr, en GFlops et avec les GPUs de la machine Titane :

	AF23560	Nlpkkt80
Processeur quadcore intel Nehalem	2.33	1.55
Tesla C1060 CSR	0.5 (spmv) – 1.2 (eram)	1.8 (spmv) – 2
Tesla C1060 Auto	7.5 (spmv) – 1.7 (eram)	17.5 (spmv) – 13 (eram)
Speed-ups Auto / CSR	15 (spmv) – 1.42 (eram)	9.72 (spmv) – 6.5 (eram)

Pour les GPUs, la première performance indique les GFlops délivrés par le produit matrice vecteur creux (spmv) selon le format de matrice. Le deuxième nombre représente la performance globale du solveur ERAM. On note une amélioration d'un facteur de 15 pour la matrice AF23560 lorsque l'on active l'autotuning, et 9.72 pour la matrice nlpkkt80. Cependant, cela se transcrit par une augmentation de la performance du solveur d'un facteur 1.42 pour AF23560 et 6.5 pour nlpkkt80. Cela est dû en partie à la structure même de ces matrices et la répartition des calculs entre opération sur la matrice et vectorielles. En effet, les opérations BLAS1 peuvent être prédominantes dans les calculs. Pour nlpkkt80, les opérations matricielles sont plus dominantes que pour AF23560. L'ordre de cette dernière est 23.560, d'où des opérations effectuées sur de « petits » vecteurs, et une performance fortement impactée par la latence de la mémoire et le bus PCI-Express reliant la carte graphique au processeur central. L'ordre de nlpkkt80 est beaucoup plus important, 1.062.400, ce qui est assez large pour mieux recouvrir la latence du bus PCI-Express et donc d'obtenir une meilleure performance globale. Malgré tout, l'autotuning apporte un gain de performance appréciable, qui potentiellement s'appliquera dans un environnement où plusieurs GPUs sont utilisés conjointement.

4.2. Parallélisation multi-nœuds avec accélérateurs

4.2.1. Utilisation de cluster

Après de premiers résultats encourageants, nous nous intéressons maintenant à l'utilisation de plusieurs nœuds de calcul ou plusieurs GPUs. La machine Titane a été mise à contribution pour cette tâche, fournissant les résultats présentés dans la figure 33, sur une matrice DingDong dense, d'ordre 12288.

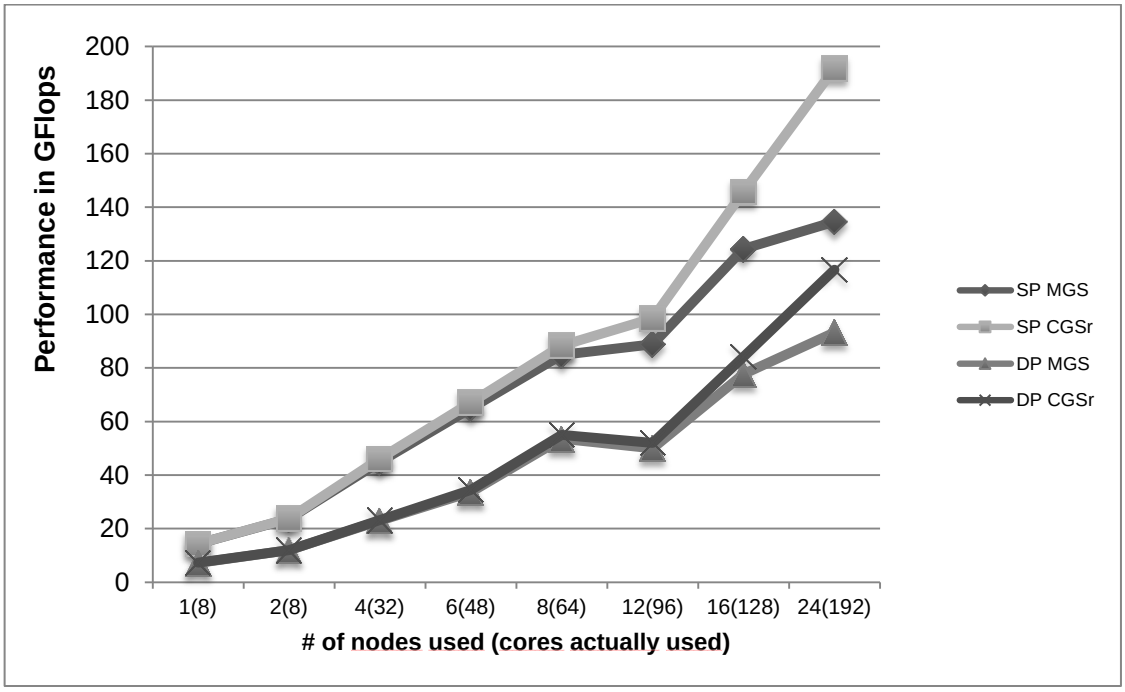


Figure 33. ERAM exécutés sur la machine Titane, utilisant de 1 noeud à 24 noeuds de 2x4 coeurs. les orthogonalisations MGS et CGSr sont utilisées, en simple et double précision.

Sur cette figure, la performance de l'exécution de la méthode d'Arnoldi débute à 14 GFlops et 7.3 GFlops, simple et double précision respectivement. Les deux orthogonalisations passent à l'échelle de manière régulière jusque 8 nœuds de calcul ou 64 cœurs, pour une performance de 88 en simple précision et 55 GFlops en double précision. Il y a ensuite un plateau dans la performance, ou une baisse selon l'orthogonalisation, lorsque l'on utilise 12 nœuds. Ce phénomène peut être expliqué par le calcul de l'efficacité dans le tableau ci-dessous. Par la suite, la performance continue d'augmenter, de manière plus limitée pour MGS, et très satisfaisante pour CGSr. Une des raisons possibles est la configuration de la machine Titane. Les switchs réseaux de cette dernière sont organisés hiérarchiquement, avec un niveau le plus bas à 16 nœuds. Autrement dit, à partir de 16 nœuds, la probabilité de devoir communiquer entre deux switchs plus lents lors de l'exécution du solveur en parallèle augmente fortement, pour être égale à un lorsque l'on utilise 24 nœuds. MGS effectue autant d'opérations de réduction que CGSr pour calculer les produits scalaires, mais elles ne peuvent pas être groupées pour MGS. Ainsi, pour un sous-espace de taille 16, MGS requiert 16*17/2 = 136 réductions, alors que CGSr n'en demande que 2*16 = 32. Au final, lorsque l'on utilise 24 nœuds de calculs, l'efficacité évolue en diminuant, comme présenté dans le tableau suivant :

Noeuds	1	2	4	6	8	12	16	24
Eff SP MGS	100%	85%	82%	80%	79%	58%	65%	57%
Eff SP CGSr						53%	55%	40%
Eff DP MGS		83%	81%	80%	95%	60%	73%	68%
Eff DP CGSr							68%	54%

On constate, au moment du plateau ou de la baisse de performance, que l'efficacité pour 8 nœuds de calcul est extrêmement bonne en double précision lors de l'exécution du solveur. Nous n'avons pas d'explication exacte de cette soudaine efficacité, mais elle explique en partie l'apparente baisse de performance à 12 nœuds de calcul en double précision. C'est en fait la performance à 8 nœuds qui est très ou trop bonne.

Après ces expérimentations sur processeur x86, où la scalabilité est correcte, nous ajoutons maintenant l'accélération GPU à ces mêmes calculs. Nous allons ainsi voir comment le calcul sur GPU passe à l'échelle. La figure 34 montre les résultats avec plusieurs GPUs sur la machine Titane.

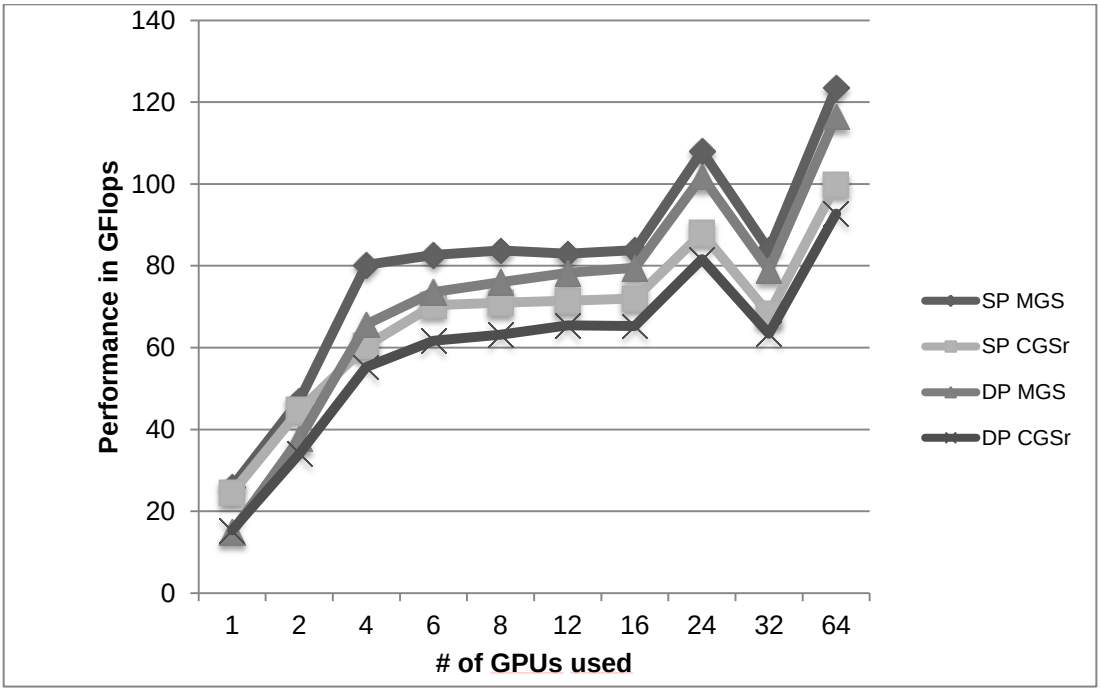


Figure 34. Performances multi-GPUs d'Arnoldi redémarré en dense, avec une matrice DingDong d'ordre 12288.

Le comportement de la scalabilité est complètement différent de celui avec plusieurs nœuds de calculs. La scalabilité est excellente de 1 à 4 GPUs, très proche de 100%, puis elle semble stagner et à peine s'améliorer. C'est un comportement plutôt déroutant, étant donné que la scalabilité sur CPUs était acceptable. La raison se trouve peut-être au niveau de la nature même des GPUs : ils peuvent être vus comme des accélérateurs vectoriels, à cause du lien PCI-Express qui les unit au processeur hôte. En effet, chaque lancement de noyau de calcul BLAS ou chaque retour de calcul vers le CPU tel qu'un produit scalaire implique l'utilisation de ce lien à la latence relativement élevée. Donc, si les noyaux de calculs s'exécutent sur des vecteurs de taille réduite, alors la performance par noyau diminue fortement. C'est tout à fait le phénomène qui pouvait avoir lieu lors de l'utilisation de machines vectorielles : pour cacher la latence mémoire, il fallait réussir à utiliser de longs vecteurs. Pour compléter la figure précédente, voici les tailles de vecteurs (ou nombres de lignes des sous-matrices) lorsque le calcul est réparti sur plusieurs GPUs :

GPUs	1	2	4	6	8	12	16	24	32	64
Matrix rows / GPU	12288	6144	3072	2048	1536	1024	768	512	384	192
Equiv. Order	12288	8689	6144	5016	4344	3546	3072	2508	2172	1536

On voit qu'au moment où la performance décroche dans la figure précédente, la taille des vecteurs utilisés est de 2048, avec une matrice qui fait donc 2048x12288 éléments, soit environ 25 millions de non zéros avec un format rectangulaire. Le tableau proposé ci-dessous montre les performances d'Eram avec une taille de matrice carrée croissant, de 512 à 8192. La performance augmente avec la taille pour s'approcher de la performance crête de 18.5 GFlops. A partir de ces éléments, on peut déduire que la performance unitaire de chaque GPU se trouve très proche de celle obtenue avec une matrice carrée d'ordre 4096 pour 6 nœuds, soit environ 10 GFlops par GPU. Cela se traduit par une performance de cet ordre sur le graphe : 60 GFlops environ.

Matrix order	512	1024	2048	4096	8192
1 GPU (Gflops)	0.34	1.12	4.03	9.4	16.22

Par ailleurs, la forme rectangulaire des matrices ne va pas dans le sens de la performance, comme (61) le montre. Pour expliquer un peu plus cette dégradation de la performance, nous allons effectuer un test dit de weak scaling, où la taille des problèmes va être fixe pour chaque sous-domaine. Ainsi, à l'ajout d'un élément de calcul (CPU ou GPU), alors la taille du problème carré global va être augmentée. L'avantage est de ne plus être influencé par la taille du problème qui diminue lorsque l'on ajoute des GPUs par exemple, mais à cause de la répartition en blocs de lignes des sous-domaines, les matrices vont avoir tendance à être de plus en plus rectangulaires à chaque ajout d'accélérateur, biaisant potentiellement les résultats obtenus. Les résultats sont présentés dans la figure 35.

Cette figure nous présente plusieurs résultats intéressants. Concernant les multicoeurs, on constate un comportement de la scalabilité parfait, avec une efficacité de 100%, et un temps de calcul constant lorsque la taille du problème augmente à l'ajout d'une unité de calcul. Parallèlement au temps constant, le temps de communication MPI va en augmentant régulièrement également. Au niveau des GPUs, l'efficacité est particulièrement moins bonne, car la tendance générale du temps d'exécution va en augmentant avec la taille du problème. Cela se traduit sur l'efficacité, qui est de l'ordre de 50 à 70%. Il faut noter que sur les classements top500 (62) établis durant cette thèse, les machines à base d'accélérateurs GPU atteignaient une efficacité finale de l'ordre de 50 à 60% lors de l'exécution du LINPACK (63). Nous ne sommes certes pas à la même échelle, avec l'utilisation de quelques dizaines de GPUs contre plusieurs milliers pour les LINPACK exécutés, mais la classe de problème traitée lors de l'exécution du LINPACK est plus dense en calcul que les solveurs ERAMs. Nous sommes donc plus contraints par les bandes passantes mémoires et le réseau que LINPACK.

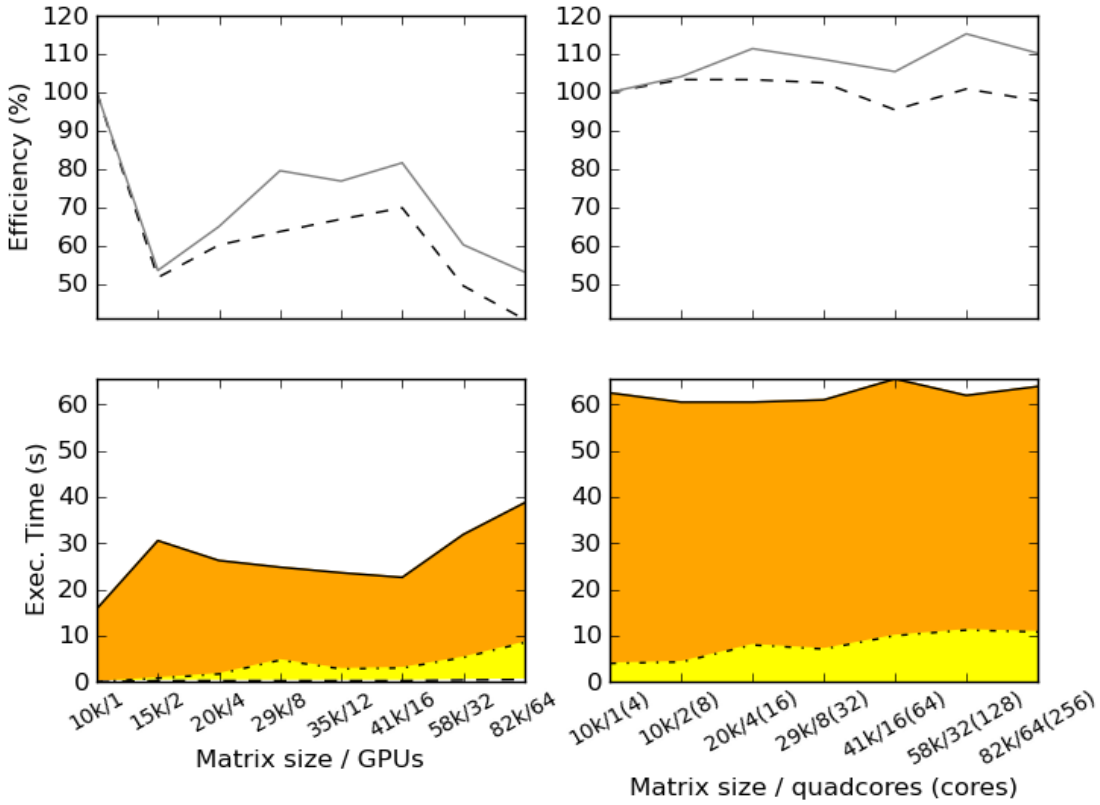


Figure 35. Weak scaling du solveur ERAM sur plusieurs GPUs et plusieurs CPUs. Les graphes supérieurs montrent l'efficacité, et les graphes inférieurs le temps de calcul. En pointillés est montrée l'efficacité de calcul totale, et en trait plein l'efficacité sans prendre en compte les communications. Pour les graphes inférieurs, la zone située en dessous du trait plein représente le temps de calcul complet, et la zone située sous le pointillé le temps de communication MPI. La matrice est Dingdong, d'ordre 10240 sur un seul processeur, et le sous-espace est fixé à 16, avec orthogonalisation CGSr.

Pour les GPUs, il faut noter que les matrices sont à tendances rectangulaires, avec donc un potentiel impact sur la performance obtenue, par rapport à une matrice carrée. Cependant, dans des conditions semblant optimales, l'efficacité de calcul sur GPU est plutôt limitée. Une partie de l'explication tient au temps de communication MPI. Il est le même que pour les CPUs, mais le temps de calcul global est réduit d'un facteur 5x environ par rapport au temps CPU. La part de temps passé dans les communications MPIs est donc plus importante, et un test de strong scaling permettra de s'en rendre compte, sur la figure 36.

Sur cette figure, on constate la bonne scalabilité des processeurs Nehalems, dont la performance va en augmentant régulièrement de 1 quadcore à 64 quadcores, avec une efficacité finale d'environ 62% en utilisant 8 nœuds de Titane (16 processeurs, 64 cœurs) contre un seul nœud (2 processeurs, 8 cœurs). Pour les GPUs, la scalabilité est satisfaisante jusque 4 cartes, puis même si l'on gagne toujours de la performance, l'efficacité finale avec 16 cartes est de 38%, alors qu'elle est légèrement supérieure à 100% sur 4 GPUs.

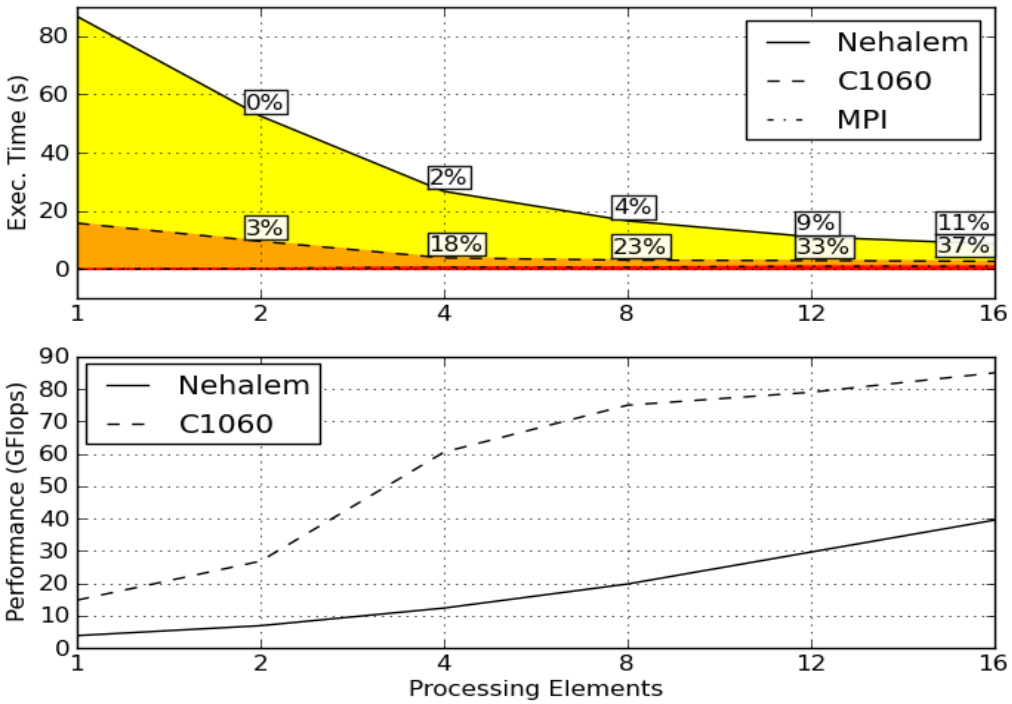


Figure 36. Test de strong scaling d'ERAM sur plusieurs GPUs et multicoeurs. La matrice testée est Dingdong, d'ordre 12288. L'orthogonalisation utilisée est CGSr avec un sous-espace de taille 16.

Au final, avec plusieurs GPUs, il faut donc s'attendre à un comportement différent par rapport à la scalabilité habituelle de l'algorithme, avec potentiellement de bonnes performances lorsque la taille du problème traitée par chaque GPU est suffisamment grande. Si la taille globale du problème est fixe, alors il est possible d'atteindre une limite au-delà de laquelle ajouter un GPU ne fournit plus de performance, voir même dégrade les performances. Parallèlement, la meilleure scalabilité sur CPU fait qu'au bout d'un certain nombre de nœuds, les deux courbes de performances vont se croiser, et le CPU deviendra un meilleur choix pour le calcul.

Nos expérimentations sur le calcul creux avec autotuning illustrent d'ailleurs cette tendance, dans la figure 37. Ainsi, La performance d'un GPU utilisant le format de stockage CSR est légèrement supérieure à celle d'un multicoeur. La bonne scalabilité jusque 4 GPUs permet au GPU d'être plus rapide que 4 processeurs Nehalems sur 2 nœuds de calcul. Par la suite, la scalabilité sur GPU se dégrade, alors que sur CPU elle reste stable, pour atteindre une performance de 12 GFlops avec 16 quadcores, contre 6.5 GFlops avec 16 GPUs.

Lorsque l'on active l'autotuning, la performance change sensiblement, à l'avantage du GPU. La scalabilité globale reste exactement la même que celle de l'exécution CSR, excepté la différence d'échelle. Cette dernière débute à 14 GFlops et non 2 GFlops, et culmine à 44 GFlops contre 6.5 précédemment. L'accélération avec 16 GPUs par rapport à un seul GPU est de 3.25 en CSR, et 3.14 en autotuné. Une remarque au sujet de l'autotuning : chaque processus MPI exécutant le solveur effectue sa propre évaluation du meilleur produit matrice vecteur, selon ses données de la matrice. Ainsi, si la matrice n'est pas régulière dans sa distribution, alors chaque sous-matrice peut avoir une forme complètement différente au vu de notre schéma de répartition des sous-domaines. Ainsi, chaque GPU peut utiliser un format de stockage de la matrice différent de celui des autres. Cela permet potentiellement d'atteindre une performance meilleure que dans le cas où le format serait identique pour tous les sous-domaines.

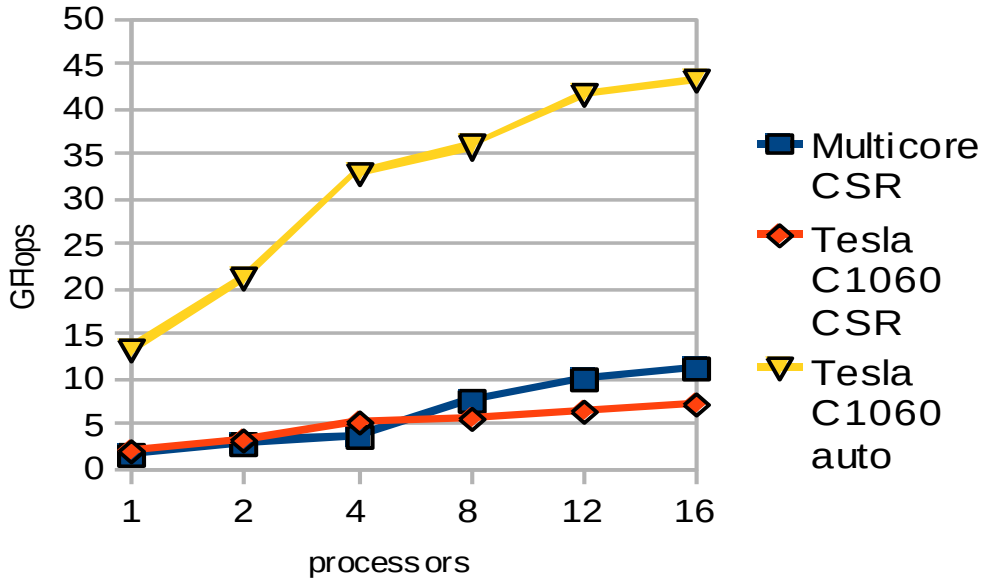


Figure 37. Performances multi-GPUS et multicoeurs d'ERAM auto-tuné sur la machine Titane, avec la matrice nlpkkt80 et un sous-espace de taille 16, CGSr.

Après ces tests effectués sur un petit nombre de processeurs et de GPUs, nous voyons que la scalabilité permet une amélioration des performances certaine. Cependant, l'avenir du calcul est dans le parallélisme massif, à plusieurs milliers de cœurs, potentiellement dans une seule machine comme vu dans le chapitre d'introduction. A cet effet, nous proposons d'exécuter ERAM sur un plus grand nombre de nœuds, avec une précision de calcul très grande sur une matrice Dingdong très difficile à résoudre.

4.2.2. Passage à l'échelle d'un supercalculateur

Durant nos expérimentations d'ERAM, nous avons eu accès à trois supercalculateurs mondiaux en cours d'installation ou déjà en capacité industrielle. Le premier, Titane, a déjà été décrit précédemment. Par la suite, nous avons pu accéder à la machine Hopper dans le cadre d'une collaboration avec le Lawrence Berkeley National Laboratory. Vers la fin de la thèse, les programmes ont pu être expérimentés sur une version en cours d'installation du supercalculateur européen Curie. Le tableau suivant résume brièvement les différences entre ces machines.

Tableau 26. Caractéristiques principales des machines Titane, Hopper et Curie. Le nombre de noeuds de Curie est une projection du total attendu. Le nombre de coeurs, de processeurs, la mémoire Ram, la bande passante et la puissance sont donnés par noeud de calcul. La Ram est en Go, la bande passante en Go/s. Le modèle de processeur AMD est Magny-Cours.

	Nœuds	Modèle	Cœurs	Procs	Ram	BP	Puiss.	Ram/cœur	Réseau
Titane	1068	Intel X5570	8	2	24	~50	~94	3 Go	Inf. DDR
Hopper	6384	AMD MC*	24	2	32	~80	~202	1,3 Go	Cray Gemini
Curie	5400	Intel X7560	32	4	128	~100	~375	4 Go	Inf. QDR

Sur la machine Titane, nous instancions un solveur ERAM utilisant l'orthogonalisation CGSr, avec un sous-espace de taille 32. La matrice DingDong est d'ordre 12288, et la précision demandée est 10^{-13}. Les tests sont effectués en double précision, et la figure 38 illustre la scalabilité d'ERAM.

Le temps d'exécution sur un nœud de 2 processeurs à 4 cœurs est d'un peu plus d'une heure, et se réduit presque linéairement jusque 256 secondes avec 15 nœuds. Au-delà de 16 nœuds, nous traversons des équipements réseaux possédant des liens plus lents, et l'amélioration de performance se réduit très fortement, pour un meilleur temps obtenu avec 50 nœuds ou 400 cœurs : 158s. Lorsque l'on augmente encore le nombre de nœuds, alors les communications deviennent prépondérantes sur les calculs, et le temps augmente. On peut d'ailleurs s'intéresser à la courbe d'efficacité des calculs, présentée dans la figure 39.

Nous constatons sur cette figure la très bonne efficacité de la méthode jusque 15 nœuds, avec ensuite une perte de performance régulière. Avec un grand nombre de nœuds, nous perdons en efficacité. Les communications prennent le pas sur la réduction du temps de calcul, et le gain en calcul est contrebalancé par l'augmentation des transferts réseaux. Pour illustrer ce fait, nous avons pu exécuter les mêmes calculs sur la machine Hopper du National Energy Research Scientific Computing Center (NERSC). Chaque nœud de cette machine est constitué de 2 processeurs AMD 12-cœurs Magny-Cours équipés de 32 Go de mémoire vive. Le réseau utilisé est Infiniband QDR, qui offre le double du débit du réseau de Titane. On s'attend donc à une meilleure scalabilité des calculs sur Hopper. La figure suivante nous montre que c'est le cas, où nous exécutons le même cas que sur Titane, avec différentes orthogonalisations.

D'autres expérimentations ont pu être menées au travers d'un Preparatory Access sur la machine Curie du Très Grand Centre de Calcul. Cette dernière est en phase expérimentale de déploiement au moment de la rédaction de cette thèse. Elle est composée de 360 nœuds de quatre processeurs octo-cœur Intel Nehalem-EX X7560 cadencés à 2.26 GHz, pour un total de 11520 cœurs. Chaque nœud dispose de 128 Go de mémoire RAM.

La matrice expérimentale d'ordre n utilisée sur Curie, que l'on nommera EXP, suit la formule de construction suivante :

- Sur les colonnes i paires, placer $-i/n$ sous la diagonale
- Sur les colonnes i impaires, placer des zéros sous la diagonale
- Sur les lignes j paires, placer j/n à droite de la diagonale
- Sur les lignes j impaires, placer des zéros à droite de la diagonale.
- À l'élément i de la diagonale, placer n + i * 0.5

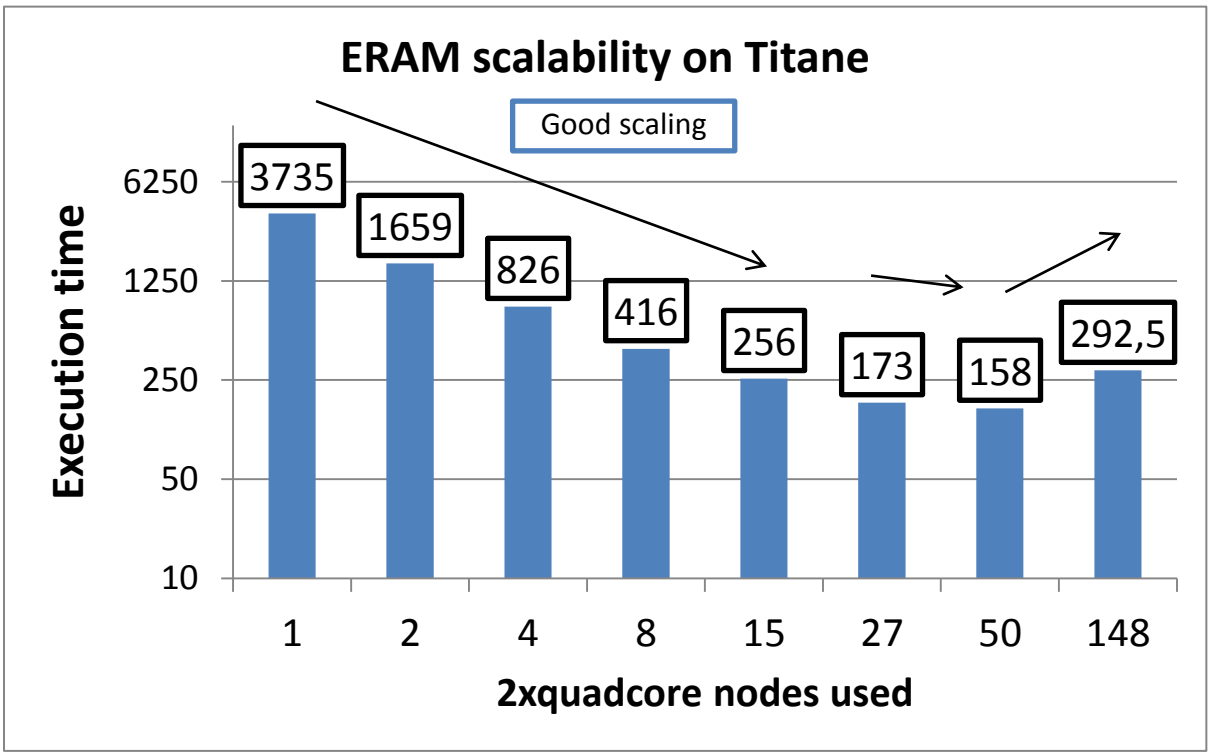


Figure 38. Scalabilité d'ERAM sur la machine Titane.

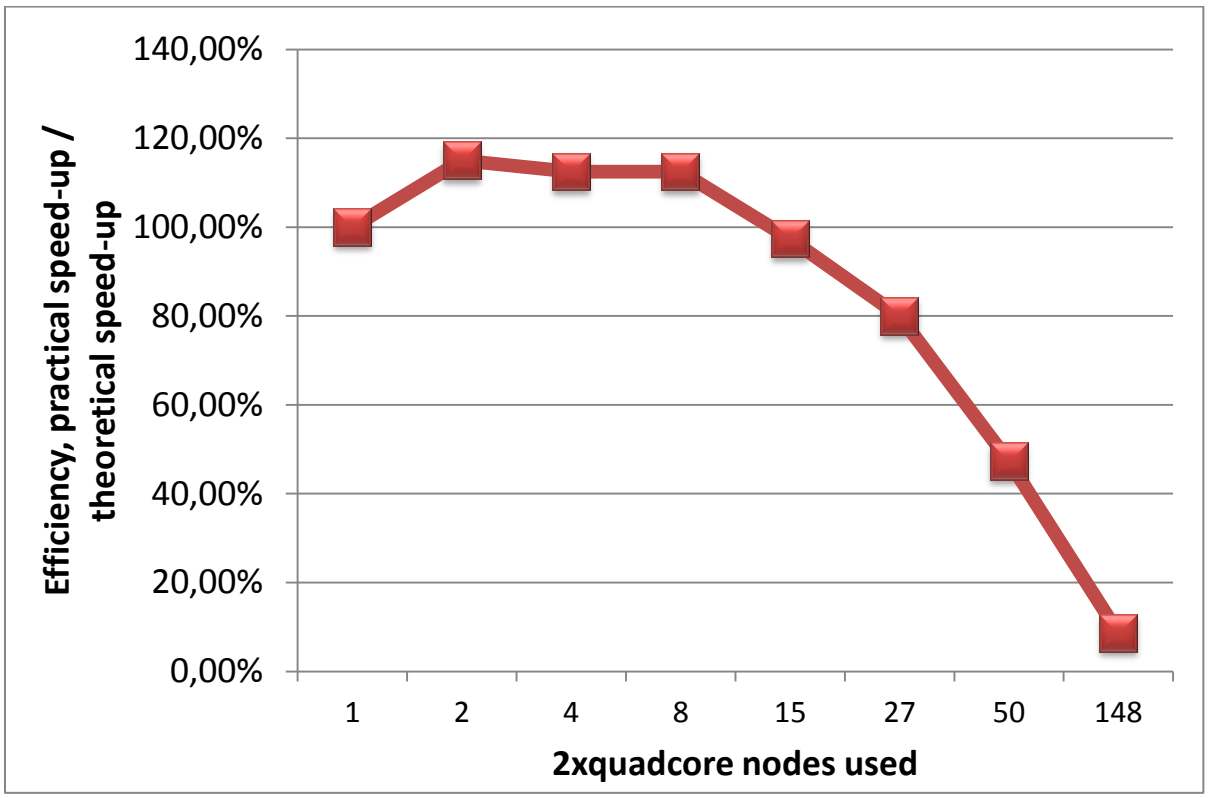


Figure 39. Efficacité d'ERAM sur la machine Titane.

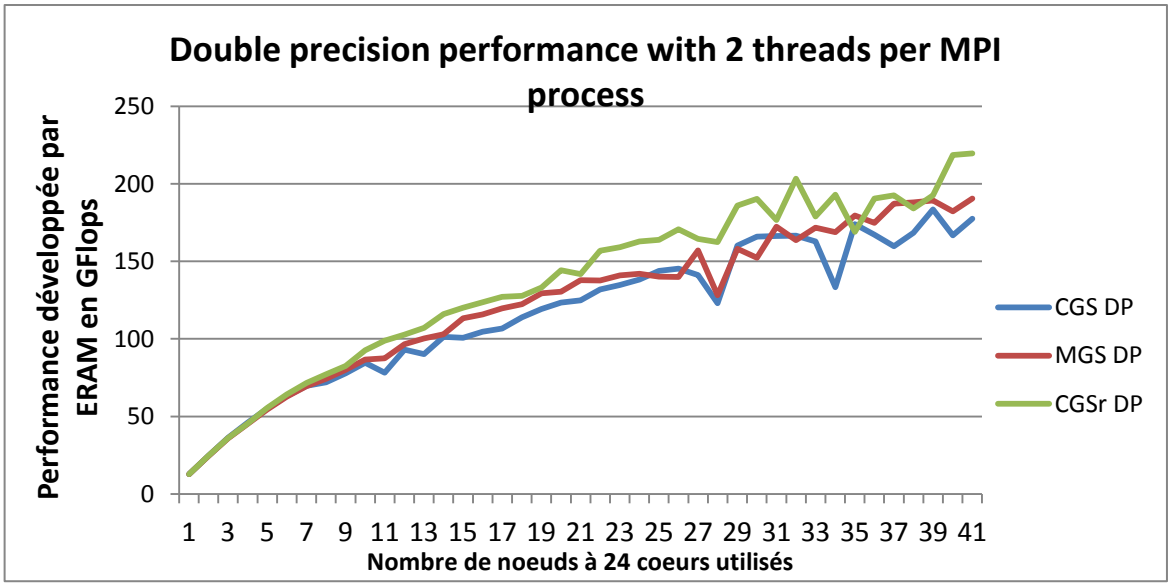


Figure 40. Performance de l'exécution d'ERAM sur la machine Hopper, de 24 à 984 cœurs.

Ainsi, la matrice EXP d'ordre 8 serait :

$$\begin{matrix} 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1/8 & 8{,}5 & 2/8 & 2/8 & 2/8 & 2/8 & 2/8 & 2/8 \\ -1/8 & 0 & 9 & 0 & 0 & 0 & 0 & 0 \\ -1/8 & 0 & -3/8 & 9{,}5 & 4/8 & 4/8 & 4/8 & 4/8 \\ -1/8 & 0 & -3/8 & 0 & 10 & 0 & 0 & 0 \\ -1/8 & 0 & -3/8 & 0 & -5/8 & 10{,}5 & 6/8 & 6/8 \\ -1/8 & 0 & -3/8 & 0 & -5/8 & 0 & 11 & 0 \\ -1/8 & 0 & -3/8 & 0 & -5/8 & 0 & -7/8 & 11{,}5 \end{matrix}$$

Cette matrice est à diagonale dominante, c'est-à-dire que la somme de ses éléments par ligne est inférieure à l'élément correspondant sur la diagonale. De plus, les valeurs propres des matrices EXP sont les éléments situés sur la diagonale. On voit qu'ils sont proches les uns des autres, et cette matrice offre une certaine complexité pour être résolue, à l'image de la matrice DingDong précédemment utilisée.

Nous avons utilisé une matrice EXP dense de taille 46080, que nous appellerons EXP46080. Nous cherchons à calculer les 4 valeurs propres de plus grande valeur absolue, avec une précision de 10^{-6}. Le Tableau 27 montre les temps d'exécution sur Curie, d'un nœud à 32 cœurs jusqu'au maximum qui nous était alloué dans le cadre du Preparatory Access : 5152 cœurs ou 161 nœuds.

Tableau 27. Temps d'exécution du solveur ERAM avec un sous-espace de taille 150 pour calculer 4 valeurs propres de plus grande valeur absolue de la matrice EXP46080 avec une précision de 1.0e-6 sur la machine Curie. Les speed-ups dépendent du nombre de nœuds car le problème est limité par la bande passante mémoire, et non la puissance de calcul.

Cœurs	31	65	129	257	513	1025	1921	3073	4609	5121
Nœuds	1	3	5	9	17	33	61	97	145	161
Temps	2730	722.3	405	325	227.4	61	62.14	52.4	63.2	34.7
Itérations	41	50	54	72	110	44	67	75	108	54
Temps par itération	66,6	14,45	7,5	4,51	2,06	1,39	0,93	0,7	0,59	0,64
Speed-up	1	4,6	8,9	14,8	32.3	47,9	71,6	95,1	112,9	104
Efficacité	1	1,53	1,78	1,64	1,9	1,45	1,17	0,98	0,78	0,65

Nous voyons que globalement, le temps d'exécution d'ERAM va en diminuant jusqu'à 5121 cœurs. Il faut cependant nuancer les performances affichées par une variabilité dans le nombre d'itérations pour chaque cas. À 5152 cœurs, 54 itérations sont nécessaires pour converger, alors qu'il en fallait 108 avec 4640 cœurs. Parallèlement, le temps a diminué d'un facteur 2, en toute logique. Pour mesurer la performance, nous regarderons donc le temps par itération. Dans ce cas-là, nous constatons la diminution de ce temps, de manières assez régulière de plus d'une minute à environ 0,6 secondes pour 4640 cœurs. On remarque d'ailleurs qu'avec 5152 cœurs, le temps par itération augmente. Nous avons atteint la limite à partir de laquelle l'augmentation du temps de communication devient prépondérante sur la diminution du temps de calcul.

4.3. Limitations de la parallélisation d'ERAM

La parallélisation d'ERAM implémentée peut utiliser les deux types de mémoires usuelles en calcul parallèle : mémoire partagée au sein d'un nœud et mémoire distribuée sur plusieurs nœuds. ERAM possède une performance intrinsèquement limitée par la bande passante mémoire à cause de ses noyaux de calculs BLAS 1 et 2 et des synchronisations globales nécessaires à son fonctionnement. A titre d'exemple, la figure 41 présente les dépendances entre les opérations BLAS1&2 de la méthode d'Arnoldi redémarrée.

Pour l'exemple présenté ci-dessous, nous disposons d'une certaine liberté dans la parallélisation des dots et axpys. Il serait ainsi possible de faire des séries de dots + axpys correspondant sur chaque processeur, avec ensuite une réduction du vecteur résultat nécessaire avant le calcul de la norme 2. Cela représente un transfert d'un vecteur de longueur égale au nombre de lignes de la matrice entre tous les processeurs, opération qui peut être assez coûteuse. De plus, cela implique un stockage différent de la matrice orthogonale utilisée lors de la projection : par paquets de colonnes complètes. L'autre solution est d'effectuer des produits scalaires locaux sur une portion des vecteurs, puis d'effectuer une réduction globale pour calculer des axpys locaux par la suite. La réduction implique moins de transferts que la solution précédente, et la matrice orthogonale est alors régulièrement répartie par bandes de ligne sur chaque processus. Une fois la norme 2 calculée, une réduction est nécessaire pour que tous les processus aient cette norme. Par la suite, l'opération scal est effectuée de manière répartie. Un transfert du résultat local du vecteur est alors nécessaire, pour que tous les processeurs puissent effectuer leur produit matrice vecteur à l'itération suivante. Au final, on comptabilise 3 communications globales par itération de la projection. Sachant qu'une itération s'exécute très rapidement, bien souvent sous la seconde ou la demi-seconde, alors on voit que les aléas des

communications globales peuvent fortement perturber la performance de la projection, et donc de la méthode globale.

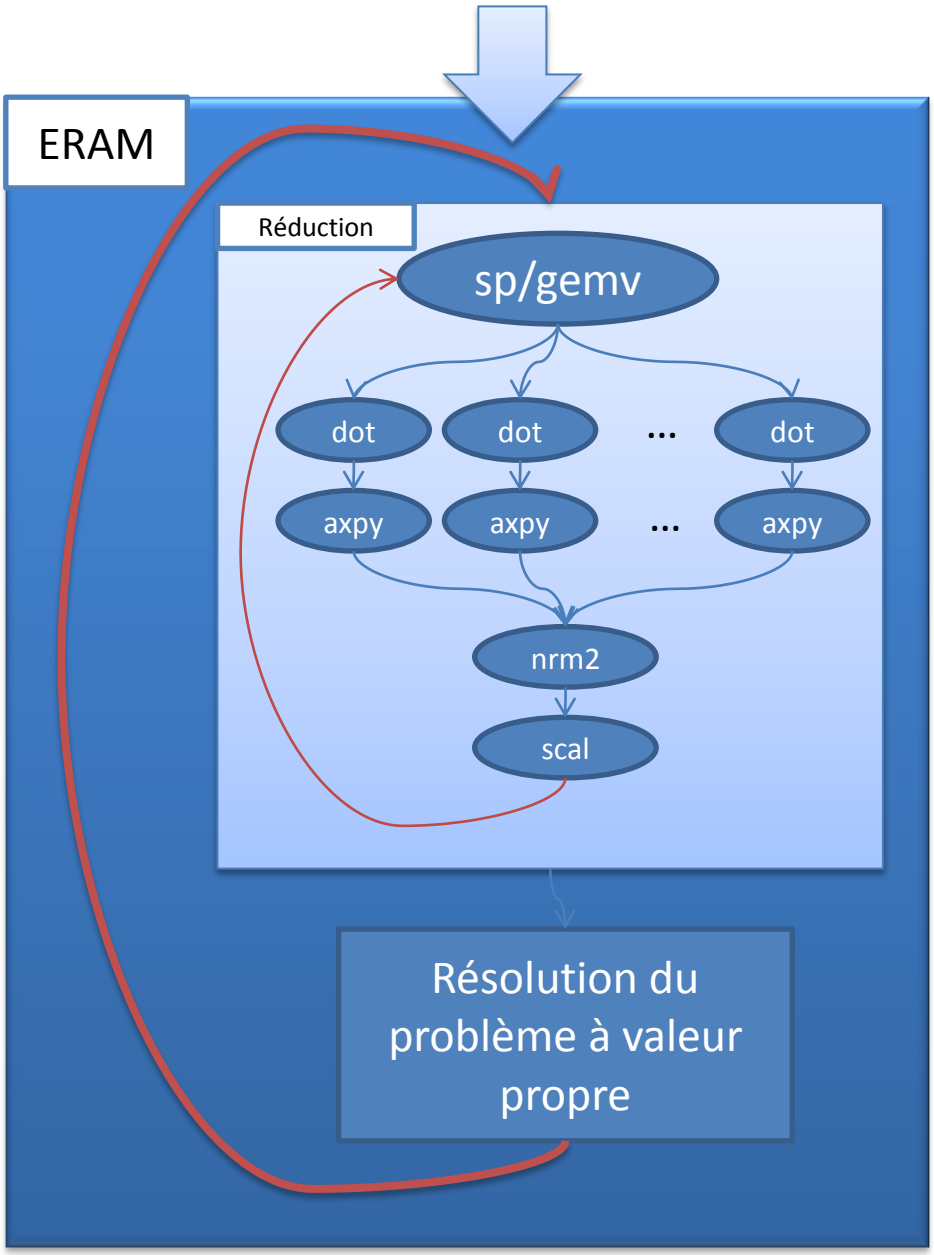


Figure 41. Dépendance des calculs pour ERAM utilisant l'orthogonalisation CGS.

Il existe plusieurs pistes pour gagner en scalabilité. On peut tenter d'augmenter la densité de calcul en réorganisant ces derniers, regroupant les opérations BLAS 1 pour faire du BLAS 2, et de même avec les BLAS 2 pour faire du BLAS 3. Cela permettrait de ne plus être limité par la bande passante mémoire, et de tirer plus partie de la puissance de calcul à disposition. De plus, il serait possible d'essayer de recouvrir les communications par des opérations de calcul. Par exemple, la densification des calculs se fait pour le calcul de la factorisation QR d'une matrice, permettant un gain de performance significatif autant sur architecture multicoeur que GPUs (64). Dans notre cas, cela semble assez difficile à cause des fortes dépendances de données, comme nous avons pu le voir dans la figure précédente. Dans (65), l'auteur propose également d'éviter les communications pour la méthode GMRES en modifiant légèrement l'algorithme de la projection de Krylov, avec une

accélération substantielle de 1.3x à 4x en mémoire partagée. Ce genre de technique serait envisageable pour ERAM. Plus proche d'Arnoldi, les auteurs utilisent dans (66) une technique de regroupement de la synchronisation de réorthogonalisation et du calcul de norme de l'orthogonalisation CGSr, permettant un gain de performance lorsqu'un grand nombre de nœuds est utilisé. Une approximation est utilisée pour décaler une communication globale et augmenter la scalabilité de l'orthogonalisation de Gram-Schmidt avec réorthogonalisation. Le gain atteint est intéressant en termes de performances, mais l'erreur introduite par les manipulations numériques n'est pas bornée.

Une autre solution est possible sans modification de l'algorithme d'ERAM, solution qui sera discutée dans le chapitre suivant.

4.4. Utilisation du framework pour les systèmes linéaires

Dans le cadre du stage de quatrième année de Laurine Decobert, étudiante à l'école Polytech'Lille, l'implémentation du Gradient Conjugué (22) avec préconditionnement polynomial a été réalisée. La maquette originale était constituée de plusieurs programmes écrits dans le langage C. Une version du programme était dédiée aux processeurs classiques, et une autre aux GPUs. Lors de ses travaux, le framework développé durant cette thèse a été utilisé pour remplacer les différentes versions du code assez rapidement, alors que l'étudiante n'avait pas encore abordé les notions de parallélisme en mémoire distribué ou sur accélérateurs.

Elle a pu étendre le framework avec de nouvelles notions telles que les matrices symmetric packed (67), qui ne contiennent qu'un peu plus de la moitié de la matrice, cette dernière étant symétrique. Des expérimentations ont pu avoir lieu sur des machines de la grille de calcul (68) Grid5000 (69; 70) ainsi que la machine Titane. On obtient une performance comparable à celle d'ERAM, avec une scalabilité assez bonne que l'on peut constater dans la figure suivante. Le solveur a été utilisé pour résoudre le système linéaire associé à une matrice de Lehmer d'ordre 21504. Les mêmes exécutions avec le même niveau de performance ont pu être effectuées sur des matrices de Hilbert, beaucoup plus difficiles à résoudre.

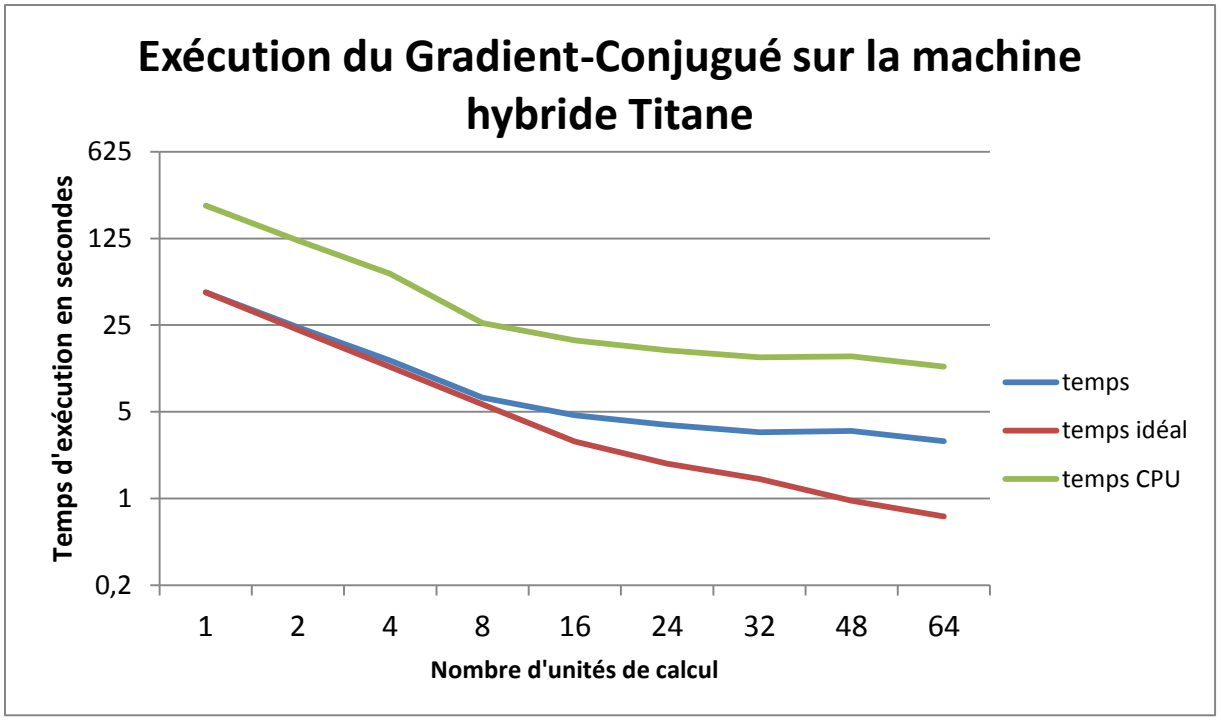


Figure 42. Résolution d'une matrice de Lehmer d'ordre 21504 par la méthode du Gradient-Conjugué avec préconditionnement polynomial. Utilisation de GPUs Tesla C1060 et de processeurs intel Nehalem. Selon l'axe des abscisses, une unité de calcul est un GPU ou un processeur Nehalem 4 cœurs.

Nous voyons ainsi la versatilité du framework d'accélération développé durant cette thèse, qui a pu être appréhendé par une étudiante apprenant le parallélisme. Elle a pu implémenter avec succès l'algorithme du Gradient Conjugué avec préconditionnement polynômial et obtenir des performances intéressantes, avec notamment un facteur d'accélération de l'ordre de 4 à 5 entre l'utilisation de N quadricoeurs ou N GPUs. Ces résultats seront présentés lors de la conférence PP2012 dans (71).

Conclusion

La parallélisation de la méthode d'Arnoldi redémarrée ERAM nous a permis de mettre en pratique le framework pour accélérateur dans le cadre de calculs de valeurs propres. Grâce à cette abstraction logicielle, nous pouvons utiliser des processeurs classiques ou des accélérateurs avec un seul et même algorithme. Les performances atteintes pour ERAM sur un accélérateur sont proches de la performance crête liée à la bande passante mémoire en calculs denses. Pour les calculs creux, des noyaux spécifiques inspirés de (57) que nous avons ajoutés au framework parallèle ont permis une bonne accélération lorsque le bon format de matrice est utilisé. Notre stratégie d'autotuning du produit matrice vecteur nous a permis de toujours prendre le meilleur format de matrice en fonction des données, pour une accélération par rapport au format de base d'un facteur 15x parfois.

Dans un environnement de type cluster, notre framework s'est bien comporté également, délivrant une performance satisfaisante pour les multicoeurs, et révélant le comportement spécifique de la scalabilité des GPUs. En effet, les GPUs offrent une efficacité de l'ordre de 50% lorsque la taille du problème est suffisante pour utiliser le parallélisme massif de ces accélérateurs.

Au final, la méthode ERAM elle-même possède des limitations dans sa scalabilité lorsque le problème et suffisamment large, et que l'on souhaite utiliser un grand nombre de nœuds. Pour un cas particulièrement pathologique pour un problème de valeur propre, nous avons pu diminuer le temps de calcul de 3800s sur 8 cœurs à 158 s avec 400 cœurs. Utiliser plus de cœurs ne nous apportait pas de performances, à cause de communications devenant trop coûteuses au regard du temps de calcul pur.

L'étude de méthodes hybrides dans le chapitre suivant permettra peut-être de gagner en scalabilité, tout en gagnant d'autres propriétés intéressantes.

Chapitre 5. Amélioration de la scalabilité de la méthode hybride MERAM

Introduction

Dans le chapitre précédent, nous discutions de la scalabilité de la méthode ERAM à cause de ses nombreuses limitations intrinsèques : un grand nombre de synchronisations et des opérations peu denses en calcul. Dans ce chapitre, nous explorerons l'utilisation d'une méthode hybride basée sur la collaboration de plusieurs solveurs permettant une accélération numérique de la méthode, en plus d'une meilleure scalabilité. Elle possède également des capacités de tolérance aux pannes, qui sont essentiels dans le but d'utiliser efficacement les futures machines Exascale (72).

5.1. Les méthodes hybrides et MERAM

Le terme méthodes hybrides a plusieurs significations possibles. Dans notre cas, il s'agit de l'utilisation de plusieurs méthodes ERAM concurrentes, qui collaborent ensemble dans le but de résoudre le même problème : le calcul de certaines valeurs propres d'une matrice. Cette méthode, développée dans (73; 74; 75) se nomme Multiple ERAM (MERAM). Les méthodes hybrides ont de nombreuses propriétés intrinsèques qui leur permettent d'améliorer des méthodes existantes sans modifications majeures. Dans le cadre d'ERAM, elles peuvent améliorer la scalabilité, et permettent naturellement une bonne résistance aux fautes matérielles.

La figure 43 montre un exemple d'exécution de MERAM avec quatre solveurs ERAM utilisant des sous-espaces de tailles variées. En effet, les ERAMs peuvent utiliser des paramètres d'entrées différents tels que la taille du sous-espace ou le vecteur initial. Les autres éléments pouvant différer entre les co-méthodes peuvent être par exemple l'orthogonalisation utilisée, la précision flottante, ou encore les valeurs propres calculées.

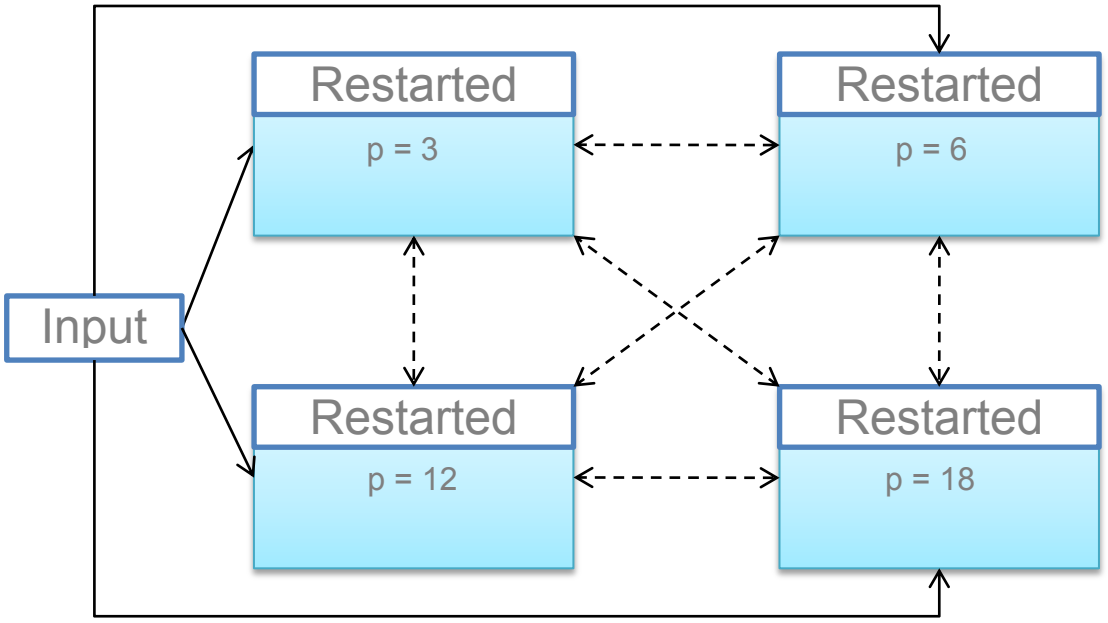


Figure 43. Exemple d'exécution de Multiple ERAM, une méthode hybride pour le calcul de valeurs propres d'une matrice.

Le fonctionnement de base de MERAM est assez simple : les ERAMs sont exécutés en parallèles, de manière asynchrone. Pour rappel, à chaque calcul de valeurs propres effectué par un ERAM, ce dernier va utiliser une combinaison linéaire de ses vecteurs propres pour calculer le vecteur de redémarrage pour l'itération suivante. MERAM propose que les ERAMs échangent leurs informations à la fin de chaque itération de manière asynchrone, et réutilise potentiellement l'information des autres ERAM pour calculer le vecteur de redémarrage.

Intuitivement, on peut comprendre l'intérêt de garder le meilleur de l'information entre tous les solveurs pour espérer avoir une convergence optimale. Si l'on prend l'exemple de deux ERAMs collaborant ensemble, avec l'un utilisant un sous-espace de taille 3 (ERAM(3)) et l'autre un sous-espace de taille 6 (ERAM(6)), alors la question de l'utilité pour ERAM(6) d'utiliser l'information d'ERAM(3) peut se poser. En effet, ERAM(6) est plus précis grâce à son plus grand sous-espace, et ne devrait donc pas bénéficier de l'exécution concurrente d'ERAM(3). La réalité est différente, car le comportement de convergence d'un ERAM n'est pas identifié à l'avance. Ainsi, il est possible qu'ERAM(3) converge plus rapidement dans un premier temps, et puisse faire bénéficier ERAM(6) d'une meilleure information sur le sous-espace contenant les valeurs propres. La figure 44 illustre ce comportement.

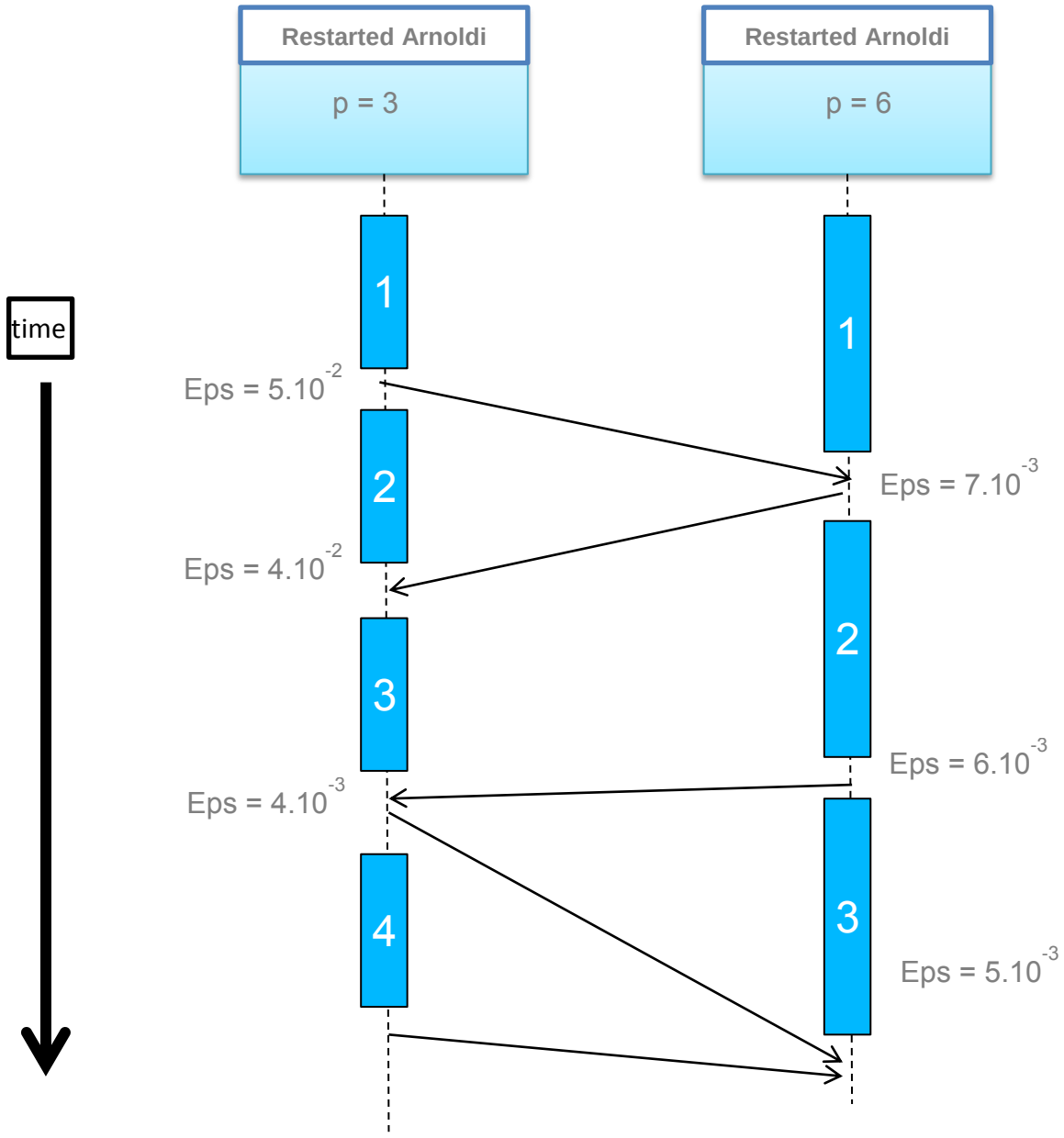


Figure 44. Illustration de l'évolution des échanges entre deux ERAM lors de l'exécution de MERAM. Les rectangles pleins représentent le temps d'une itération, et les flèches l'envoi d'information à un autre solveur. A la fin de l'itération 2, le solveur ERAM(3) utilise l'information de l'autre solveur, bénéficiant d'une accélération numérique. ERAM(6) bénéficie à son tour d'un meilleur résultat pour l'itération 4, non présentée sur le schéma.

Dans ce schéma, les deux solveurs ont bénéficié du résultat de l'autre pendant l'exécution, et ont globalement convergé plus rapidement que s'ils avaient été exécutés

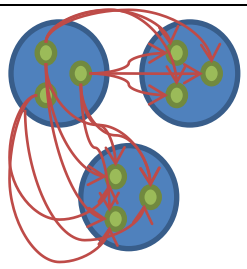
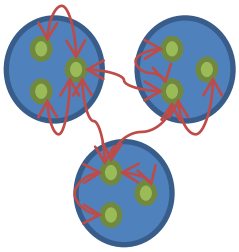
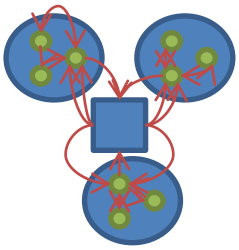
séparément avec le même nombre de processeurs. Nous illustrons ainsi à la fois l'accélération numérique que peut apporter MERAM, mais aussi l'augmentation de la scalabilité. En effet, chaque solveur conserve sa scalabilité propre, et le fait d'en exécuter plusieurs de manière concurrente permet d'utiliser plus de processeurs, sans perte de performance. De plus, si un ou plusieurs d'entre eux venaient à s'arrêter à cause d'une faute matérielle, alors il suffirait qu'au moins un solveur s'exécute pour que les calculs puissent continuer à converger. MERAM est ainsi intrinsèquement tolérant aux pannes.

5.2. Optimisation des communications

Pour implémenter MERAM, nous proposons une optimisation des communications entre les différents processus ERAM. En effet même si les communications sont exécutées de manière asynchrone, des délais liés à une implémentation non-optimale peuvent entraver l'accélération numérique globale de MERAM, et donc indirectement la scalabilité de l'algorithme. Avoir la possibilité d'exécuter une centaine d'ERAMs est intéressante si ces derniers peuvent effectivement échanger de l'information, et accélérer le calcul. Si l'implémentation les en empêche, alors il n'y a pas d'accélération numérique, et la performance de MERAM est égale à la performance du plus rapide des ERAMs à converger.

La version naïve des échanges d'informations serait que chaque processus de chacun des ERAMs échange de l'information avec tous les processus des autres ERAMs. En termes de nombre de messages, ce schéma est clairement non optimal, comme le montre le tableau 28. La version naïve voit son nombre de messages augmenter de manière quadratique avec le nombre de processus et de solveurs. Une première amélioration consiste à hiérarchiser les échanges en élisant un processus par solveur, qui sera autorisé à communiquer avec les autres solveurs. Cela permet de rompre le caractère quadratique des échanges, et d'éviter potentiellement des échanges de messages entre nœuds très distants d'une machine. Par contre, la symétrie des solveurs est rompue, avec un processus qui devra recevoir et transmettre l'information de l'extérieur à ses processus subordonnés. Cette solution offre donc une meilleure scalabilité des échanges à priori, avec une diminution d'un facteur équivalent au nombre de processeurs par solveur, mais pourrait être limitée si le nombre de solveurs devient grand : les échanges entre tous les processus élus de chaque solveur pourraient tout comme pour la solution précédente introduire des délais de communication. Nous proposons donc dans (76) d'introduire une entité dédiée à recevoir les données de tous les solveurs, à la manière d'une banque de données. La complexité des échanges de chaque solveur ERAM est ainsi diminuée, comme le montre le tableau 28.

Tableau 28. Optimisation des communications de MERAM. Les paramètres représentent le nombre de solveurs ERAM et p le nombre de processus alloués à chaque solveur respectivement.

MERAM version	0. Naive MERAM	1. Comm. hierachy	2. Data Collector
Comm. Scheme			
ERAM total messages MERAM total messages.	$2(n-1)p^2$ $2n(n-1)p^2$	$2(p+n-2)$ $2n(p+n-2)$	$2(p-1)+n$ $2n(p-1)+n^2$
For p=n=3 ERAM MERAM	36 108	8 24	7 21
For p=100, n=20 ERAM MERAM	38000 380000	236 2360	218 2180

Pour résumer l'évolution des complexités de communications présentée dans le tableau 28, nous avons tout d'abord diminué cette dernière de $O(2n^2p^2)$ pour la version naïve à $O(2np + 2n^2)$, permettant une meilleure scalabilité des échanges. Par la suite, avec l'introduction du collecteur de donnée, cette complexité a encore été diminuée à $O(2np + n^2)$.

Cette dernière solution autorise également l'utilisation de fonctionnalités avancées du passage de messages, telles que les « one-sided communications », des communications où n'est activement impliqué que le processus à l'origine de la communication. Ces fonctionnalités sont présentes au sein de la deuxième version de la norme MPI, appelée MPI-2. Pour utiliser ces fonctionnalités, chaque processus participant doit créer une fenêtre mémoire, qui agit un peu à la manière d'une mémoire partagée entre les processus. Chacun pourra alors faire une opération d'écriture (put) ou de lecture (get) dans la fenêtre d'un autre processus. Le maintien de la cohérence mémoire est à la charge du développeur, et ainsi si plusieurs processus effectuent des opérations put ou get au même endroit, le résultat n'est assuré que si des verrous (Lock) sont posés à l'endroit de l'opération mémoire.

Dans une de nos implémentations, nous avons donc utilisé ces fonctionnalités, en ne faisant participer activement que le collecteur. Ce dernier sondera périodiquement les processus élus des solveurs ERAMs par des Gets, et mettra à jour sa banque d'information si nécessaire. Il effectuera également la mise à jour des données de tous les processus élus n'étant plus à jour. De cette manière, les processus ERAM ne sont pas perturbés par les échanges MPI, hormis lorsqu'un verrou est posé sur la fenêtre mémoire par le contrôleur, et que le processus ERAM doit lire une donnée à cet endroit.

La figure 45 illustre de manière plus précise que le tableau 28 notre implémentation de la version optimisée de MERAM. Dans cet exemple, MERAM instancie trois solveurs ERAM utilisant des sous-espaces de taille 4, 7 et 9. Sur cette figure, nous voyons qu'il est possible pour chaque ERAM d'utiliser un nombre de processeurs différent par rapport aux autres.

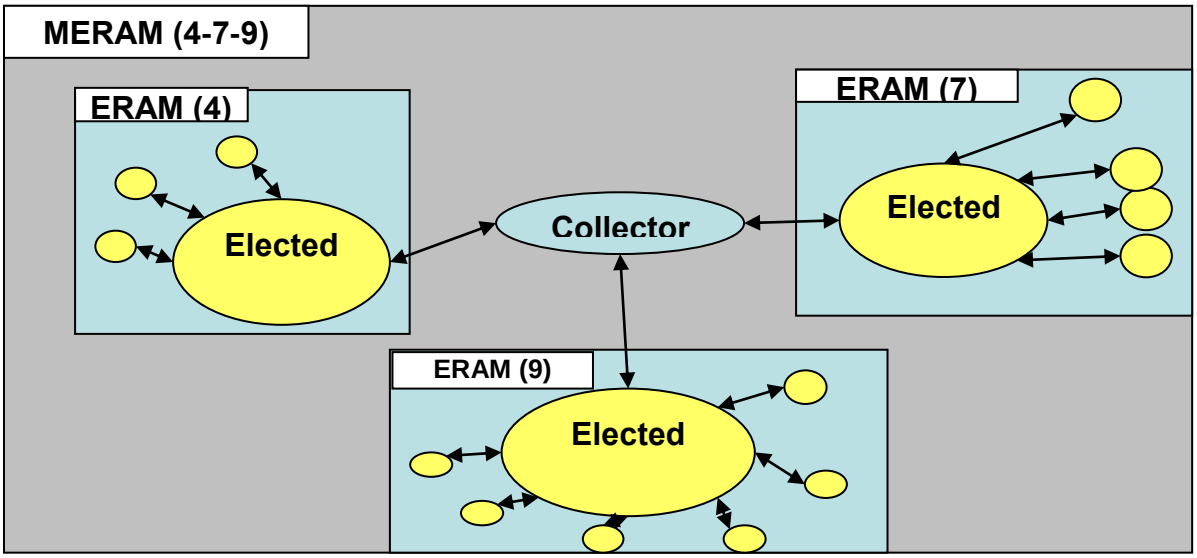


Figure 45. Instanciation de la version MERAM optimisée, utilisant trois solveurs ERAM avec un sous-espace de taille 4, 7 et 9.

Une autre implémentation a été développée également, utilisant simplement les communications asynchrones de MPI de type Isend et Irecv.

5.3. Performances à l'échelle d'un supercalculateur

Dans le chapitre précédent, nous avions rencontré une limite dans la scalabilité d'ERAM(9) sur le supercalculateur Titane avec une matrice de Dingdong, ainsi que pour le supercalculateur Curie avec une matrice EXP46080. Nous expérimentons maintenant le cas Dingdong avec MERAM, en utilisant 3 solveurs comme présenté dans la figure 45. Nous espérons qu'avec des sous-espaces de taille 4, 7 et 9, MERAM s'exécutera plus rapidement. Par rapport au cas du chapitre précédent, cette expérimentation revient à adjoindre deux solveurs ERAM avec des sous-espaces de taille 4 et 7 au solveur ERAM(9). La différence ici est que ces trois solveurs se partagent le même nombre de processeurs total qui était uniquement alloué au solveur ERAM(9) du chapitre précédent.

Nous voyons que le temps de résolution du problème est fortement réduit par rapport à ERAM, pour toutes les exécutions. Pour rappel, il était de 3800s sur un seul nœud avec un ERAM(9). Il nous permet d'atteindre un temps d'exécution réduit à 4,4 secondes, contre 158s au mieux avec un seul ERAM(9) précédemment. Le speed-up entre 1 nœud (8 cœurs) et 148 nœuds (1184 cœurs) est ainsi de 64,6x, soit une efficacité de 44%. Comparativement, l'utilisation d'un seul ERAM nous permettait une efficacité de 9% dans le chapitre précédent. Nous voyons sur la figure 47 l'efficacité atteinte par MERAM pour chaque nombre de nœud.

Globalement, nous voyons que cette efficacité est souvent bonne et proche de 100%, voir même superlinéaire. A 148 nœuds l'efficacité est plus faible que précédemment, pour une raison assez simple : environ 300 itérations sont nécessaires à ERAM pour converger. A chaque itération, 9 produits matrices vecteurs sont nécessaires, d'où un nombre total de 2700 produits matrices vecteurs. Ces opérations sont exécutées en 4 secondes, ce qui implique un temps de 0,00148s par produit. Pour rappel, dans la section 1.3.2, la latence d'un réseau local est estimée entre 10^6 et 10^8 ns soit 0,001s. Les opérations de calculs sont donc tellement accélérées, qu'elles atteignent le niveau de la latence du réseau. Nous sommes donc contraints par les variations de cette dernière, et nous pourrons difficilement accélérer encore plus la méthode.

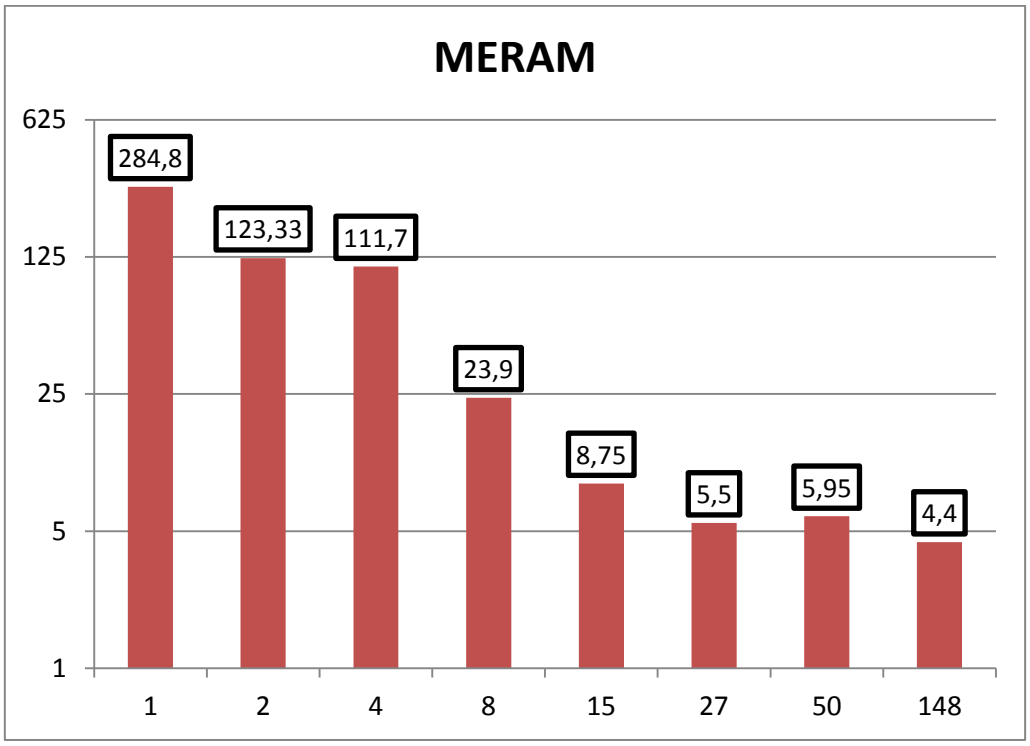


Figure 46. Performances de MERAM utilisant trois solveurs ERAM avec des sous-espaces de taile 4, 7 et 9. Le cas traité est le même que dans le chapitre précédent, lors de l'étude de la scalabilité d'ERAM. L'axe des abscisses indique le nombre de nœuds de 2*quadcores de la machine Titane. Sur les ordonnées, on peut lire le temps en secondes.

Une autre baisse d'efficacité a lieu à 4 nœuds. L'explication de cette baisse se trouve dans la stratégie de redémarrage de MERAM. Chaque solveur cherche invariablement à prendre le meilleur résultat dont il dispose. C'est un choix pertinent, mais le phénomène suivant se produit en général lorsqu'il y a changement de vecteur de redémarrage en incluant les données d'un autre solveur : l'itération suivante s'accompagne d'une perte de précision, corrigée en général à l'itération suivante du solveur par un gain substantiel de précision. Cependant, il est possible qu'avant le démarrage de l'itération suivante, un résultat meilleur soit disponible à partir d'un autre solveur. Ce résultat est alors utilisé pour redémarrer le solveur, avec potentiellement une perte de précision du même niveau que précédemment. Ce processus peut se produire si par exemple un solveur converge lentement mais avec un meilleur résultat que les autres solveurs à un instant donné. Les autres essaieront d'utiliser le résultat de ce solveur, avec potentiellement le comportement décrit précédemment. Il s'agit en quelque sorte d'un deadlock numérique. Cela implique plus d'itérations car la convergence de MERAM n'est alors plus assurée que par le solveur qui nourrit les autres, et les déstabilise. Le temps de calcul est alors plus long, et l'efficacité finale moindre. Pour corriger ce défaut, nous proposons d'optimiser la stratégie de redémarrage.

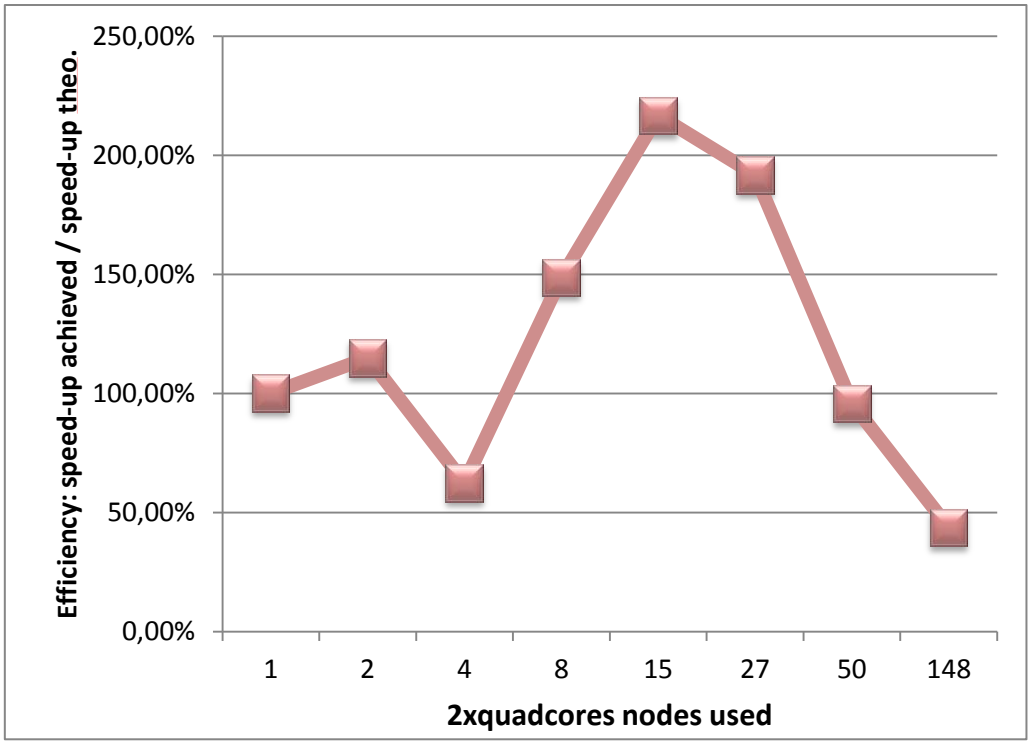


Figure 47. Efficacité de MERAM selon le nombre de nœuds de Titane utilisés.

5.4. Autotuning hybride : adaptation de la stratégie de redémarrage

La stratégie de redémarrage peut donc s'avérer trop sensible, et entrainer un ralentissement de la convergence de MERAM globalement lorsqu'un solveur semble avoir le meilleur résultat mais offre une convergence plus lente. Les autres solveurs se mettent alors au pas de ce dernier. On peut même imaginer à l'extrême qu'un des solveurs ait atteint le nombre maximum d'itérations, et arrête de calculer. Son résultat est alors figé, mais peut sembler meilleur que celui des autres solveurs, qui essaieront alors de l'utiliser pour leur redémarrage, et qui ne convergeront jamais.

Nous proposons dans (76) d'améliorer cette stratégie en affinant les critères de redémarrage en utilisant une heuristique. A la fin de chaque itération, nous essayons d'avoir le meilleur résultat provenant des autres solveurs, que nous appellerons best_result. Le résultat de l'itération précédente est previous_result, et celui de l'itération courante est current_result. Nous calculons le taux de convergence du solveur avec la formule :

Taux_convergence = current_result / previous_result,

et le comparons au taux de convergence que nous aurions eu en utilisant le meilleur résultat à la place de celui obtenu à l'itération actuelle :

Taux_hypothetique = best_result / previous_result.

Par la suite, nous étudions le ratio speed-up = Taux_hypothetique / Taux_convergence, et si ce dernier est suffisamment grand, alors le solveur a un intérêt à utiliser le meilleur résultat en provenance d'un autre solveur. Avec un speed-up suffisamment grand, le solveur est assuré de bénéficier d'une amélioration importante de la qualité de sa solution, ce qui permet d'éviter les situations de deadlocks numériques.

Cette technique a permis de régulariser la courbe précédente, et d'obtenir une efficacité proche de 100% dans le cas à quatre nœuds. Des expérimentations sur la matrice EXP46080 ont ensuite été menées avec cette stratégie de redémarrage activée, pour obtenir les résultats suivants présentés dans la figure 48. Étant donné que nous n'avons pas accès à la totalité des 10000 processeurs de Curie en une seule exécution, nous avons pris

comme base l'exécution à 1056 cœurs. On peut imaginer qu'elle soit la limite au-delà de laquelle les communications deviennent trop coûteuses pour que l'on ait un gain de temps à utiliser plus de 1056 cœurs. Nous choisissons alors de faire appel à MERAM avec trois ou quatre solveurs ERAM utilisant chacun 1024 cœurs. Par rapport à une exécution d'un seul ERAM à 1056 cœurs, le temps est diminué suivant les accélérations du tableau 29.

On voit qu'un MERAM utilisant 3073 cœurs offre une meilleur performance qu'un ERAM utilisant le même nombre de cœurs, avec une amélioration d'un facteur 1,5x. Ce même MERAM offre également un niveau de performances presque équivalent à un ERAM utilisant 5121 cœurs. Lorsque l'on ajoute un solveur supplémentaire à MERAM, utilisant donc 1024 cœurs de plus pour un total de 4096, alors la performance s'améliore légèrement, offrant un gain de 18% par rapport à l'ERAM utilisant 5121 cœurs.

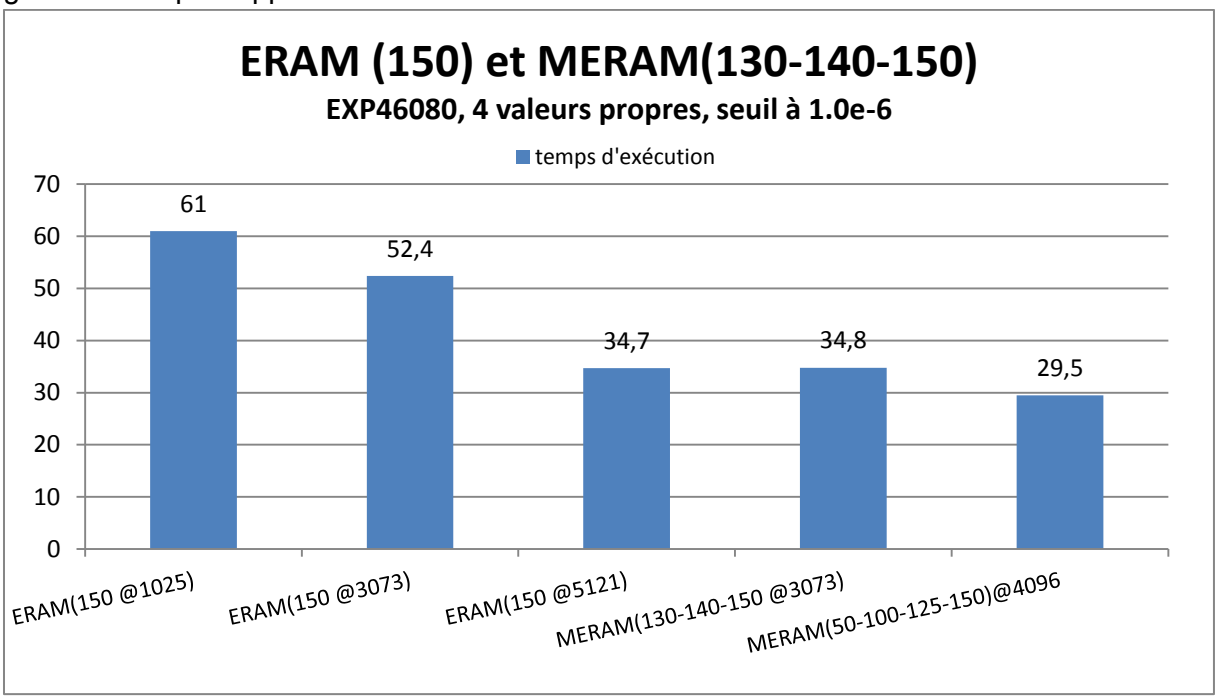


Figure 48. Exécutions d'ERAM et MERAM sur la machine Curie pour calculer les 4 valeurs propres de plus grande valeur absolue de la matrice EXP46080 avec un seuil de 1.0e-6.

Tableau 29. Accélération apportée par le parallélisme "brut" d'ERAM ou l'utilisation de co-méthodes via MERAM.

	ERAM (150@1025)	ERAM (150@3073)	ERAM (150@5121)	MERAM (130-140-150 @3073)	MERAM (50-100-25-50 @4096)
itérations	44	75	54	14	19
Temps d'exécution (s)	61	52.4	34.7	34.8	29.5
Speed-up	1	1,16	1,76	1,75	2,07

Nous n'avons pas pu tester l'exécution à 5121 cœurs d'un MERAM pour des contraintes de temps, mais les résultats actuels offrent des perspectives intéressantes pour augmenter la performance globale d'un algorithme ERAM plafonnant à cause des communications. Nous avons vu qu'utiliser plusieurs ERAMs permet un gain de performance, mais ce dernier n'est pas équivalent à celui qui aurait été obtenu si un seul ERAM avait pu maintenir sa scalabilité avec un nombre de processeurs équivalent. Cependant, on constate une amélioration grâce à MERAM, qui n'aurait pas été possible par utilisation de parallélisme « brut ». MERAM

introduit également d'autres propriétés, telles que la tolérance aux pannes, que nous verrons dans la partie suivante.

5.5. Tolérance aux pannes

La tolérance aux pannes est une capacité intrinsèque de la méthode MERAM, et nous proposons de la tester dans l'hypothèse de l'utilisation d'une machine très large qui subirait des pannes matérielles. Plusieurs séries de tests ont été effectuées : l'arrêt à un nombre d'itérations donné d'un des solveurs, qui représente une panne aléatoire sur un ou plusieurs des nœuds d'un seul solveur, et une étude statistique de la résistance de MERAM à une dégradation massive de la stabilité de la machine.

5.5.1. Stabilité lors de la perte d'un solveur

Nous choisissons une matrice, AF23560 en provenance du site MatrixMarket, et nous fixons différentes configurations d'exécutions de MERAM, par rapport à un ERAM standard. Cet ERAM standard utilise un sous-espace de taille 10 et converge en 1001 itérations à une précision de 10^{-15}. Les deux premières configurations testées utilisent respectivement des ERAM de sous-espaces 4 et 10, et 3, 4, 7 et 10. Leurs convergences s'effectuent respectivement en 140 et 100 itérations. Le tableau 30 résume ces configurations.

Tableau 30. Différentes configurations pour tester la résilience de MERAM, en comparant le résultat à un ERAM standard.

Solveur	Sous-espace	Itérations
ERAM(10)	10	1001
MERAM(4-10)	4 - 10	140
MERAM(3-4-7-10)	3-4-7-10	100

La figure 49 et la figure 50 montrent le résultat de quelques expérimentations. Pour MERAM(4-10), le solveur 0 est ERAM(4) et le solveur 1 ERAM(10). Il en va de même avec MERAM(3-4-7-10), où les solveurs 0, 1, 2, 3 sont respectivement les ERAM 3, 4, 7 et 10.

Sur ces deux graphes, nous voyons premièrement que MERAM est toujours capable de fournir un résultat malgré la perte d'un solveur. Cependant, dans le graphe concernant MERAM(4-10), les tests où le solveur ERAM(10) est arrêté et qu'il ne reste plus que le solveur ERAM(4) ne convergent pas ou plus vers la précision demandée en un nombre d'itérations inférieur à celui d'un ERAM(10) seul. Ce comportement est normal car en réalité le solveur ERAM(4) est incapable d'atteindre une précision de 10^{-15} à cause d'une taille de sous-espace trop petite. Par contre, lorsqu'il ne reste plus que le solveur ERAM(10), nous voyons que ce dernier converge vers un résultat satisfaisant, en ayant bénéficié de l'accélération d'ERAM(4).

Dans le deuxième graphe, l'arrêt d'un des solveurs n'a pas une incidence majeure sur la convergence finale de MERAM(3-4-7-10). Le scénario le pire est lorsque le solveur de plus petit sous-espace ERAM(4) est arrêté rapidement, ne faisant plus bénéficier les autres solveurs de ses calculs. Alors le nombre d'itération nécessaires augmente de 100 à 202, diminuant la performance finale de MERAM, qui reste cependant meilleure que celle d'un seul ERAM(10) utilisant tous les processeurs pour son calcul : 1001 itérations. Dans les autres cas, MERAM(3-4-7-10) converge en 96 à 120 itérations contre 100 itérations pour MERAM(3-4-7-10). On voit que lorsque le solveur 2 est coupé artificiellement prématurément, MERAM converge légèrement plus vite. Il doit donc encore y avoir des possibilités d'améliorer la stratégie de redémarrage.

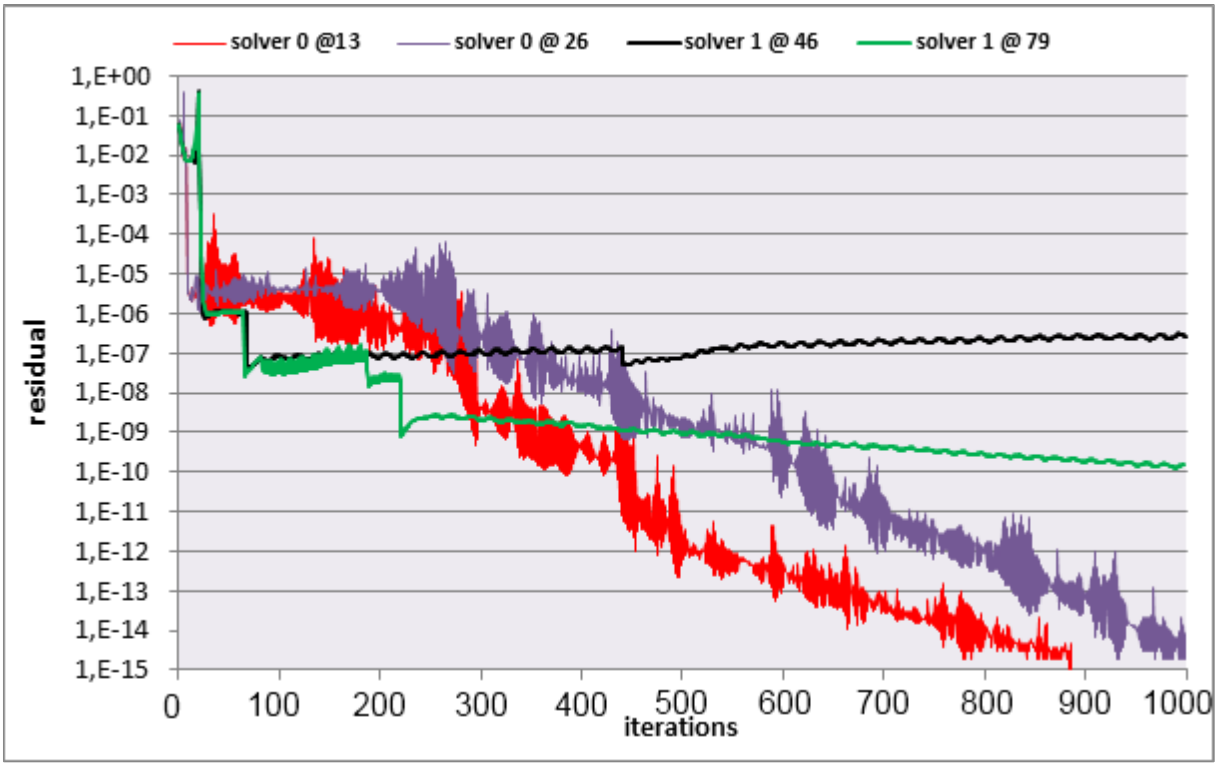


Figure 49. Évolution de la convergence lorsque MERAM(4-10) perd un de ses solveurs. La légende indique quel solveur a été arrêté artificiellement à quelle itération.

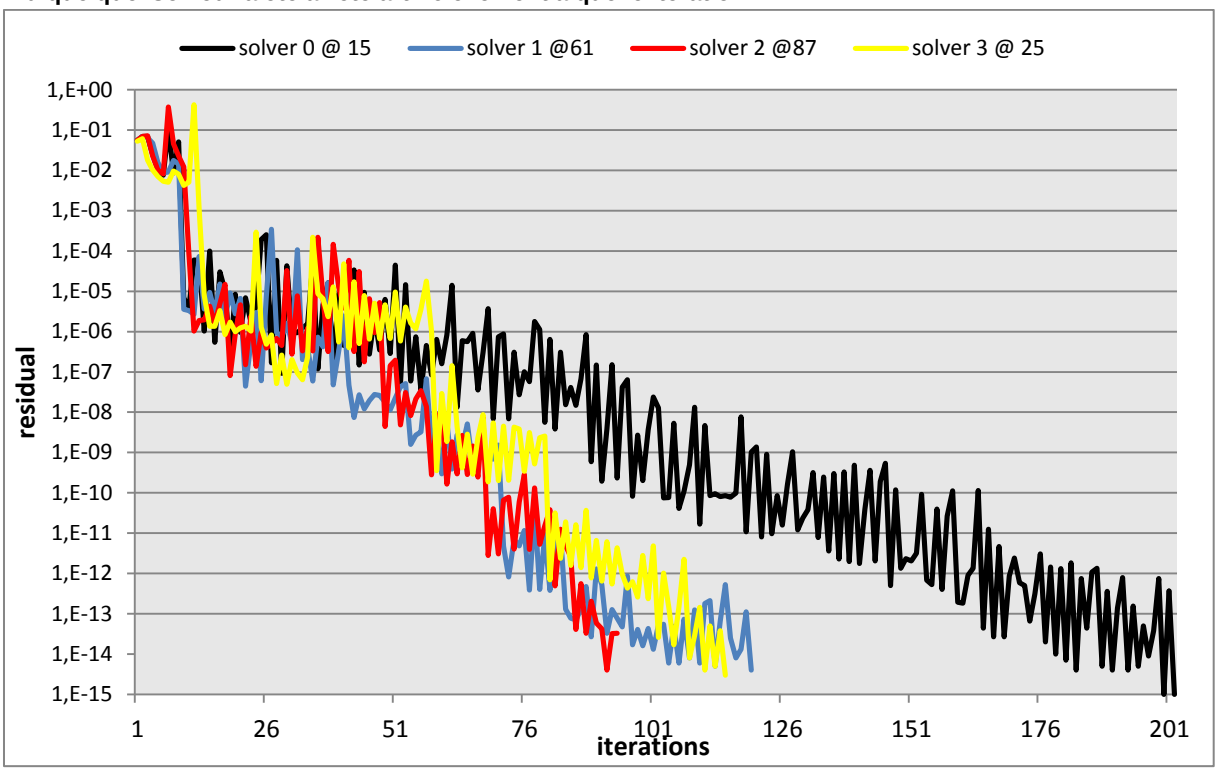


Figure 50. Évolution de la convergence lorsque MERAM(3-4-7-10) perd un de ses solveurs. La légende indique quel solveur a été arrêté artificiellement, à quelle itération.

5.5.2. Statistiques lors de la perte progressive de plusieurs solveurs

Nous avons repris les tests effectués sur la machine Curie avec la matrice EXP46080 comme base d'une étude de différents scénarii. Étant limité à environ 5000 cœurs, nous avons fixé un ERAM utilisant 4097 cœurs comme base de résultats. Nous avons ensuite choisi d'exécuter un MERAM avec 4 solveurs ERAM, dont les tailles de sous-espaces sont 50, 100, 125, 150. Les tests effectués sont plutôt pessimistes : 2 cas où 2 solveurs sont arrêtés rapidement, et deux cas où trois solveurs sont arrêtés. Les résultats sont fournis dans le tableau suivant, où un astérisque symbolise les solveurs subissant une panne.

Dans chaque case, le nombre d'itérations est donné pour chaque solveur, avec le temps d'exécution global dans la colonne de droite.

solveur	**50**	**100**	**125**	**150**	**Temps (s)**
Cas 0	57	29	23	19	29,5
Cas 1	53	**10***	23	**8***	29,96
Cas 2	**13***	62	**3***	44	52,7
Cas 3	**13***	**10***	**3***	44	59,6
Cas 4	900	**10***	**3***	**8***	365,86

Dans les cas 1 à 3, MERAM converge dans un temps égal à celui sans défaillance ou en mettant deux fois plus de temps. Le seul cas qui n'apporte pas une convergence au seuil de 10^{-6} est le cas 4. Cependant, le résultat final obtenu est d'une précision de 1.1×10^{-6}. Si l'on compare les cas 1 à 3 à l'exécution d'un seul ERAM avec un sous-espace de taille 150 dont le temps d'exécution est 61 secondes dans le tableau 27, alors on voit que l'on est au pire comparable au cas où un seul solveur est utilisé. Le cas 4, plus pathologique, n'est pas satisfaisant.

Pour avoir une statistique de plus grande échelle, nous avons pris comme base un solveur utilisant 256 cœurs, et nous avons instancié un MERAM utilisant 10 solveurs ERAM de 256 cœurs chacun avec des sous-espaces de 75, 100, 110, 120, 125, 130, 135, 140, 145, 150. Les scenarii envisagés sont les suivants :

- Cas 1 : les 7 solveurs de sous-espaces inférieurs à 140 s'arrêtent
- Cas 2 : les 3 solveurs de sous-espace supérieurs à 135 s'arrêtent
- Cas 3 : les solveurs de rang pairs s'arrêtent
- Cas 4 : les solveurs de rang impairs s'arrêtent
- Cas 5 : le solveur de rang le plus grand s'arrête

Les résultats sont présentés dans le tableau suivant, avec les solveurs subissant une panne marqué d'un astérisque.

solveur	**75**	**100**	**110**	**120**	**125**	**130**	**135**	**140**	**145**	**150**	**Tps**
Cas 0	102	73	66	58	58	55	40	47	32	35	208
Cas 1	**4***	**10***	**6***	**8***	**2***	**15***	**20***	77	49	51	274
Cas 2	77	56	50	47	41	41	28	**5***	**8***	**7***	154
Cas 3	**5***	81	**8***	59	**15***	60	**20***	49	**11***	29	220
Cas 4	119	**10***	80	**8***	69	**15***	44	**5***	43	**7***	240
Cas 5	-	-	-	-	-	-	-	-	-	**7***	215

Contrairement à l'expérimentation précédente, tous les cas atteignent le critère de convergence. Les temps sont globalement proches ou supérieurs, augmentant de 3 à 32% par rapport à un cas non perturbé par des pannes. Par contre le cas 2 est intéressant car il offre un temps meilleur de 26% et un nombre d'itérations plus faible que dans le cas où il n'y a pas de panne. Cela indique que certains solveurs n'apportent peut-être pas toujours une contribution utile aux calculs, et peuvent perturber la convergence globale de MERAM. Il

existe encore des voies d'amélioration à MERAM au niveau de la gestion des co-méthodes, qui mériteraient d'être investiguées dans des travaux ultérieures à cette thèse.

Cependant, lorsque l'on compare au temps d'exécution d'un seul solveur ERAM utilisant un sous-espace de taille 150 avec 257 processeurs, le temps constaté est de 325 secondes (tableau 27). Tous les MERAMs proposent un temps d'exécution inférieur, et l'accélération liée à la méthode numérique est de 1,18 à 2,11, alors que l'on utilise 10 fois plus de ressources. Par contre, si l'on compare à un seul ERAM utilisant 2561 processeurs, le temps global est de 57 secondes, soit une accélération de 5,7 par rapport à un ERAM avec 257 processeurs, et même une accélération de 2,7 à 4,8 par rapport au MERAM à 10 solveurs. Cela illustre un commentaire précédent indiquant que MERAM ne parvient pas toujours à être meilleur que la scalabilité d'un ERAM, lorsqu'elle est satisfaisante. MERAM apportera par contre une amélioration si la scalabilité d'un seul ERAM devient moins optimale.

5.6. Accélération GPUs

Dans l'état actuel des travaux, l'accélération GPU n'a pas pu être testée avec MERAM. Cependant, cette dernière étant orthogonale au parallélisme MERAM, les temps devraient être réduits de la même manière qu'avec un ERAM standard : la scalabilité serait moins bonne pour chaque ERAM à cause des communications inter-GPUs, et avec un faible nombre de GPUs, un facteur de l'ordre de 5 serait observé pour le même nombre de GPUs.

Cependant, une des propriétés de MERAM, l'asynchronisme des communications, pourrait permettre une utilisation novatrice de cette méthode hybride. Sur chaque nœud de calcul hybride, plusieurs multicoeurs sont présents, ainsi que plusieurs GPUs. Prenons l'exemple de la machine Titane, qui possède deux processeurs quadricoeurs Intel par nœud et deux GPUs Nvidia Tesla C1060. Plusieurs utilisations intéressantes de ce nœud seraient possible :

1. Exécuter un ERAM sur les 8 cœurs CPUs, et un ERAM sur les 2 GPUs.
2. Exécuter un ERAM sur les 8 cœurs CPUs, et un ERAM sur chaque GPU.

Dans le cas 1, la puissance développée par les deux GPUs est supérieure à celle des 8 cœurs CPUs. Il serait donc possible d'exécuter un ERAM de sous-espace plus grand sur GPUs, ce qui apporterait à priori une meilleure information à l'ERAM présents sur CPUs. Le temps d'exécution serait ainsi égal au pire à celui convergeant le plus vite entre les deux solveurs, et au mieux on pourrait combiner la puissance de calcul des deux méthodes à une accélération numérique. Le cas 1 permet ainsi d'utiliser la puissance des GPUs et CPUs pour obtenir le meilleur résultat possible. De manière générale, les problèmes limités par la bande passante mémoire n'utilisent que très rarement à la fois les CPUs et les GPUs pour du calcul, à cause du lien PCI-Express plutôt lent et de sa latence élevée. MERAM permettrait de potentiellement utiliser les deux à la fois et d'offrir la meilleure performance entre les deux matériels dans le pire des cas, et une accélération supérieure à ce que peuvent fournir les puissances de calcul cumulées des deux matériels au mieux.

Le cas 2 est une variante du cas 1 partageant une bonne partie des propriétés en termes de performances, mais reposant plus sur l'accélération numérique que peut apporter un MERAM. Ici aucun matériel n'est limité par les communications inter-GPUs, mais les 2 GPUs ne bénéficient pas directement de leur puissance de calcul cumulée.

Si l'on étend les cas 1 et 2 à un supercalculateur hybride, alors les possibilités sont assez grandes, et peuvent certainement apporter des résultats probants.

Conclusion

Dans ce chapitre, nous avons exploré une possibilité pour améliorer la scalabilité de la méthode d'Arnoldi redémarrée ERAM : la méthode hybride MERAM.

Nous avons proposé une optimisation du schéma de communications de cette méthode connue tout en utilisant notre framework parallèle, et appliqué MERAM à une plus grande échelle qu'ERAM. Une accélération numérique a été constatée à nombre de nœuds constant, tout comme une augmentation sensible du nombre de nœuds utilisables pour l'exécution de MERAM. Quelques deadlocks numériques liés à la convergence plus lente d'un sous-solveur de MERAM ont été corrigés via l'autotuning de la stratégie de redémarrage. Cela a globalement permis une stabilisation des performances, et une meilleure adaptabilité de l'algorithme à l'ajout de solveurs.

La tolérance aux pannes de MERAM a pu être étudiée et prouvée, via la simulation de l'arrêt de certains solveurs. Nous constatons que sauf cas très critiques, MERAM est capable de converger là où un ERAM normal aurait demandé un redémarrage complet de la méthode avec la perte de temps associée. Comme note finale, tout ERAM stoppé peut redémarrer simplement en rejoignant les calculs, et s'alimentant sur le collecteur de MERAM, qui contient les meilleurs résultats à un instant donné.

D'un point de vue hybridation plus poussée, il serait possible d'exécuter un ERAM sur GPUs ou CPUs uniquement, et de mêler ces deux exécutions au sein d'un seul MERAM. La réflexion apportée vers la fin de ce chapitre laisse à penser que des schémas de calculs et des performances encore supérieures à ce qui a pu être présenté durant cette thèse sont atteignables et très appropriées à des supercalculateurs hybrides.

Chapitre 6. Application aux solveurs neutroniques

Introduction

Dans le domaine de la simulation de cœurs de réacteurs, il existe différentes problématiques qui contribuent toutes à cette simulation telles que la gestion de la thermodynamique ou des flux de neutrons pour l'énergie. Ces problématiques sont adressées par différents solveurs itératifs qui calculent l'état du cœur de réacteur de manière couplée ou non. Ces solveurs peuvent demander une puissance de calcul très élevée dans le cadre de calculs de cœurs complets très fins. L'intégration de la puissance de calcul des GPUs a donc un intérêt dans ce cadre : augmenter la densité de calculs d'une machine, pour atteindre ou un temps de calcul réduit, ou une précision plus grande. Une autre thématique chère à la simulation numérique est la simulation temps réel de l'état d'un réacteur. Pour atteindre ce but, les calculs sont simplifiés et offrent une précision moindre, mais dans une enveloppe de temps contrainte et réduite. Ainsi, l'utilisation de GPUs pourrait permettre d'atteindre une précision meilleure dans le cadre de calculs temps réels de cœurs de réacteur. Dans ce chapitre, nous étudierons l'intégration effective du framework d'accélération GPU au sein du solveur neutronique SPn MINOS, et également l'accélération OpenMP apportée au sein du solveur neutronique Sn MINARET.

6.1. Présentation du code APOLLO3®

Durant les dix dernières années, il y a eu un intérêt grandissant dans l'industrie nucléaire pour des codes de simulations nucléaires de plus en plus performants. Étant donné la diversité des concepts de réacteurs (PWR, BWR, GFR, SFR, SCWR, …) et la recherche de la maitrise au plus près des incertitudes permettant d'améliorer la sûreté et les marges de fonctionnement, il y a un intérêt certain pour le développement d'un nouveau code multifilière couvrant toutes les échelles de modélisation.

De plus, l'évolution de la technologie de calcul à disposition des scientifiques pour effectuer ces simulations a suivi différentes lignes en termes de développement matériel. Ainsi, il est possible de bénéficier de l'amélioration constante de la performance de calcul des machines parallèles. Cela donne de nouvelles directions pour le développement des codes futurs ainsi que des extensions des méthodes numériques pour effectuer des résolutions déterministes à grande échelle.

Ainsi, le CEA, avec le support d'EDF et d'AREVA, renouvelle ses codes principaux dédiés à ces problématiques : APOLLO2, CRONOS2 et ERANOS2 (77; 78; 79; 80; 81; 82) pour les codes déterministes et TRIPOLI4 (83) pour les codes Monte Carlo. Le code d'évolution MENDEL (successeur de DARWIN (84)) fait partie de ce renouvellement, tout comme le code de traitement des données nucléaires GALLILEE (85), et finalement CONRAD (86), l'outil d'analyse de réaction nucléaire pour créer des évaluations.

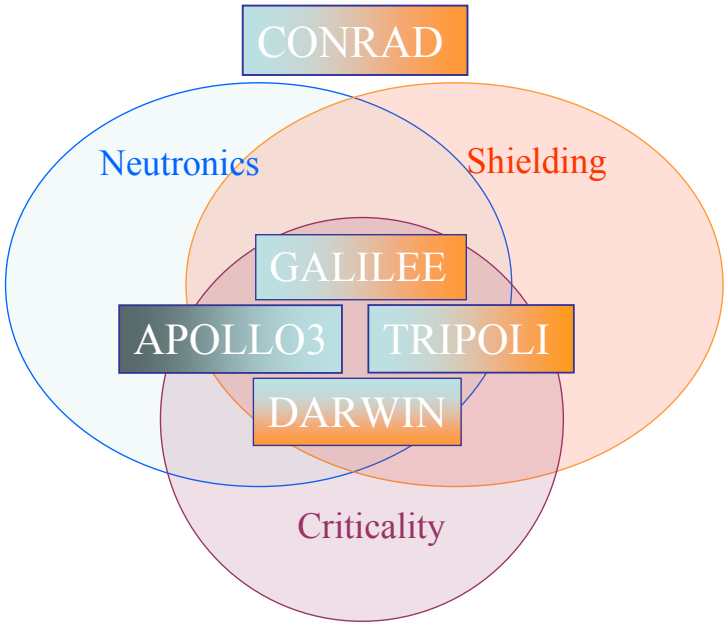


Figure 51. Programme de développement global du CEA pour modéliser finement les systèmes nucléaires.

APOLLO3® (87) est un projet commun du CEA, AREVA et EDF, pour le développement d'une nouvelle génération de systèmes pour l'analyse de la physique des réacteurs. Les contraintes liées à la R&D et celles de l'industrie ont été prises en compte pour le design de l'architecture système d'APOLLO3®. APOLLO3® est la poursuite du développement de la famille de codes APOLLO2, CRONOS2 et ERANOS2. L'expérience sur ces précédents codes fournit une base initiale complète de cheminements des calculs pour l'évaluation neutronique. Par cette expérience, des manières d'améliorer les modèles existants ont pu

être établies, telles que l'amélioration des solveurs de flux, de nouvelles accélérations pour la radioprotection ou encore de nouvelles manières de paralléliser efficacement les méthodes.

Pour atteindre ces objectifs, les principales qualités d'APOLLO3® sont les suivantes :

- Flexibilité : des calculs avec estimations très fines jusqu'au design de modèles industriels
- Possibilité de couplage avec les codes provenant d'autres disciplines (thermomécanique, thermo-hydraulique) via la plateforme logicielle SALOME (88)
- Couplage facilité avec les codes Monte Carlo au travers d'une interface de données partagées
- Un domaine d'application étendu : criticité, modélisation de tous types de réacteurs (PWR, BWR, GFR, SFR, SCWR, …)
- Calculs d'incertitudes avec des méthodes de perturbations ou avec des méthodes non intrusives (cf. plateforme URANIE (89))
- Possibilité d'effectuer des calculs sur des machines parallèles
- Adapté à l'utilisateur : interface utilisateur facile à appréhender, bases de données, …
- Portabilité du code

6.2. Présentation du solveur MINOS

Le solveur MINOS du code APOLLO3® résout l'équation de transport de Boltzmann appliquée au transport de neutrons par la méthode de la diffusion ou du transport simplifié. En résolvant cette dernière, il est possible de calculer la criticité d'un cœur de réacteur, c'est-à-dire si les réactions au sein de ce derniers sont stables (critique), augmentent (surcritique), ou diminuent (sous-critique). Le terme caractérisant la criticité est appelé k-eff pour k-effective ratio. Il est utilisé pour décrire le nombre de neutrons produits à un pas de temps, et il peut être décrit par :

$$k_{eff} = \frac{production}{absorption + fuite}$$

Ainsi, le k-eff représente la proportion de neutrons produits sur l'absorption et les fuites dans le milieu. La production dépend de la masse et le type de matière fissile, mais également des conditions de modérations. L'absorption dépend des poisons neutroniques présents ou injectés dans le milieu (bore, …) et d'autres facteurs. La fuite, elle, dépend de la densité de la matière fissile, la forme géométrique, les conditions de réflexion et des intéractions. Les états sous-critiques, critiques et sur-critiques sont caractérisés par un k_{eff} respectivement inférieur à 1, égal à 1 ou supérieur à 1.

Le problème de diffusion critique est un problème à valeur propre, dans lequel on recherche le flux et le courant associés à la plus grande valeur propre k_{eff}. La méthode de la Puissance est de ce fait souvent appliquée pour résoudre ce problème, car elle assure la convergence vers la valeur propre dominante, au prix d'une convergence parfois lente. D'autres méthodes offrent une convergence meilleure, sans assurer qu'elle soit vers la valeur propre dominante. La méthode d'Arnoldi ou de Jacobi-Davidson en sont quelques exemples.

6.2.1. Méthode des puissances

La méthode utilisée pour résoudre ce problème à valeur propre est un algorithme itératif sur le vecteur source. A chaque itération, un problème à source est résolu, puis la source et la valeur propre sont mises à jour pour l'itération suivante. La problématique à résoudre s'écrit sous la forme du problème à valeur propre généralisé suivant :

$$HU = \frac{1}{keff} FU$$

Pour plus de détails sur les termes H, U et F, voir la thèse de Pierre Guérin à la page 29, ou encore (25). Supposons que l'on fixe un nombre d'itérations maximum Next, et un critère d'arrêt faisant intervenir une borne ε. L'algorithme des puissances pour résoudre le problème est le suivant :

Tableau 31. Algorithme des Puissances utilisé dans le Solveur MINOS.

Initialisation du vecteur U1 et de la valeur propre k1
Calcul de la source normalisée $S_1 = FU_1 / k_1$

{Itérations externes}
pour n = 1 à Next faire

1. Résolution de $HU_{n+1} = S_n$
2. Mise à jour de la source : $S_{n+1} = FU_{n+1}$
3. Calcul de la nouvelle valeur propre : $k_{n+1} = \frac{||S_{n+1}||_2^2}{(S_{n+1},S_n)}$
4. Calcul de la source normalisée : $S_{n+1} = \frac{S_{n+1}}{k_{n+1}}$
5. Mise à jour de l'erreur : Err $= \frac{||S_{n+1} - S_n||_2}{||S_{n+1}||_2}$

   Si Err < ε on sort de la boucle

fin pour

6.2.2. Algorithme de MINOS

Cet algorithme est une idée des étapes importantes à accomplir pour effectuer le calcul du keff. L'implémentation effective du solveur MINOS est présentée dans le tableau 32.

Tableau 32. Algorithme du solveur MINOS dans le code APOLLO3®.

Initialisation de la source S0 et des fuites directionnelles J0

{Itérations externes}
pour n = 1 à Next faire
  {Itérations internes}
  pour m = 1 à Nint faire
    {Itérations sur les directions}
    pour d = 1 à Nd faire
      Inversion du système : $W_d P_d^n =$ BdT^{-1} [S^{n-1} – $\sum_{d'<d} J_{d'}^n - \sum_{d'>d} J_{d'}^{n-1}$]
      Actualisation du vecteur de fuite : $J_d^n = B_d^T P_d^n$
    fin pour
  fin pour

  Mise à jour du flux Fn
  Mise à jour de la source de fission Sn = $\sum_f F_n$
  Calcul de la nouvelle valeur propre : $k_{eff}^n = \frac{(S_n,S_n)}{(S_n,S_{n-1})}$
  Normalisation de la source : Sn = $\frac{Sn}{k_{eff}}$
  Si les critères de convergence sur Sn et kn eff sont atteints, on sort de la boucle
fin pour

6.3. Problématique d'intégration du framework parallèle

Le choix du framework parallèle présenté à la section 4.1.1 a été en partie dicté par la structure des solveurs du code APOLLO3®, et notamment par le solveur MINOS qui a servi d'exemple. Lors du début de la thèse, une première accélération du code a été effectuée, avec des modifications profondes dans le code. L'ensemble des vecteurs contenant des données a été ainsi transformé en tableaux dynamiques adressés par pointeurs. Par exemple, une matrice représentée par :

vector< vector < float> > matrice

devenait :

vector< float * > matrice

Ce code était ainsi complètement tourné vers l'utilisation de GPUs, et il n'était plus envisageable d'utiliser ce solveur avec un processeur x86 classique. Ce « toy-model » nous a permis de rapidement expérimenter l'utilisation d'un accélérateur au sein de MINOS, et d'obtenir un speed-up de l'ordre de 20x entre une station de travail à base de processeur Intel Harpertown et un GPU de calcul de même génération. Les performances étaient très satisfaisantes, car proche du speed-up théorique que l'on attendait de la différence de matériel comme le montre le tableau 23. Ces calculs ont ensuite été exécutés sur processeur Intel Nehalem, dont la bande passante mémoire a fortement augmenté. Le résultat est une performance améliorée sur processeur x86, et un speed-up ramené à 7,5x.

L'intérêt des GPUs pour le calcul scientifique, déjà illustré dans les chapitres précédents, a donc pu être mis en pratique dans le cadre de ce toy-model. Cependant, la finalité des travaux de cette thèse étant une intégration industrielle de l'accélération GPU, le fait d'avoir deux codes différents introduit une maintenabilité difficile. A titre d'exemple, durant le développement de ce toy model, des fonctionnalités ont entre-temps été ajoutées au solveur MINOS par l'équipe de R&D, fonctionnalités dont la version accélérée par GPU ne disposait pas. Il aurait donc fallu un deuxième développement, augmentant ainsi les risques d'erreur de programmation et les coûts de développement.

Le framework développé précédemment permet de dissimuler la différence de matériel au scientifique développant dans le solveur. Un seul code existe, avec un seul algorithme à gros grain. Les seuls éléments changeants sont le nom du conteneur vecteur, qui offrent la même interface que les vecteurs classiques, et les opérations de calculs de type BLAS. Le risque de différence de code au niveau de l'algorithme est ainsi déplacé dans les noyaux de calculs élémentaires sur les vecteurs. La maintenabilité du code est facilitée, avec cependant un potentiel impact sur les performances de l'accélérateur. C'est une problématique connue : essayer de garder un code générique qui adresse plusieurs architectures ou programmation différentes de manière efficace.

Concernant l'intégration effective, il faut retenir que le choix de l'accélération ou non se fait lors de la compilation du projet. Une directive de compilation, appelée GPU_ACCELERATION_ON est passée au compilateur si le GPU est nécessaire aux calculs. Au sein d'un fichier d'en-tête « Acceleration.hxx » est placée une garde à base de macro-définition, qui effectue le choix du type de conteneur vecteur :

Fichier d'en-tête « Acceleration.hxx »

```
#ifdef GPU_ACCELERATION_ON
    #include « GPU_Vector.hxx »
    #define AcceleratedVector GPU_Vector
#else
    #include <vector>
    #define AcceleratedVector std ::vector
#endif
```

Grâce à ce fichier d'en-tête, tout « AcceleratedVector » présent dans le programme sera automatiquement remplacé par un vecteur de la STL ou un vecteur GPU. Pour illustrer cette mécanique et les changements apportés au solveur, prenons comme exemple la méthode de la puissance dans sa forme la plus simple, que nous implémenterons en C++ :

```
#include « Acceleration.hxx »
#include « BLAS.hxx »

Template <class T> Matrice
{
    vectorAcceleratedVector <T> data ;
    /* … */
    void matrix_vector( const vectorAcceleratedVector<T> & v_mul, vectorAcceleratedVector<T> & v_res)
    {
        BLAS ::gemv( this->data, v_mul, v_res) ;
    }
} ;

Void Puissance(const Matrice<T> & A, vectorAcceleratedVector<T> & v, T & lambda)
{
    Do
    {
        1. A.matrix_vector(v,v)
        2. lambda_old = lambda
        3. lambda = BLAS::norm2(v)
        4. BLAS ::scal(v, 1/lambda )
        5. err = |λ – λ_old|
    } while ( iteration < MAX && err > epsilon) ;
}
```

On voit que les changements apportés à haut niveau sont relativement contenus. Seules les structures contenant des données sont modifiées, en apparence. En réalité, les modifications sont plus importantes dans les opérations BLAS et peuvent simplement se présenter sous la forme de surcharge. BLAS.hxx contient tout comme Acceleration.hxx une garde permettant la déclaration des fonctions nécessaires aux GPUs seulement lorsque l'accélération est activée. Cela permet au code d'être indépendant de la présence d'accélérateurs ou non, ce qui est une qualité importante dans un cadre industriel. De plus, la dépendance aux GPUs est limitée aux noyaux numériques les plus élémentaires, ce qui facilite la maintenabilité. De manière générale, ce type de noyau est très stable dans le temps, et en cas de modification de gemv ou norm2 sur CPU, alors les changements seront à effectuer dans les fonctions idoines sur GPUs. Ainsi le fichier BLAS.hxx pourrait être structuré de la manière suivante :

```
#include « Acceleration.hxx »
namespace BLAS
{
    // Implémentation CPU
    Template <class T>
    void gemv(const std ::vector<T> & matA, const std ::vector<T> & v_mul, std ::vector<T> & v_res)
    { /* code CPU, appel à BLAS standard, … */}

    Template <class T>
    void norm2(const std ::vector<T> & v) { /* code CPU, appel à BLAS standard, … */}

    Template <class T>
    void scal(std ::vector<T> & v, const T & alpha){ /* code CPU, appel à BLAS standard, … */}

#ifdef GPU_ACCELERATION_ON
    // Implémentation GPU
    Template <class T>
    void gemv(const GPU_Vector<T> & matA, const GPU_Vector<T> & v_mul, GPU_Vector<T> & v_res)
    { /* code GPU, appel à CUBLAS, … */}

    Template <class T>
    void norm2(const GPU_Vector<T> & v) { /* code GPU, appel à CUBLAS, … */}

    Template <class T>
    void scal(GPU_Vector<T> & v, const T & alpha){ /* code GPU, appel à CUBLAS, … */}
#endif
}
```

6.4. Utilisation de multiples GPUs

Nous avons vu comment utiliser un GPU à la place du CPU pour effectuer certains calculs plus rapidement au sein de MINOS, et nous allons maintenant étudier le cas de plusieurs GPUs appelés dans le même calcul. Ce travail se base sur la décomposition de domaine introduite dans MINOS par Pierre Guérin (90). Cette décomposition de domaine permet une répartition des calculs sur plusieurs processus MPI, c'est-à-dire en mémoire distribuée et par extension (voir section 1.2.1) en mémoire partagée. L'utilisation d'accélérateurs est orthogonale à cette décomposition, et seuls quelques ajustements sont nécessaires à cause de l'adressage mémoire différents des GPUS.

En effet, le transfert de données entre GPUs requiert pour le moment des transferts intermédiaires en mémoire principale, comme la figure 52 le montre. Si le GPU x processus X doit envoyer une information au GPU y processus Y, alors trois transferts sont nécessaires :

1. GPU x vers la mémoire principale du processus X(CUDA)
2. Du processus X vers le processus Y (MPI)
3. De la mémoire principale du processus Y vers le GPU y (CUDA)

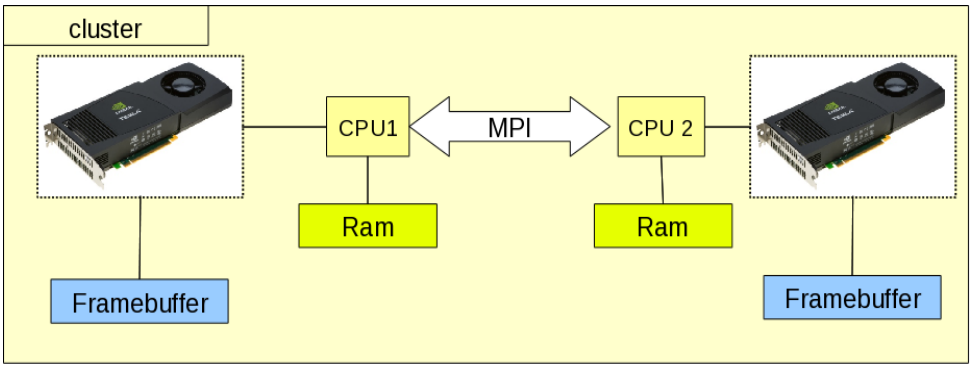


Figure 52. Échange de données entre deux GPUs dans un environnement distribué utilisant MPI. Les échanges passent nécessairement par la mémoire principale de chacun des CPUs.

Par rapport à une exécution pure CPU, il y a donc un temps de transfert supplémentaire, qui peut impacter les calculs comme nous le verrons par la suite. Par défaut, ces transferts ont lieu en deux étapes :

1. Le driver CUDA copie une partie des données de la mémoire principale vers une mémoire spéciale réservée à ce dernier.
2. Les données copiées dans cette zone spéciale sont copiées vers le GPU

Dans le sens GPU vers CPU, c'est la même chose qui a lieu, dans l'autre sens : GPU vers zone spéciale puis vers la mémoire principale. Ce phénomène est appelé « double copy ». Pour optimiser ces transferts, nous pouvons allouer notre propre mémoire spéciale dite « Pinned memory ». Ce type de mémoire, alloué dans la mémoire principale, a la particularité de posséder une adresse fixe de son allocation à sa libération. Cela permet au driver vidéo d'aller directement accéder à des données dans cette dernière pour les transférer via le PCI-Express à la mémoire du GPU. L'utilisation de ce type de mémoire permet donc au mieux de diviser par deux le temps des transferts mémoire. Nous employons cette technique pour optimiser les transferts inter-GPUs dans MINOS. La figure 53 montre l'exécution parallèle du solveur MINOS sur deux sous-domaines, avec les échanges de mémoires entre sous-domaines, et donc entre GPUs lorsque ces derniers sont utilisés.

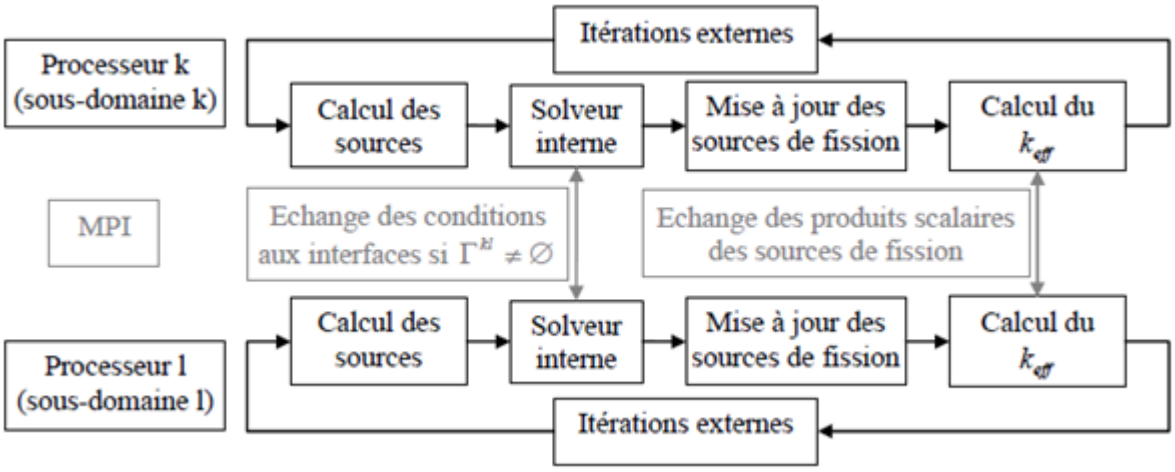


Figure 53. Schéma de la parallélisation du solveur MINOS avec décomposition de domaines.

6.5. Performances atteintes

Au moment de ces travaux de thèse, une petite partie des calculs utilisant le solveur MINOS ont été paramétrés pour la parallélisation MPI, et le cas de référence que nous utiliserons se nomme Hete3D. Ce calcul simule un cœur de réacteur à eau pressurisé de 900MWe en trois dimensions. 2 groupes d'énergie sont utilisés, avec un maillage de 289x289x60 mailles. Le calcul est réalisé cellule par cellule, et les résultats sont obtenus à partir de la machine Titane. Ces résultats ont été présentés dans (91; 92; 93).

6.5.1. Accélération avec un seul GPU

Les calculs avec un seul GPU ont été exécutés avec trois approximations différentes : Diffusion, et SP_1 puis SP_3. Chaque niveau d'approximation demande plus de temps de calcul, comme on peut voir sur la figure 54.

6.5.1.1. Performance

Globalement, les facteurs d'accélération en faisant appel au GPU sont de 5,11x, 5,38x et 4.13x pour la diffusion, SP_1 et SP_3.

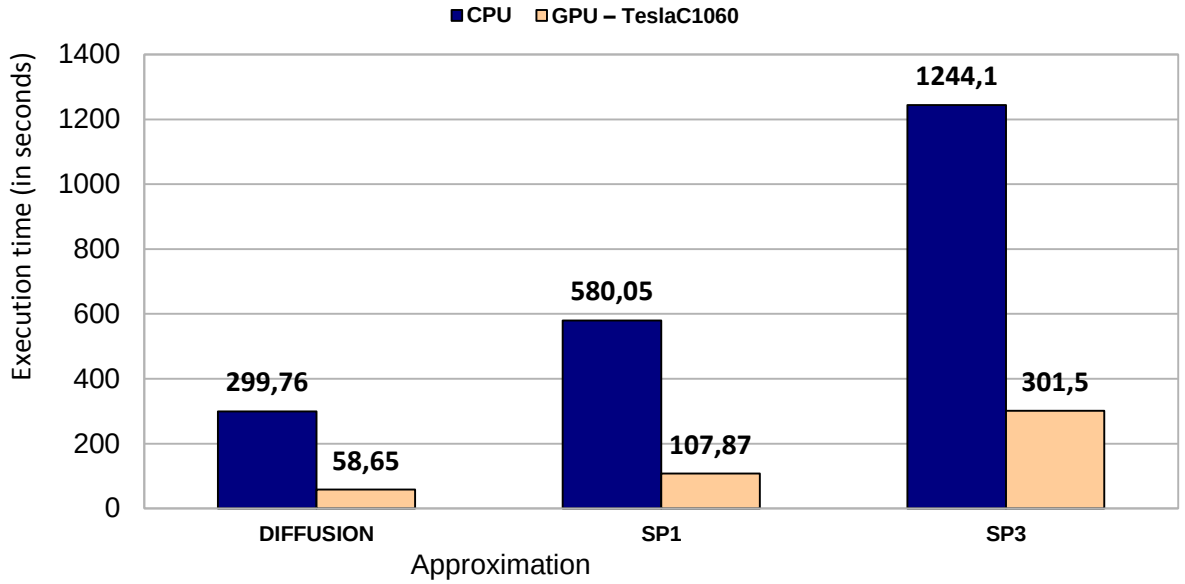


Figure 54. Calculs MINOS avec un seul processeur ou GPU pour Hete3D.

Cela correspond à peu près à la différence de bande passante mémoire entre les Nehalems de Titane et les GPUs C1060 de Nvidia, décrite dans le tableau 23, page 67. De manière intéressante, d'autres tests ont été exécutés, et illustrent les faiblesses des GPUs selon les cas d'utilisation. Par exemple, l'exécution d'un test portant sur le calcul de contre réaction d'un quart de cœur REP. Ce test très rapide demande environ 0.01s sur CPU. Ce test est de taille très réduite, et son exécution sur GPU s'entraine d'un fort ralentissement : le temps augmente alors à 20s. Lors de ce test, de nombreux noyaux de taille très réduite sont lancés sur le GPU, avec une performance proportionnellement très faible. Sur CPU, ces calculs tiennent intégralement dans les caches du processeur, et la performance y est donc très grande. Pour rappel, la latence dans les caches du processeur est de moins de 1 à 10 ns, et celle du bus PCI-Express nécessaire pour chaque exécution de noyau CUDA est de 15 µs. Le facteur entre les deux est de 1500x à 15000x et l'écart mesuré entre la performance CPU et GPU est évalué à environ 2000x, ce qui correspond assez bien.

Un autre exemple défavorable aux GPUs a été l'exécution de MINOS sur des maillages hexagonaux. L'utilisation de ces derniers entraîne quelques calculs supplémentaires par rapport aux maillages cartésiens. Parmi ces calculs, l'un des noyaux est difficilement parallélisable sur GPU, car il possède de nombreuses dépendances de données et des accès ponctuels irréguliers en mémoire globale du GPU. Au final, une itération du solveur est 3x plus lente sur GPU que sur CPU à cause de ces raisons.

6.5.1.2. Précision

Le résultat obtenu lors des expérimentations était complètement identique entre CPU et GPU : même nombre d'itérations du solveur et mêmes valeur propre et flux calculés. Ces bons résultats ont été atteints avec des GPUs supportant la double précision. Lors des premières expérimentations sur GPU, cette dernière n'était pas disponible, alors qu'elle est nécessaire lors du calcul de la norme du flux, qui effectue un produit scalaire utilisant un accumulateur double. Sans double précision, la perte de précision est importante, et le solveur ne converge pas vers la précision souhaitée. Ainsi il avait été nécessaire de transférer le vecteur contenant le flux du GPU vers la mémoire principale pour que le CPU effectue le calcul en double précision, puis de retransférer ce dernier vers le GPU ensuite. La pénalité de cette solution sur la performance était importante : une accélération divisée par deux par rapport à la solution sans double transfert.

6.5.2. Accélération multi-GPUs

Le cas Hete3D est maintenant exécuté avec l'approximation de diffusion sur plusieurs processeurs et plusieurs GPUs sur la machine Titane. Nous utilisons de 1 à 128 unités de calcul pour autant de sous-domaines. La figure 55 montre l'évolution du temps de calcul de MINOS. Pour les GPUs, nous n'avons pas pu utiliser 128 cartes malgré la présence de 192 GPUs au total. La raison est simple : les GPUs sont reliés par deux à chaque nœud de calcul de Titane, au travers d'un seul lien PCI-Express. Autrement dit, et nos expérimentations l'ont montré, utiliser deux GPUs par nœud pour l'exécution de MINOS entrainait des performances dégradées. Au maximum, nous ne pouvions donc utiliser que 86 GPUs. Or, la décomposition de domaines de MINOS est réglée pour 64 sous-domaines puis 128, sans découpage intermédiaire.

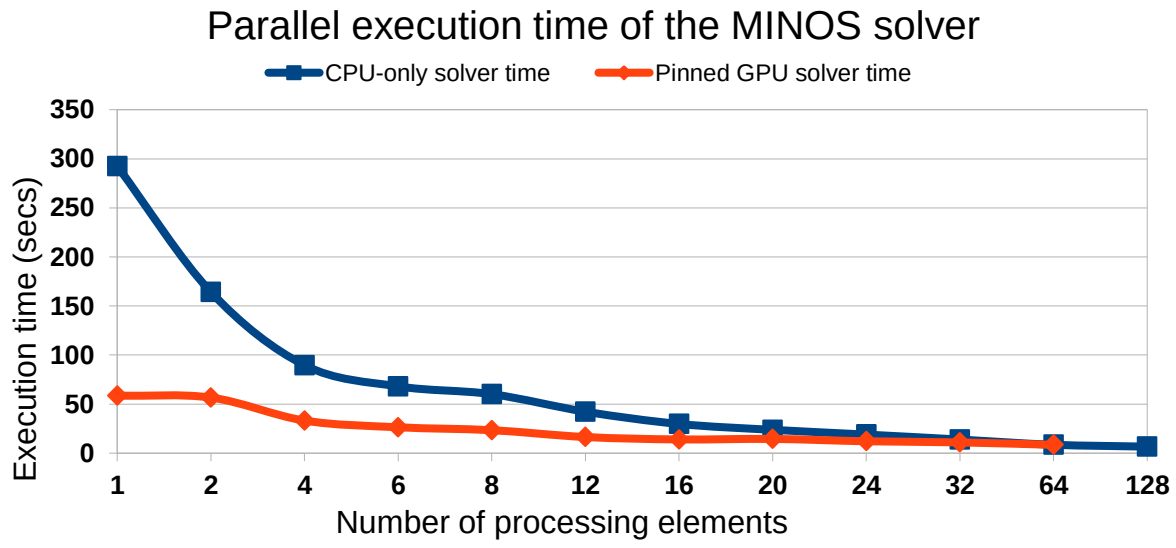


Figure 55. Calcul MINOS exécuté avec décomposition de domaine sur plusieurs processeurs et GPUs.

La performance CPU tend à bien se comporter, diminuant de manière régulière à l'ajout d'unités de calculs. Sur GPU, le temps reste constant pour l'utilisation d'un ou deux accélérateurs. Cette constance dans le temps d'exécution est due aux transferts mémoires CUDA entre GPU et CPU et vice-versa. En effet, nos mesures montrent que 50% du temps de calcul multi-gpus est passé dans les communications inter-GPUs. A partir de deux GPUs, la scalabilité de MINOS se réduit, comme nous le montre le calcul de l'efficacité à partir de deux unités de calculs :

	2	4	6	8	12	16	20	24	32	64	128
Eff CPU	100%	92%	80%	68%	65%	69%	70%	72%	74%	60%	39%
Eff GPU	100%	88%	74%	62%	60%	56%	41%	40%	34%	22%	-

Ce comportement ressemble très fortement à celui présenté dans les chapitres précédents, lors des expérimentations avec ERAM : lorsque le cas d'une taille fixe est décomposé en plusieurs sous-domaines, alors chaque sous-domaine traite un problème de plus en plus petit. Le GPU fournit alors de moins en moins de performance, et ce phénomène ajouté au coût des communications implique une bien moins bonne scalabilité. L'utilisation de 64 processus MPI en mode CPU donne un temps de 8,5s et celle de 64 GPUs un temps de 8,43s à 128 processus MPI, les CPUs permettent un temps d'exécution de 6,6s. Nous n'avons pas pu exécuter le code sur 128 GPUs, mais en projetant le temps de 128 GPUs avec une efficacité de 20%, on obtient 4,53 s. Si l'on prend une efficacité de 15% par rapport à deux GPUS, alors le temps serait de 6 secondes.

6.6. Extension à d'autres solveurs

Dans le cadre d'un Grand Challenge organisé entre le Direction de l'énergie Nucléaire et la Direction des Applications militaires, un accès à la machine Tera100 en cours d'installation en 2010 a été accordé. Le solveur MINARET a été sélectionné pour tenter une exécution massivement parallèle sur cette machine. Cette dernière est constituée de nœuds quadri-processeurs Intel Nehalem EX à 8 cœurs associés à 64 Go de mémoire vive. La machine totalise 4324 nœuds de calcul pour un total de 138368 cœurs. Cette machine fut le premier supercalculateur européen à franchir la limite du PetaFlops.

6.6.1. Le solveur de Transport Sn MINARET

MINARET (92)[1] est un solveur du code APOLLO3® qui résout l'équation de transport de Boltzmann multigroupe appliquée aux neutrons ou gammas. Il vise à effectuer des calculs 3D transport avec les approximations SPN ou SN sur des géométries non structurées cylindriques. Le code est basé sur une conception orientée objet et le langage de programmation C++, et notamment les conteneurs vecteurs de la STL C++.

Le maillage du domaine physique de la géométrie est un ensemble de triangles conforme dans la direction radiale. La direction axiale est divisée en plans.

La discrétisation spatiale est basée sur une approximation éléments finis discontinus de type Galerkin. Les approximations spatiales sont P0 (constante par morceaux) ou P1 (degré 1 en x, y et z). La continuité du flux angulaire sur chaque triangle ou (élément prismatique pour un calcul 3D) n'est pas imposée. La variable angulaire est discrétisée par la méthode des ordonnées discrètes, pour laquelle différentes formules sont disponibles (niveau symétrique, formule « equiweight » produit). L'algorithme de résolution est un balayage sur l'ensemble des directions angulaires. Pour une direction donnée, le flux angulaire sur un triangle donné est calculé dès que toutes les contributions des sources lumineuses venant des frontières de tous les triangles voisins sont connues. Compte tenu de l'indépendance de

chaque balayage directionnel, un algorithme parallèle a été conçu : chaque processeur prend en compte un ensemble de directions et une émission. Une réduction est effectuée pour les sources, avant et après le processus de balayage. Le synoptique de l'algorithme parallèle est présenté dans la figure 56. L'implémentation parallèle à gros grain a été réalisée en utilisant la bibliothèque d'échange explicite de messages MPI.

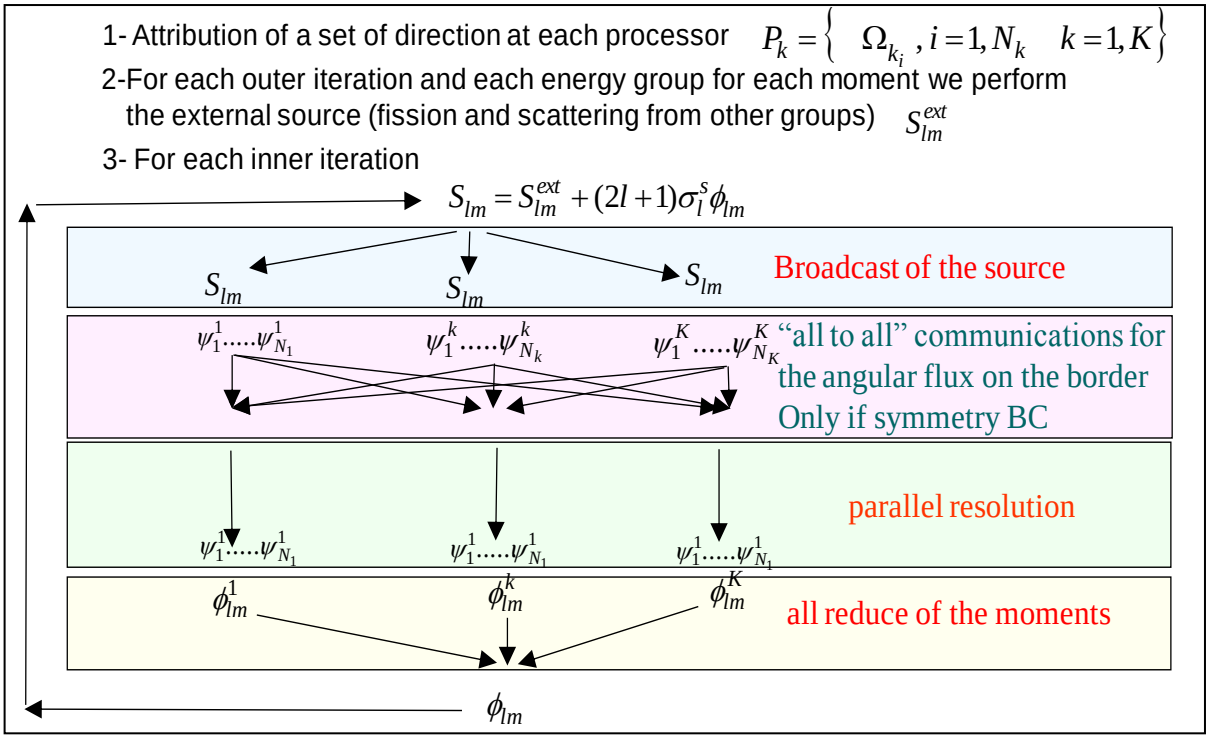


Figure 56. Synoptique de l'algorithme parallèle MINARET

6.6.2. Parallélisation grain fin

En outre suivant une direction de balayage, le calcul et la résolution des systèmes associés à chaque front peuvent se faire également de manière indépendante. Cet autre niveau de parallélisme a été réalisé grâce à une approche multiprocessus légers en mémoire partagée et implémentée en OpenMP. Nous illustrons ce principe dans la figure 57. Suivant une direction angulaire, on représente par des couleurs différentes les différents fronts, perpendiculairement à la direction angulaire, qui peuvent être calculés de manière concurrente.

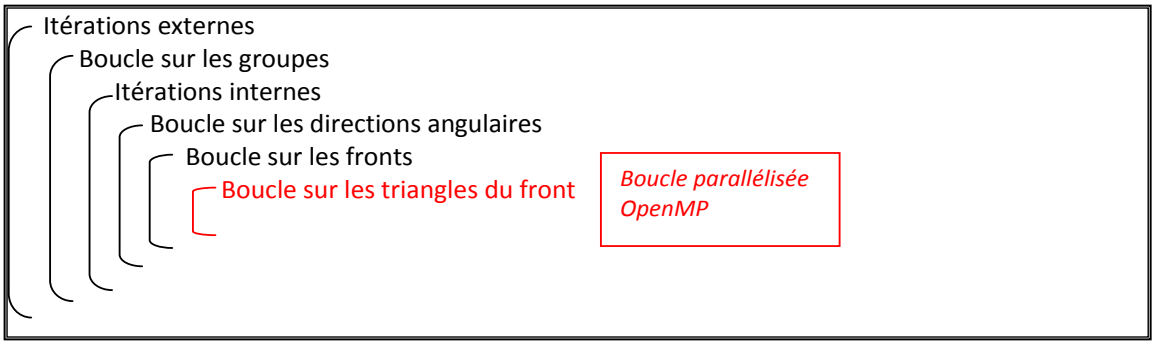


Figure 57. Parallélisation OpenMP du solveur Minaret.

En outre, la parallélisation grain fin pourrait potentiellement faire appel aux GPUs, car la résolution des triangles du front est complètement parallèle. Cependant, pour des raisons de temps, le framework parallèle n'a pu être introduit dans le solveur MINARET, et nous nous sommes concentrés sur l'ajout de la parallélisation OpenMP.

Cette dernière propose globalement une amélioration des performances par rapport à une version purement MPI, d'un facteur situé entre 0 et 10%. Cette faible amélioration est liée au temps réduit passé dans la boucle parallélisée, qui ne permet pas de profiter du nombre de threads à disposition. Ainsi, pour tirer parti des multicoeurs via OpenMP, il faudrait déplacer la parallélisation à un niveau supérieur, tel que le niveau des fronts. Le problème est qu'il existe dans la version actuelle de MINARET des dépendances qui empêchent une parallélisation directe de la boucle. Les contraintes de temps du Grand Challenge ont empêché une amélioration de la parallélisation OpenMP.

Cette amélioration aurait dans l'idéal permis à nombre de processeurs égal d'utiliser moins de processus MPI pour obtenir une performance accrue grâce à un nombre de communications réduit. Cependant, les bibliothèques MPI parallèle de BULL, le constructeur de Tera100, offrent déjà des optimisations en mémoire partagée, qui limitent l'intérêt de l'utilisation d'OpenMP d'un point de vue performance. Les expérimentations du solveur MINARET sur la machine Tera100 ont été présentées dans (94).

Conclusion

Nous avons pu utiliser avec succès le framework d'accélération dans le solveur MINOS du code APOLLO3®. Son intégration a demandé un effort modeste, et la performance obtenue est proche de celle attendue en fonction des performances crêtes des GPUs et CPUs utilisés. Ce framework prouve ainsi sa versatilité, mais il ne résout pas pour autant tous les problèmes inhérents à l'utilisation de GPUs. Ainsi, nous avons vu que l'utilisation de GPUs pour les simulations de cœurs travaillant sur des données de petites tailles se révélait désastreuse. Le facteur rencontré par rapport au CPU était de 1000 à 10000 fois plus lent. Également, lorsque l'algorithme possède des dépendances très forte et exprime peu de parallélisme, une contre-performance peut être rencontrée, comme nous avons pu constater lors de la simulation d'un cœur avec un maillage hexagonal : le GPU était trois fois plus lent. Dans un cadre où les calculs sont suffisamment grands et parallèles, alors le GPU brille avec une accélération de 4 à 5x. L'accélération multi-GPUs permet un gain de temps, modulo les communications qui sont plus importantes en quantité que lors de l'exécution de la méthode d'Arnoldi redémarrée. Ces communications impliquent une efficacité moindre lors de l'ajout de GPUs, surtout par rapport à l'utilisation de multiples CPUs. Nous avons vu que l'accélération entre un CPU et un GPU était d'un facteur 5x pour le cas Hete3D, et pour 64 GPUs ou 64 CPUs le temps d'exécution était presque le même. Cette perte d'efficacité, en partie due aux communications plus importantes, et aussi expliquée par les expérimentations multi-GPUs du chapitre 4. L'augmentation du nombre de sous-domaines s'accompagne d'une diminution de la taille de chacun d'entre eux. Les GPUs ne bénéficient pas nécessairement de cette baisse de taille à cause d'une performance moindre sur des données de petite taille, au contraire des CPUs.

Au final, l'utilisation de GPUs dans le cadre des calculs de neutronique étudiés semble indiquée pour accélérer le calcul sur une station de travail ou un petit cluster. L'utilisation pour du calcul temps réel semble également envisageable : en étant 5 fois plus performant qu'un seul processeur, le GPU permet des calculs plus précis à temps d'exécution constant, proposant donc une amélioration de la complexité des calculs dans un cadre temps réel.

Chapitre 7. Conclusion et perspectives

La précision des calculs et l'utilisation efficace du parallélisme massif des nouveaux supercalculateurs hétérogènes sont des thématiques récurrentes et critiques dans le domaine calcul hautes performances. En effet, la thématique de la précision est essentielle car elle conditionne la possibilité d'obtenir des résultats et la validité de ceux-ci. L'utilisation efficace des nouveaux moyens de calcul conditionne la performance que l'on pourra exploiter de ces derniers. Une performance maximale permettra de résoudre des problèmes plus complexes, et donc de plus grandes tailles, tout en améliorant la précision. Ainsi, au travers de cette thèse nous avons exploré ces deux thématiques par l'étude de noyaux très utilisés dans le calcul scientifique : le calcul de valeurs et vecteurs propres et la résolution de systèmes linéaires.

7.1. Synthèse

Les nouveaux matériels de calcul tels que les accélérateurs Cell et cartes graphiques n'offrent pas les mêmes résultats pour les problématiques d'orthogonalisation ou des applications industrielles en termes de précisions et de programmabilité.

À cause de la perte potentielle d'une ou deux décimales lors de calculs simple précision, le processeur Cell n'est pas un accélérateur apte à traiter des calculs sensibles à la précision. C'est pourquoi les expérimentations réalisées ont permis de déterminer que les solveurs neutroniques, pour lesquels la précision est critique, ne pourront faire appel au processeur Cell. Cependant la précision en double précision du Cell est satisfaisante, mais le faible écart entre sa performance et celle de processeurs multicoeurs classiques ne justifie pas l'investissement nécessaire en temps de développement. Cela est dû à sa programmation très spécifique et non réutilisable et à une complexité assez importante due à l'hétérogénéité de son architecture.

Les cartes graphiques proposent une interface de programmation plus accessible où pour le moment deux espaces d'adressage mémoire sont à gérer : la mémoire du processeur hôte et la mémoire de la carte graphique. Le langage proposé pour utiliser ces cartes graphiques facilite l'utilisation du parallélisme massif de ces architectures et permet d'atteindre une bonne performance avec un programme lisible et maintenable. En termes de précision, les cartes graphiques récentes et à venir offrent une précision très bonne et proche des résultats de référence d'un processeur standard. Ces résultats ont pu être produits tant sur des noyaux d'orthogonalisation que sur ces codes industriels tels qu'APOLLO3®. Nous voyons ainsi la viabilité des accélérateurs graphiques pour du calcul scientifique, et leur potentiel sur les codes futurs.

Les études réalisées sur la précision dans cette thèse se concentraient sur la simple et la double précision, mais dans un futur proche la quadruple précision sera à considérer également, notamment pour la résolution de problèmes de taille conséquente appropriés aux futures machines exaflopiques (95).

Parmi les problématiques à résoudre dans le cadre de la programmation d'architectures hétérogènes (processeurs hôtes et accélérateurs), l'une concerne l'espace mémoire séparé des différentes unités de calculs et l'autre est la différence de langage : OpenCL/CUDA pour accélérateurs ou C++/Fortran pour processeurs hôtes par exemple. Ces deux problématiques entraînent une dépendance entre la formulation de l'algorithme général

d'une application et son implémentation effective sur un accélérateur donné. Pour une bonne maintenabilité et une bonne portabilité d'une application, il est nécessaire de placer une couche d'abstraction entre le logiciel (algorithme) et le matériel (accélérateur ou non). Nous avons proposé un framework proposant d'abstraire l'utilisation d'un accélérateur ou non au sein d'un même programme. Ce framework offre de bonnes performances sur différentes familles d'accélérateurs graphiques ou différents multicoeurs dans plusieurs supercalculateurs, et son extension à d'autres est en cours. Dans le cadre de calculs creux, une technique d'auto-tuning adaptant le produit matrice-vecteur creux au GPU utilisé a permis d'augmenter la performance, pour un facteur d'accélération proche de 10. Ce framework a également pu être utilisé par une étudiante ayant quelques notions de parallélisme pour expérimenter l'algorithme du Gradient Conjugué avec préconditionnement polynomial avec à la clé des performances intéressantes sur un supercalculateur hybride. Ainsi ce framework a montré sa versatilité et sa facilité d'utilisation tout en offrant une bonne prestation sur plusieurs supercalculateurs hybrides ou non. On voit ainsi qu'il sera possible à l'avenir d'abstraire certaines notions de parallélisme et ainsi de se concentrer sur la méthode numérique. Ces résultats offrent des perspectives encourageantes pour l'utilisation de machines exaflopiques probablement hétérogènes avec des nœuds de calculs très denses.

Cependant, les méthodes itératives classiques que nous avons utilisées ont montré une limite de scalabilité sur plusieurs supercalculateurs. C'est notamment le cas de la méthode ERAM. Nous avons amélioré les performances en utilisant la méthode hybride MERAM. MERAM a été optimisée en utilisant des fonctionnalités avancées de la bibliothèque MPI2, en introduisant un collecteur de données, et avec l'autotuning du redémarrage de la méthode. Ces différentes améliorations ont permis d'exploiter pleinement la puissance de la partie disponible du supercalculateur européen Curie actuellement en cours d'installation. Ainsi, MERAM est une méthode numérique très riche, et est intéressante pour l'utilisation massive du parallélisme des futures machines exaflopiques, mais également en termes de possibilités combinatoires. Les paramètres d'entrées : taille de sous-espace, précision arithmétique, stratégie de redémarrage, utilisation d'accélérateur ou non, permettent autant de choix à explorer pour la résolution de problèmes à valeurs propres massif sur supercalculateurs exaflopiques

Les expérimentations précédentes ont permis de montrer que le framework pour utiliser efficacement les accélérateurs était robuste et offrait de bonnes performances. Nous avons ainsi expérimenté l'utilisation de ce framework dans un cadre industriel au sein du code de physique des réacteurs APOLLO3. Le solveur MINOS a été accéléré sur plusieurs dizaines de GPUs pour une accélération d'un facteur 2 à 5. Ces résultats permettraient à terme une simulation temps réel deux à cinq fois plus complexe que celle actuellement proposée, ou des temps de calcul fortement réduits. Le solveur MINARET a pu lui profiter de l'expertise développée en multithreading pour réduire ses communications parallèles. Les mêmes améliorations que celles apportées au solveur MINOS semblent possibles par la suite.

7.2. Perspectives

Les travaux développés durant cette thèse ouvrent la voie à plusieurs pistes intéressantes. L'hybridation de la méthode ERAM par MERAM permet un niveau de parallélisme supplémentaire : les co-méthodes. Ce grain de parallélisme est assez large, et les communications inter-méthodes sont peu coûteuses. Cela permettrait par exemple de couvrir le lien lent qu'il y a actuellement entre accélérateur et processeur hôte, ou même entre accélérateurs. L'idée serait d'exécuter par exemple un ERAM sur des processeurs multicoeurs standards uniquement, et un ou plusieurs autres sur des accélérateurs uniquement. Ainsi, on pourrait bénéficier de la puissance de calcul cumulée de tous les matériels, avec une accélération numérique à la clé.

Pour calculer un problème à valeur propre dominante, tel que le solveur MINOS le fait par exemple, MERAM pourrait être une solution intéressante, qui à priori convergerait plus rapidement (96). Cependant, la méthode d'Arnoldi utilisée pour ERAM ou MERAM n'assure pas nécessairement la convergence vers la valeur propre dominante, alors que la méthode de la puissance le fait. La structuration de MERAM proposée utilise un collecteur de données et nous branchons actuellement plusieurs ERAM sur ce collecteur. Il serait ainsi possible de brancher une ou plusieurs méthodes de la puissance, qui intégrerait les résultats des autres méthodes s'ils sont meilleurs. La convergence de la méthode de la Puissance serait ainsi améliorée par MERAM, et l'on aurait la certitude d'obtenir la valeur propre dominante une fois la convergence atteinte.

Les stratégies de redémarrage de MERAM pourraient également être améliorées. Actuellement, tous les ERAMs tentent de calculer les mêmes valeurs propres, et seules les tailles de sous-espace changent. Une idée pourrait être de leur demander de calculer différentes valeurs propres, et de combiner les informations différemment sur chaque ERAM, pour améliorer numériquement la solution finale demandée. Ce type de méthode s'applique déjà pour accélérer la résolution de systèmes linéaires en utilisant les valeurs propres les plus faibles du système à résoudre (97).

La taille de sous-espace pourrait également être optimisée dans le cadre d'un auto-tuning de la méthode tel que présenté dans (98). Ce principe serait appliqué à chaque ERAM en parallèle, qui ferait évoluer séparément sa taille de sous-espace. Leur précision intrinsèque serait ainsi meilleure, menant à une amélioration globale de MERAM. À cette adaptation automatique du sous-espace pourrait être jointe un choix de la précision flottante (simple, double ou quadruple) en fonction des résidus observés à chaque itération. Cela permettrait un gain de temps d'exécution pour les ERAMs concernés et probablement au niveau du MERAM global.

Également, MERAM pourrait être branchée au sein du solveur MINOS, et potentiellement apporter une accélération à la fois numérique et matérielle, permettant au solveur d'utiliser plusieurs milliers d'unités de calcul, alors qu'actuellement sa scalabilité ne lui permet pas de dépasser les 128 unités de calculs (92).

Un niveau supplémentaire de parallélisme pourrait enfin être utilisé au sein du code APOLLO3 : plusieurs solveurs neutroniques qui s'exécuteraient en parallèles, collaborant de la même manière qu'un MERAM. Le grain de parallélisme, encore supérieur à celui de MERAM, permettrait à priori d'atteindre une scalabilité à l'échelle de la machine Exaflopique, avec une collaboration massive de plusieurs solveurs hautement parallèles.

Table des figures

Table des tableaux

Bibliographie

1. *Computer and information science and engineering: one discipline, many specialties.* **Snir, Marc.** 3, 2011, Commun. ACM, Vol. 54, pp. 38-43.

2. *Parallel Subspace Method for non-Hermitian eigenproblems on the Connection Machine (CM2).* **Petiton, S. G.** 1, s.l. : IMACS, Juin 1992, Appl. Num. Math., Vol. 10, pp. 19-36.

3. *Characteristics of performance-optimal multi-level cache hierarchies.* **S. Przybylski, M. Horowitz, and J. Hennessy.** 1989, SIGARCH Comput. Archit. News 17, 3 .

4. **Demmel, James and Volkov, Vasily.** *LU, QR and cholesky factorizations using vector capabilities of GPUs.* 2008.

5. *A Data Parallel Scientific Computing Introduction, The Data Parallel Programming.* **Petiton, S. and Emad, N.** [ed.] G.-R. Perrin and A. Darte. s.l. : Springer Verlag, 1996, Vol. LNCS 1132.

6. *Brook for GPUs: stream computing on graphics hardware.* **Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan.** New-York : Joe Marks, 2004. pp. 777-786.

7. **AMD.** *Compute Abstraction Layer (CAL) programming guide v2 .0.* 2010.

8. **Frederic P. Miller, Agnes F. Vandome, and John McBrewster.** *AMD Firestream.* s.l. : Alpha Press., 2009.

9. *A Standard Platform for programming Heterogenous parallel computers.* **Tim Mattson, Ian Buck, Michael Houston, Ben Gaster.** Portland, Oregon, USA : s.n., 2009. Int. conf. on Supercomputing 2009.

10. *Three-level hybrid vs. flat MPI on the Earth Simulator: Parallel iterative solvers for finite-element method.* **Nakajima, Kengo.** 2, 2005, 6th IMACS, Vol. 54, pp. 237-255.

11. *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More.* Tsukuba : s.n., 2010. 6th International Workshop on OpenMP, IWOMP 2010.

12. **Satoshi OHSHIMA, Shoichi HIRASAWA, Hiroki HONDA.** OMPCUDA : OpenMP Execution Framework for CUDA Based on Omni OpenMP Compiler. *Lecture Notes in Computer Science.* 2010, pp. 161-173.

13. **Chatelin, Françoise.** *Valeurs Propres de Matrices.* s.l. : Masson, 1988. EAN13 : 9782225809682.

14. **Ford, Brian J. and Chatelin, Françoise.** *Problem Solving Environments for Scientific Computing.* s.l. : Elsevier Science Ltd, 1987. ISBN : 0444702547.

15. **Golub, H. A. van der Vorst and G. H.** *150 years old and still alive: Eigenproblems.* s.l. : Clarendon Press, 1997. pp. 93-119.

16. **Wilkinson, J. H.** *The Algebraic Eigenvalue Problem (Monographs on Numerical Analysis).* s.l. : Oxford University Press, USA, 1988. ISBN-10: 0198534183.

17. **E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.** *LAPACK Users' Guide.* Philadelphia : SIAM, 1999.

18. **Ward, Yihua Bai and Robert C.** A parallel symmetric block-tridiagonal divide-and-conquer algorithm. *ACM Trans. Math. Softw.* 2007, Vol. 33, 4.

19. **Lascaux, Patrick and Théodor, Raymond.** *Analyse numérique matricielle appliquée à l'art de l'ingénieur.* s.l. : Dunod, 2004. Tome 2 - Méthodes itératives. EAN13 : 9782100484294.

20. *Certification of Algorithm 254: Eigenvalues and Eigenvectors of a real symmetric matrix by the QR method.* **Welsch, John H.** 6, 1967, Commun. ACM, Vol. 10, pp. 376-377.

21. *Limit orbits of a power iteration for dominant eigenvalue problems.* **Leader, Jeffery J.** 4, 1991, Applied Mathematics Letters, Vol. 4, pp. 41-44.

22. **Meurant, Gerard.** *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations (Software, Environments, and Tools).* s.l. : SIAM, 2006.

23. *Computing eigenvalues of very large symmetric matrices--An implementation of a Lanczos algorithm with no reorthogonalization.* **Jane Cullum, Ralph A. Willoughby.** 2, 1981, Journal of Computational Physics, Vol. 44, pp. 329-358.

24. *The principle of minimized iterations in the solution of the matrix eigenvalue problem.* **Arnoldi, W. E.** 1951, Quarterly of Applied Mathematics, Vol. 9, pp. 17-29.

25. *Domain decomposition methods for the neutron diffusion problem, Mathematics and Computers in Simulation.* **Pierre Guerin, Anne-Marie Baudron, Jean-Jacques Lautard.** 11, 2010, Mathematics and Computers in Simulation, Vol. 80.

26. **Gene H. Golub, Richard R. Underwood, and James H. Wilkinson.** *The Lanczos Algorithm for the Symmetric Ax =Lambda Bx Problem.* Stanford University. Stanford : s.n., 1972.

27. *The loss of orthogonality in the Gram-Schmidt orthogonalization process.* **L. Giraud, J. Langou, and M. Rozloznik.** 7, 2005, Comput. Math. Appl., Vol. 50, pp. 1069-1075.

28. *Thick-Restart Lanczos Method for Large Symmetric Eigenvalue Problems.* **Simon, Horst and Kesheng, Wu.** 2, Mai 2000, SIAM J. Matrix Anal. Appl., Vol. 22, pp. 602-616.

29. **Greub, Werner.** *Linear Algebra .* s.l. : Springer, 1975.

30. **Saad., Y.** *Iterative methods for sparse linear systems.* New York : Publishing Company, 199-.

31. **Saad., Youcef.** *Numerical solution of large nonsymetric eigenvalue problems.* s.l. : Comput. Phys. Commun., 1989.

32. *Deflation Techniques for an Implicitly Restarted Arnoldi Iteration.* **Sorensen, R. B. Lehoucq and D. C.** 4, Octobre 1996, SIAM J. Matrix Anal. Appl., Vol. 17, pp. 789-821.

33. *Basic Linear Algebra Subprograms for FORTRAN usage.* **C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh.** 1979, ACM Trans. Math. Soft., Vol. 5, pp. 308-323.

34. *An extended set of FORTRAN Basic Linear Algebra Subprograms.* **J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson.** 1988, ACM Trans. Math. Soft., Vol. 14, pp. 1-17.

35. **J.Dongarra, R. C.** *Automatically Tuned Linear Algebra Software.* University of Tennessee. Knoxville : s.n., 1997.

36. *High-performance implementation of the level-3 BLAS.* **Geijn, Kazushige Goto and Robert Van De.** 1, Juillet 2008, ACM Trans. Math. Softw., Vol. 35, p. 14.

37. *LAPACK: a portable linear algebra library for high-performance computers.* **E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen.** Los Alamitos : IEEE Computer Society Press, 1990.

38. **Jack Dongarra, Loic Prylli, Cyril Randriamaro, and Bernard Tourancheau.** *Array Redistribution in ScaLAPACK Using PVM.* 1995.

39. **M. E Hayder, David E. Keyes, and Piyush Mehrotra.** *A Comparison of PETSC Library and HPF Implementations of an Archetypal PDE Computation.* Institute for Computer Applications in Science and Engineering (ICASE). 1997.

40. *SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems.* **Vicente Hernandez, Jose E. Roman, and Vicente Vidal.** 3, 2005, ACM Trans. Math. Softw., Vol. 31, pp. 351-362.

41. *P_ARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures.* **D. C., Sorensen and K. J., Maschhoff.** 1996. Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization (PARA '96).

42. *An MPI Implementation of the BLACS.* **Sawyer, Vaibhav Deshpande and William.** Washington, DC : IEEE Computer Society, 1996, Third International Conference on High-Performance Computing (HiPC '96).

43. *What every computer scientist should know about floating-point arithmetic.* **Goldberg, David.** 1, 1991, ACM Comput. Surv., Vol. 23, pp. 5-48.

44. Fastest Fast Fourrier Transfor in the West. [Online] 2008. http://www.fftw.org/cell/index.html.

45. **NVIDIA Corporation.** *NVIDIA CUDA, Programming Guide Version 3.0.* [Online] 2010. [Cited: Avril 4, 2010.] http://developer.nvidia.com/object/cuda_3_0_downloads.html.

46. *Performance and numerical accuracy evaluation of heterogeneous multicore systems for Krylov orthogonal basis computation.* **Jérôme Dubois, Christophe Calvin, and Serge Petiton.** Berkeley : Springer Verlag, 2010. 9th international conference on High performance (VECPAR2010).

47. —. **Dubois, Jérôme, Calvin, Christophe and Petiton, Serge.** [ed.] J.M.L.M. Palma, et al., et al. Berlin : Springer Verlag, 2011, High Performance Computing for Computational Science -- VECPAR 2010, Vol. 6449, pp. 45-57. Lecture Notes in Computer Science. ISBN 978-3-642-19327-9.

48. *Matrix market: a web resource for test matrix collections.* **Boisvert, Ronald F., et al., et al.** 1997. IFIP TC2/WG2.5 working conference on Quality of numerical software: assessment and enhancement.

49. *Sparse matrix test problems.* **Duff, I.S, Grimes, R. G and Lewis, J. G.** 1, 1989, ACM Transactions on Mathematical Software, Vol. 15, pp. 1-14.

50. *Krylov Subspace Computation on Hybrid Multicore Platforms.* **Dubois, Jérôme and Calvin, Christophe.** Denver, Colorado, USA : s.n., 2009. SIAM Annual Meeting.

51. **Bell, Nathan and Garland, Michael.** Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. [Online] 2010. http://cusp-library.googlecode.com.

52. *Accelerating the Explicitly Restarted Arnoldi Method with GPUs using Autotuning for Matrix Vector Product.* **Dubois, Jérôme, Calvin, Christophe and Petiton, Serge.** s.l. : SIAM, 2011, SIAM Journal on Scientific Computing (SISC), Copper Mountain Special Issue.

53. *Accelerating the Explicitly Restarted Arnoldi Method with GPUs using Autotuning For Matrix Vector Product.* **Dubois, Jérôme, Calvin, Christophe and Petiton, Serge.** Copper Mountain, Colorado, USA : SIAM, 2010. Eleventh Copper Mountain Conference on Iterative Methods.

54. *Some Lessons on Hybrid High Performance Computing in Reactor Physics.* **Calvin, Christophe and Dubois, Jerome.** Vancouver, Canada : s.n., 2011. SIAM ICIAM 2011 .

55. *Improving particle filter performance using SSE instructions.* **Peter Djeu, Michael Quinlan, and Peter Stone.** Piscataway, NJ, USA : IEEE Press, 2009. IEEE/RSJ international conference on Intelligent robots and systems (IROS'09).

56. *Implementing parallel conjugate gradient on the EARTH multithreaded architecture.* **Fei Chen, K. B. Theobald, and G. R. Gao.** Washington, DC : IEEE Computer Society, 2004. IEEE International Conference on Cluster Computing (CLUSTER '04).

57. *Implementing sparse matrix-vector multiplication on throughput-oriented processors.* **Garland, Michael and Bell, Nathan.** New York, NY, USA : ACM, 2009. Conference on High Performance Computing Networking, Storage and Analysis (SC '09).

58. *MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks.* **Cappello, Franck and Etiemble, Daniel.** Washington, DC, USA : IEEE Computer Society, 2000. ACM/IEEE conference on Supercomputing (CDROM)(Supercomputing '00).

59. *Auto-Tuned Linear Algebra Computations for Krylov Methods on Multicore and GPUs.* **Jérôme Dubois, Christophe Calvin, Serge Petiton.** Reno, Nevada : SIAM, 2011. SIAM CSE2011.

60. *Autotuning of Sparse Matrix-Vector Multiplication.* **Yuji Kubota, Daisuke Takahashi.** Reno, Nevada, USA : s.n., 2011. SIAM CSE 2011.

61. *Benchmarking GPUs to tune dense linear algebra.* **Demmel, James W. and Volkov, Vasily.** Piscataway, NJ, USA : IEEE Press, 2008. ACM/IEEE conference on Supercomputing (SC '08).

62. *On the Interpretation of Top500 Data.* **Feitelson, Dror G.** 2, 1999, Vol. 13, pp. 146-153.

63. *A note on scaling the Linpack benchmark.* **Numrich, Robert W.** 4, 2007, J. Parallel Distrib. Comput., Vol. 67, pp. 491-498.

64. **Agullo, E., et al., et al.** *QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators.* University of Tennessee. 2010.

65. **Hoemmen, Mark.** *Communication-Avoiding Krylov Subspace Methods.* Berkeley, California, USA : University of California at Berkeley, 2010. Thèse de doctorat. Advisor(s) James W. Demmel.

66. *Parallel Arnoldi eigensolvers with enhanced scalability via global communications rearrangement.* **V. Hernandez, J.E. Roman, A. Tomas.** 7-8, 2007, Parallel Computing, Vol. 33, pp. 521-540.

67. **D'Azevedo, Ed and Dongarra, Jack.** *Packed Storage Extension for ScaLAPACK.* University of Tennessee. Knoxville, TN, USA. : s.n., 1998.

68. *Information grids: managing and mining semantic data in a grid infrastructure; open issues and application to geno-medical data.* **Lionel Brunie, Maryvonne Miquel, Jean Marc Pierson, Anne Tchounikine, Clarisse Dhaenens, Nouredine Melab, El Ghazali Talbi, Abdelkader Hameurlain, and Franck Morvan.** Washington, DC : IEEE Computer Society, 2003. 14th International Workshop on Database and Expert Systems Applications (DEXA '03).

69. *Experiments in running a scientific MPI application on Grid'5000.* **Stéphane Genaud, Marc Grunberg, Catherine Mongenet.** 2007, p. 354.

70. *Grid'5000: A Large Scale and Highly Reconfigurable Grid Experimental Testbed.* **Cappello, F., et al., et al.** Washington, DC, USA : IEEE Computer Society, 2005, 6th IEEE/ACM International Workshop on Grid Computing (GRID '05), pp. 99-106.

71. *Hybrid eigenvalue solver using a linear algebra framework for multi-core accelerated petascale supercomputer.* **C. Calvin, L. Decobert, J. Dubois and S. Petiton.** Savannah : s.n., 2012. 15th SIAM Conference on Parallel Processing for Scientific Computing .

72. *Toward Exascale Resilience.* **Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir.** 4, 2009, Int. J. High Perform. Comput. Appl., Vol. 23, pp. 374-388.

73. *Multiple Explicitly Restarted Arnoldi Method for Solving Large Eigenproblems.* **Emad, Nahid, Petiton, Serge and Edjlali, Guy.** 1, 2005, SIAM J. Sci. Comput., Vol. 27, pp. 253-277.

74. *An asynchronous algorithm on the NetSolve global computing system.* **Emad, Nahid, Shahzadeh-Fazeli, S. -A. and Dongarra, Jack.** 3, 2006, Future Gener. Comput. Syst., Vol. 22, pp. 279-290.

75. **Sedrakian, Ani.** *Vers une aide à la décision pour les méthodes itératives hybrides parallèles réutilisables.* s.l. : Université Pierre et Marie Curie, 2005. Thèse de doctorat.

76. *Improving Scalability with Asynchronous Hybrid Methods for non-Hermitian Eigenproblems.* **Dubois, Jérôme, Petiton, Serge and Calvin, Christophe.** Tsukuba, Japan : s.n., 2011, ICCS 2011.

77. *APOLLO2 Twelve Years Later.* **S. Loubiere, R. Sanchez, M. Coste, A. Hebert, Z. Stankovski, C. Van Der Gucht and I. Zmijarevic.** Madrid, Espagne : s.n., 1999, Int. Conf. on Math. and Computations, Reactor Physics and Environmental Analysis in Nucl. Applications, M&C 1999.

78. *APOLLO II: A user-oriented, portable, modular code for multigroup transport assembly calculations.* **R. Sanchez, J. Mondot, Z. Stankovski, A. Cossic and I. Zmijarevic.** 1988, Nucl. Sci. and Eng., Vol. 100, pp. 352-362.

79. *The ERANOS data and code system for fast reactor neutronic analyses.* **G. Rimpault, et al.** Seoul, Corée du Sud : s.n., 2002. Int. Conf. on the New Frontier of Nuclear Technology: Reactor Physics, Safety and High-Performance Computing, PHYSOR, .

80. *ERANOS 2.1: International Code System for GEN IV Fast Reactor Analysis.* **J-M. Ruggieri, J. Tommasi and J-F. Lebrat.** Reno, Nevada, USA : s.n., 2006. ICAPP '06.

81. *Algorithmic Features of the ECCO Cell Code for Treating Heterogeneous Fast Reactor Subassemblies.* **Rimpault, G.** Portland, Oregon, USA : s.n., 1995. Int. Topical Meeting on Reactor Physics and Computations.

82. *SAPHYR : A Code System from Reactor Design to Reference Calculations.* **R. Sanchez, C. Magnaud, J-J. Lautard, D. Caruge et al.** Paris, France : s.n., 2003. Int. Conf. on Supercomputing in Nuclear Applications.

83. *TRIPOLI-4: A 3D Continuous Energy Monte Carlo Transport Code.* **Diop, C.M. and al.** Marrakech, Maroc : s.n., 2007. Int. Conf. On Physics and Technology of Reactors and Applications, PHYTRA-1.

84. *DARWIN: An Evolution Code System for a Large Range of Applications.* **Tsilanizara, A. and al.** Tsukaba, Japan : s.n., 1999. Int. Conf on Radiation Shielding, ICRS-9.

85. *GALILEE: A Nuclear Data Processing System for Transport, Depletion and Shielding Codes.* **Coste-Delclaux, M.** Interlaken, Switzerland : PHYSOR, 2008. Int. conf. PHYSOR'08 Advances in Reactor Physics.

86. *Status of CONRAD, a nuclear reaction analysis tool.* **Jean, C. De Saint and al.** Nice, France : s.n., 2007. Int. Conf. on Nuclear Data for Science and Technology 2007.

87. *APOLLO3: a common project of CEA, AREVA and EDF for the development of a new deterministic multi-purpose code for core physics analysis.* **Golfier, H. and al.** , Saratoga Springs, New York : s.n., 2009. Int. Conf. on Mathematics, Computational Methods & Reactor Physics, M&C 2009.

88. **Commissariat à l'Energie Atomique et aux Energies Alternatives.** Salome: The open source integration platform for numerical simulation. [Online] 2011. http://www.salome-platform.org.

89. *Uncertainty assessments in severe accident scenario using the URANIE software.* **F. Gaudier, M. Marques, B. Spindler and B. Tourniare.** Marseille, France : s.n., 2008. 35th ESREDA Seminar.

90. *Domain decomposition methods for the neutron diffusion problem.* **Guérin, Pierre, Baudron, Anne-Marie and Lautard, Jean-Jacques.** 11, 2010, Math. Comput. Simul., Vol. 80.

91. *GPGPU Programming to Solve the Boltzman Neutron Transport Equation.* **Calvin, Christophe, et al., et al.** Seattle : s.n., 2010. 14th SIAM Conference on Parallel Processing for Scientific Computing.

92. *High Performance 3D Neutron Transport on Petascale and Hybrid Architectures within APOLLO3 Code.* **Jamelot E., Dubois J., Lautard J-J., Calvin C, Baudron A-M.** Rio de Janeiro, Brésil : s.n., 2011. M&C2011.

93. *Using GPUs and the Cell to accelerate iterative methods in neutronics.* **Dubois, J. and Petiton, S.** Lille, France : s.n., 2009. CIGIL09.

94. **R. Baron, C. Calvin, J. Dubois, J-J. Lautard, S. Salmons.** *Calcul massivement parallèle en transport 3D d'un réacteur rapide sodium avec APOLLO3.* DEN/DANS/DM2S/SERMA/LLPR, Commissariat à l'Energie Atomique et aux Energies Alternatives. Saclay : CEA Saclay, 2011. Jalon AG APOLLO3 – 2010. DEN/DANS/DM2S/SERMA/LLPR/RT/10-5031/A.

95. *Fast Quadruple Precision Arithmetic Library on Parallel Computer SR11000/J2.* **Nagai, T., et al., et al.** [ed.] Marian Bubak, et al., et al. Berlin : Springer-Verlag, 2008. 8th international conference on Computational Science, Part I (ICCS '08). pp. 446-455. DOI=10.1007/978-3-540-69384-0_50.

96. *An Arnoldi-Extrapolation algorithm for computing PageRank.* **Wu, Gang and Wei, Yimin.** 11, 2010, Vol. 234, pp. 3196-3212.

97. *A Restarted GMRES Method Augmented with Eigenvectors.* **Morgan, Ronald B.** 4, October 1995, SIAM J. Matrix Anal. Appl. , Vol. 16, pp. 1154-1171. DOI=10.1137/S0895479893253975 .

98. *A new method for accelerating Arnoldi algorithms for large scale eigenproblems.* **Dookhitram, K., Boojhawon, R. and Bhuruth, M.** 2, Octobre 2009, Math. Comput. Simul., Vol. 80, pp. 387-401. DOI=10.1016/j.matcom.2009.07.009.

Communications effectuées durant cette thèse

1. *Journaux, livres et rapports*
   - *Accelerating the Explicitly Restarted Arnoldi Method with GPUs using Autotuning for Matrix Vector Product.* **Dubois, Jérôme, Calvin, Christophe et Petiton, Serge.** s.l. : SIAM, 2011, SIAM Journal on Scientific Computing (SISC), Copper Mountain Special Issue.
   - *Performance and numerical accuracy evaluation of heterogeneous multicore systems for Krylov orthogonal basis computation.* **Dubois, Jérôme, Calvin, Christophe et Petiton, Serge.** [éd.] J.M.L.M. Palma, et al. Berlin : Springer Verlag, 2011, High Performance Computing for Computational Science -- VECPAR 2010, Vol. 6449, pp. 45-57. Lecture Notes in Computer Science. ISBN 978-3-642-19327-9.
   - **R. Baron, C. Calvin, J. Dubois, J-J. Lautard, S. Salmons.** *Calcul massivement parallèle en transport 3D d'un réacteur rapide sodium avec APOLLO3.* DEN/DANS/DM2S/SERMA/LLPR, Commissariat à l'Energie Atomique et aux Energies Alternatives. Saclay : CEA Saclay, 2011. Jalon AG APOLLO3 – 2010. DEN/DANS/DM2S/SERMA/LLPR/RT/10-5031/A.
2. *Conférences*
   - *Hybrid eigenvalue solver using a linear algebra framework for multi-core accelerated petascale supercomputer.* **C. Calvin, L. Decobert, J. Dubois and S. Petiton.** Savannah : s.n., 2012. 15th SIAM Conference on Parallel Processing for Scientific Computing .
   - *Improving Scalability with Asynchronous Hybrid Methods for non-Hermitian Eigenproblems.* **Dubois, Jérôme, Petiton, Serge et Calvin, Christophe.** Tsukuba, Japan : s.n., 2011, ICCS 2011.
   - *High Performance 3D Neutron Transport on Petascale and Hybrid Architectures within APOLLO3 Code.* **Jamelot E., Dubois J., Lautard J-J., Calvin C, Baudron A-M.** Rio de Janeiro, Brésil : s.n., 2011. M&C2011.
   - *Some Lessons on Hybrid High Performance Computing in Reactor Physics.* **Calvin, Christophe et Dubois, Jerome.** Vancouver, Canada : s.n., 2011. SIAM ICIAM 2011 .
   - *Auto-Tuned Linear Algebra Computations for Krylov Methods on Multicore and GPUs.* **Jérôme Dubois, Christophe Calvin, Serge Petiton.** Reno, Nevada : SIAM, 2011. SIAM CSE2011.
   - *Performance and numerical accuracy evaluation of heterogeneous multicore systems for Krylov orthogonal basis computation.* **Jérôme Dubois, Christophe Calvin, and Serge Petiton.** Berkeley : Springer Verlag, 2010. 9th international conference on High performance (VECPAR2010).
   - *Accelerating the Explicitly Restarted Arnoldi Method with GPUs using Autotuning For Matrix Vector Product.* **Dubois, Jérôme, Calvin, Christophe et Petiton, Serge.** Copper Mountain, Colorado, USA : SIAM, 2010. Eleventh Copper Mountain Conference on Iterative Methods.
   - *GPGPU Programming to Solve the Boltzman Neutron Transport Equation.* **Calvin, Christophe, et al.** Seattle : s.n., 2010. 14th SIAM Conference on Parallel Processing for Scientific Computing.

- *Krylov Subspace Computation on Hybrid Multicore Platforms.* **Dubois, Jérôme et Calvin, Christophe.** Denver, Colorado, USA : s.n., 2009. SIAM Annual Meeting.
- *Using GPUs and the Cell to accelerate iterative methods in neutronics.* **Dubois, J. et Petiton, S.** Lille, France : s.n., 2009. CIGIL09.

Résumé

Les travaux de cette thèse concernent dans un premier temps l'évaluation des nouveaux matériels de calculs tels que les cartes graphiques ou les puces massivement multicoeurs, et leur application aux problèmes de valeurs propres pour la neutronique. Ensuite, dans le but d'utiliser le parallélisme massif des supercalculateurs, nous étudions également l'utilisation de méthodes hybrides asynchrones pour résoudre des problèmes à valeur propre avec ce très haut niveau de parallélisme. Nous expérimentons ensuite le résultat de ces recherches sur plusieurs supercalculateurs nationaux tels que la machine hybride Titane du Centre de Calcul, Recherche et Technologies (CCRT), la machine Curie du Très Grand Centre de Calcul (TGCC) qui est en cours d'installation, et la machine Hopper du Lawrence Berkeley National Laboratory (LBNL), mais également sur des stations de travail locales pour illustrer l'intérêt de ces recherches dans une utilisation quotidienne avec des moyens de calcul locaux.

Abstract

In science, simulation is a key process for research or validation. Modern computer technology allows faster numerical experiments, which are cheaper than real models. In the field of neutron simulation, the calculation of eigenvalues is one of the key challenges. The complexity of these problems is such that a lot of computing power may be necessary.

The work of this thesis is first the evaluation of new computing hardware such as graphics card or massively multicore chips, and their application to eigenvalue problems for neutron simulation. Then, in order to address the massive parallelism of supercomputers national, we also study the use of asynchronous hybrid methods for solving eigenvalue problems with this very high level of parallelism.

Then we experiment the work of this research on several national supercomputers such as the Titane hybrid machine of the Computing Center, Research and Technology (CCRT), the Curie machine of the Very Large Computing Centre (TGCC), currently being installed, and the Hopper machine at the Lawrence Berkeley National Laboratory (LBNL). We also do our experiments on local workstations to illustrate the interest of this research in an everyday use with local computing resources.